



CODENAME ONE • MOBILE DEVELOPMENT FOR JAVA

Codename One Developer Guide

Build cross-platform mobile apps in Java, from UI to native services and deployment

iOS • Android • Desktop • Web

Shai Almog, Steve Hannah & others

Version 7.0.265 - August 7, 2026

Codename One Developer Guide

Codename One Documentation Team; Shai Almog; Steve Hannah; Chen Fishbein

Version 7.0.265, 2026-08-07

Contents

Preface . . . 1
How to contribute updates . . . 1
1. Maven Project Workflow . . . 5
1.1. Introduction . . . 5
1.1.1. Conventions . . . 5
1.2. Getting started . . . 5
1.2.1. Prerequisites . . . 5
1.2.2. Creating a new project . . . 6
Codename One initializr . . . 6
Generating a new project from the command-line . . . 6
1.2.3. Migrating an existing project . . . 9
Command line . . . 9
IntelliJ IDEA . . . 10
NetBeans . . . 10
Eclipse IDE . . . 10
Example: Migrating kitchen sink app . . . 17
1.2.4. Adding project dependencies . . . 18
Which `pom.xml` to add the `<dependency>` snippet to . . . 18
Example: Adding Google Maps dependency via Maven central . . . 19
Compatibility with Codename One . . . 20
The compliance check . . . 20
Managing add-ons in Codename One Settings . . . 20
Installing legacy cn1libs . . . 22
1.3. Updating Codename One . . . 22
1.4. Creating project templates . . . 24
1.4.1. Converting a Codename One application project into a project template . . . 24
1.4.2. Test your project template . . . 25
1.4.3. Add your template to Codename One intializr . . . 25
1.5. Codename One libraries . . . 25
1.5.1. Creating a library project . . . 26
Command line . . . 26
IntelliJ IDEA . . . 27
NetBeans . . . 30
Eclipse IDE . . . 32
Project structure . . . 36
Building the library . . . 43
Editing Java Code . . . 45
Using the library in an application project . . . 45

1.5.2. Adding menus to the simulator . . . 46
The contract . . . 46
The action method . . . 47
Worked example using cn1-bluetooth . . . 47
Calling hooks from CN1 unit tests . . . 48
What's intentionally not exposed . . . 49
1.5.3. Distributing your library . . . 49
2. Introduction . . . 51
2.1. Build cloud . . . 51
2.1.1. Why build servers . . . 52
2.1.2. How does Codename One work . . . 53
Lightweight architecture . . . 53
Why ParparVM . . . 54
Historical windows ports . . . 54
JavaScript port . . . 55
Desktop and Android . . . 55
2.2. History . . . 55
2.3. Core concepts of mobile programming . . . 56
2.3.1. Density . . . 56
2.3.2. Touch interface . . . 57
2.3.3. Device fragmentation . . . 58
2.3.4. Performance . . . 58
App size . . . 58
Power drain . . . 58
2.3.5. Sandbox and permissions . . . 59
2.4. Installing Codename One . . . 59
2.4.1. Important notes for new projects . . . 60
2.4.2. Runtime . . . 61
2.4.3. The source code of the hello world app . . . 62
2.4.4. Building and deploying on devices . . . 64
Signing/Certificates . . . 65
Build and install . . . 70
2.5. Kotlin . . . 72
2.5.1. Hello Kotlin . . . 72
3. Basics: Themes, styles, components, and layouts . . . 73
3.1. Components . . . 73
3.1.1. Layout managers . . . 74
Terse syntax . . . 78
Flow Layout . . . 78
Box Layout . . . 80
Border Layout . . . 82

Grid Layout . . . 83
3.1.2. Table Layout . . . 85
The full potential . . . 87
TextMode Layout . . . 88
3.1.3. Layered Layout . . . 89
Insets and reference components . . . 91
`auto` insets . . . 92
`%` insets . . . 92
Insets, margin, and padding . . . 93
Component references: Linking components together . . . 93
3.1.4. GridBag Layout . . . 96
3.1.5. Group Layout . . . 98
3.1.6. Mig Layout . . . 100
3.2. Themes and styles . . . 101
3.2.1. Theme . . . 102
3.2.2. Working with CSS themes . . . 103
3.3. GUI builder . . . 103
3.3.1. Hello world . . . 103
Basic usage . . . 104
Events . . . 107
Underlying XML . . . 108
3.3.2. Auto layout mode . . . 110
The inset control . . . 110
Auto snap . . . 111
Smart insets . . . 111
The widget control pad . . . 111
Keyboard Short-Cuts . . . 112
Sub-Containers . . . 112
The canvas resize tool . . . 113
4. Advanced theming . . . 115
4.1. Working with UIIDs . . . 115
4.2. Theme layering . . . 115
4.3. Override resources in platform . . . 116
4.4. Theme constants . . . 117
4.4.1. Scroll motion tuning . . . 130
4.5. Native theming . . . 131
4.6. Under the hood of the theme engine . . . 132
4.7. Understanding images and Multi-Images . . . 134
4.8. Use millimeters for Padding/Margin and font sizes . . . 135
4.8.1. Fractions of millimeters . . . 136
4.9. Creating a great looking side menu . . . 136

4.10. Converting a PSD to a theme . . . 140
4.10.1. Breaking down the PSD . . . 142
Removing the noise . . . 144
The camera Button . . . 145
4.10.2. The Code . . . 148
4.10.3. Styling the UI . . . 151
Not quite there yet . . . 153
5. Native modern themes . . . 155
5.1. Selecting a theme . . . 155
5.2. Light and dark mode . . . 155
5.3. Material 3 color palette (Android) . . . 156
5.4. iOS modern color palette . . . 157
5.5. Accent palette override . . . 158
5.5.1. Override from your app's `theme.css` (recommended) . . . 159
5.5.2. Runtime override . . . 159
5.5.3. When the override path doesn't apply . . . 160
5.6. Platform-specific UIIDs . . . 160
5.7. Switching CheckBox / RadioButton glyphs . . . 160
5.8. Component-specific tuning constants . . . 161
5.9. iOS 26 Liquid Glass tab selection morph . . . 162
5.10. Customizing in your own theme . . . 164
5.11. Inheriting from a native UIID . . . 165
5.12. Translucency, glass effects, and the test harness . . . 165
6. CSS . . . 167
6.1. Activating CSS . . . 167
6.2. Supported CSS selectors . . . 167
6.2.1. Inheriting properties using `cn1-derive` . . . 168
6.3. Special selectors . . . 168
6.3.1. `#device` . . . 168
6.3.2. `#constants` . . . 168
6.3.3. `Default` . . . 169
6.4. Standard CSS properties . . . 169
6.5. Custom properties . . . 170
6.6. CSS variables . . . 170
6.7. CSS properties . . . 171
6.7.1. Text-decoration . . . 171
6.7.2. Border . . . 172
Round borders . . . 173
6.7.3. Background . . . 175
Background images . . . 175
Gradients . . . 175

filter and backdrop-filter . . . 177
6.7.4. cn1-background-type . . . 179
6.8. Images . . . 179
6.8.1. Image DPI and device densities . . . 179
6.8.2. Multi-Images vs regular images . . . 180
6.8.3. Multi-Images as inputs . . . 181
6.8.4. Image constants . . . 181
6.9. Image recipes . . . 182
6.9.1. Import multiple images in single selector . . . 182
6.9.2. Loading images from URLs . . . 183
6.9.3. Generating 9-Piece Image borders . . . 183
6.9.4. Image backgrounds . . . 185
Example setting background Image to scale fill . . . 186
6.10. Image compression . . . 186
6.11. Fonts . . . 186
6.11.1. `font-family` . . . 186
Using bundled fonts . . . 187
Where to put the font file . . . 187
Family names, weights and styles . . . 188
Remote and GitHub-hosted fonts . . . 188
6.11.2. `font-size` . . . 189
6.11.3. `text-decoration` . . . 191
6.11.4. Some sample CSS directives . . . 191
6.12. Media queries . . . 193
6.12.1. Compound media queries . . . 195
6.12.2. Order or precedence . . . 196
6.12.3. Font scaling constants . . . 197
7. The components of Codename One . . . 199
7.1. Container . . . 199
7.1.1. Composite components . . . 199
7.2. Form . . . 200
7.2.1. Safe areas . . . 201
Automatic Safe-Area handling . . . 201
Marking your own containers as safe . . . 202
Safe-Area roots and advanced layouts . . . 202
7.3. Dialog . . . 203
7.3.1. Styling dialogs . . . 205
7.3.2. Tint and blurring . . . 205
7.3.3. Popup dialog . . . 207
Styling the arrow of the popup dialog . . . 208
7.4. InteractionDialog . . . 209

7.5. Label . . . 210
7.5.1. Label gap . . . 211
7.5.2. Autosizing labels . . . 211
7.6. TextField and TextArea . . . 212
7.6.1. Masking . . . 213
7.6.2. The virtual keyboard . . . 214
Action Button client property . . . 214
Next and done on iOS . . . 214
7.6.3. Clearable text field . . . 215
7.7. TextComponent . . . 215
7.7.1. Error handling . . . 216
7.7.2. InputComponent and PickerComponent . . . 217
7.7.3. Underlying theme constants and UIID's . . . 218
7.8. Button . . . 218
7.8.1. Uppercase buttons . . . 220
7.8.2. Raised Button . . . 220
7.8.3. Ripple effect . . . 221
7.9. CheckBox/RadioButton . . . 221
7.9.1. Toggle Button . . . 222
7.10. ComponentGroup . . . 223
7.11. MultiButton . . . 224
7.11.1. Styling the MultiButton . . . 225
7.12. SpanButton . . . 225
7.13. SpanLabel . . . 225
7.14. OnOffSwitch . . . 226
7.14.1. Validation . . . 227
7.15. InfiniteProgress . . . 228
7.16. InfiniteScrollAdapter and InfiniteContainer . . . 228
7.16.1. The InfiniteContainer . . . 229
7.17. List, MultiList, renderers & models . . . 230
7.17.1. InfiniteContainer/InfiniteScrollAdapter vs. List/ContainerList . . . 230
7.17.2. Why isn't list deprecated . . . 230
7.17.3. MVC in lists . . . 230
7.17.4. Understanding MVC . . . 231
Why is this useful . . . 232
7.17.5. Important - lists and Layout managers . . . 232
7.17.6. MultiList & DefaultListModel . . . 232
Going further with the ListModel . . . 234
7.17.7. List cell renderer . . . 234
7.17.8. Generic list cell renderer . . . 235
Custom UIID of entry in GenenricListCellRenderer/MultiList . . . 237

Rendering prototype . . . 237
7.17.9. ComboBox . . . 237
7.18. Slider . . . 238
7.19. Table . . . 240
7.19.1. Sorting tables . . . 245
7.20. Tree . . . 245
7.21. ShareButton . . . 246
7.21.1. Share Result Callback . . . 247
7.22. iOS Share Extension Authoring Helper . . . 248
7.22.1. What the helper produces . . . 248
7.22.2. Generating the bundle . . . 249
7.22.3. Reading the payload from the host app . . . 249
7.22.4. Activation rules and validation . . . 249
7.22.5. Customising the generated controller . . . 250
7.23. Tabs . . . 250
7.23.1. Animated tab indicator . . . 252
7.24. MediaManager & MediaPlayer . . . 253
7.25. ImageViewer . . . 254
7.26. ScaleImageLabel & ScaleImageButton . . . 257
7.27. Toolbar . . . 257
7.27.1. Search mode . . . 260
7.27.2. South Component . . . 261
7.27.3. Title animations . . . 261
7.28. BrowserComponent & WebBrowser . . . 263
7.28.1. Troubleshooting "Failed to create CEF browser" on Linux . . . 264
7.28.2. BrowserComponent hierarchy . . . 265
7.28.3. NavigationCallback . . . 265
7.28.4. JavaScript . . . 267
What was wrong with the old API . . . 267
The new API . . . 267
Synchronous wrappers . . . 268
Multi-use callbacks . . . 268
Passing parameters to JavaScript . . . 269
Proxy objects . . . 269
Legacy JSObject support . . . 269
The JavaScript bridge . . . 270
7.28.5. Cordova/PhoneGap integration . . . 272
7.29. Text editors . . . 272
7.30. AutoCompleteTextField . . . 272
7.30.1. Using images in AutoCompleteTextField . . . 273
7.31. Picker . . . 275

7.32. SwipeableContainer . . . 278
7.33. EmbeddedContainer . . . 279
7.34. MapComponent . . . 279
7.35. Chart Component . . . 281
7.35.1. Device support . . . 281
7.35.2. Features . . . 281
7.35.3. Chart types . . . 281
7.35.4. How to create a chart . . . 284
7.36. Calendar . . . 284
7.37. ToastBar . . . 284
7.37.1. Actions in ToastBar . . . 285
7.38. SignatureComponent . . . 286
7.39. Accordion . . . 287
7.40. Floating hint . . . 288
7.41. Floating Button . . . 288
7.41.1. Using floating Button as a badge . . . 289
7.42. SplitPane . . . 290
8. Rich Text and Code Editing . . . 293
8.1. Editor architecture and backend selection . . . 293
8.1.1. The editor package . . . 294
8.2. Focus, change events, and editability . . . 294
8.3. Platform text input and input methods . . . 295
8.4. Lightweight fields with EditField . . . 295
8.5. Selection, bidirectional text, and scrolling . . . 296
8.6. RichTextArea . . . 296
8.6.1. HTML, RTF, Markdown, and AsciiDoc . . . 297
8.7. Read-only rich text with RichTextComponent . . . 298
8.8. CodeEditor . . . 299
8.8.1. Custom syntax highlighters . . . 300
8.8.2. Asynchronous code completion . . . 301
8.8.3. Diagnostics . . . 301
8.9. Port integration checklist . . . 302
Accessibility Semantics . . . 303
9. Built-in semantics . . . 305
10. Roles, names, values, and state . . . 307
11. Ranges and editable values . . . 309
12. Custom accessibility actions . . . 311
13. Grouping and virtual descendants . . . 313
14. Traversal order . . . 315
15. Live regions and screen changes . . . 317
16. User accessibility preferences . . . 319

17. Collections . . . 321
18. Inspecting and testing semantics . . . 323
19. Diagnosing and correcting an inaccessible screen . . . 325
19.1. Assertions that prevent regressions . . . 327
19.2. Native verification . . . 328
20. Platform mappings . . . 329
MCP Headless API . . . 331
21. The MCP menu . . . 333
22. Starting the server . . . 335
23. Development builds only . . . 337
24. Built-in user interface tools . . . 339
25. Publishing application tools . . . 341
26. Screenshots and the simulator . . . 343
27. Debug logging . . . 345
28. Registering with installed hosts . . . 347
Media and Audio . . . 349
29. MediaManager and MediaPlayer . . . 351
30. PCM Audio Buffers, Mixing, and Effects . . . 353
31. Animations . . . 355
31.1. Layout reflow . . . 355
31.2. Layout animations . . . 355
31.2.1. Unlayout animations . . . 359
31.2.2. Hiding & visibility . . . 360
31.2.3. Synchronicity in animations . . . 360
Animation fade and hierarchy . . . 361
31.2.4. Sequencing animations via AnimationManager . . . 362
Animation manager to the rescue . . . 363
31.3. Low-level animations . . . 363
31.3.1. Why not just write code in paint . . . 364
31.4. Transitions . . . 364
31.4.1. Replace . . . 365
31.4.2. Slide transitions . . . 366
31.4.3. Fade and flip transitions . . . 368
31.4.4. Bubble transition . . . 369
31.4.5. Morph transitions . . . 370
Snapshot mode . . . 371
31.4.6. SwipeBackSupport . . . 371
32. The EDT - Event Dispatch Thread . . . 373
32.1. What’s The EDT . . . 373
32.2. Call Serially (And Wait) . . . 373
32.2.1. callSerially On The EDT . . . 374

32.3. Debugging EDT Violations . . . 375
32.4. Invoke And Block . . . 375
33. Graphics, drawing, images, and fonts . . . 379
33.1. Basics - where and how to draw manually . . . 379
33.1.1. Painting with gradients . . . 381
33.1.2. Image blur . . . 381
33.2. Glass pane . . . 381
33.3. Shapes & transforms . . . 383
33.4. Device support . . . 383
33.5. A 2D drawing app . . . 383
33.5.1. Implementing addpoint() . . . 384
33.5.2. Using Bézier curves . . . 385
33.5.3. Detecting platform support . . . 386
33.6. Transforms . . . 386
33.6.1. Device support . . . 386
33.7. Example: Drawing an analog clock . . . 387
33.7.1. The AnalogClock Component . . . 387
33.7.2. Setting up the parameters . . . 388
33.7.3. Drawing the tick marks . . . 388
33.7.4. Drawing the numbers . . . 389
33.7.5. Drawing the hands. . . . 389
33.7.6. The final result . . . 390
33.7.7. Animating the clock. . . . 391
33.8. Starting and stopping the animation . . . 391
33.9. Shape clipping . . . 391
33.10. The coordinate system . . . 392
33.10.1. Relative coordinates . . . 393
33.10.2. Transforms and rotations . . . 394
33.10.3. Global alpha & Anti-Aliasing . . . 396
33.10.4. Event coordinates . . . 397
33.11. Images . . . 397
33.11.1. Loaded Image . . . 397
33.11.2. The RGB image’s . . . 398
Internal . . . 398
RGBImage class . . . 398
33.11.3. EncodedImage . . . 399
33.11.4. MultiImage . . . 400
33.11.5. FontImage & Material design icons . . . 400
Material design icons . . . 401
33.11.6. Timeline . . . 402
33.11.7. Image masking . . . 402

33.11.8. URLImage . . . 403
Authenticated image URLs — `RequestDecorator` . . . 404
Mask adapter . . . 405
URLImage in lists . . . 405
33.12. Charts . . . 406
34. 3D Graphics and Shaders . . . 409
34.1. Concepts . . . 409
34.2. A first scene: A spinning cube . . . 410
34.3. Materials . . . 410
34.4. Meshes and buffers . . . 411
34.5. Loading models . . . 411
34.6. Animation . . . 412
34.7. Camera and math . . . 412
34.8. Platform notes . . . 413
34.9. Threading . . . 413
35. Augmented Reality . . . 415
35.1. Concepts . . . 415
35.2. Getting started: Place a model on the floor . . . 416
35.3. Plane detection . . . 416
35.4. Hit testing and anchors . . . 417
35.5. Light estimation . . . 417
35.6. Image tracking . . . 417
35.7. Face tracking . . . 418
35.8. Events and threading . . . 418
35.9. The simulator and debugging . . . 418
35.10. Platform notes . . . 419
36. Virtual Reality and 360 Media . . . 421
36.1. Concepts . . . 421
36.2. Getting started: A stereo scene . . . 421
36.3. Head tracking . . . 423
36.4. The 360 photo viewer . . . 423
36.5. 360 video and dynamic content . . . 423
36.6. The simulator . . . 424
36.7. Limitations and notes . . . 424
37. Game Development . . . 425
37.1. Sample games . . . 425
37.2. The game loop: `GameView` . . . 426
37.2.1. Lifecycle . . . 426
37.2.2. Fixed timestep and interpolation . . . 427
37.2.3. Threading . . . 427
37.3. Input: `GameInput` . . . 427

37.4. On-screen touch controls . . . 427
37.5. Sprites . . . 428
37.5.1. Scenes . . . 428
37.5.2. Sprite sheets and animation . . . 428
37.6. Case study: The card game (flat 2D) . . . 429
37.7. Case study: Faux-3D checkers (isometric projection) . . . 430
37.8. 3D and perspective . . . 431
37.8.1. Perspective camera and billboard sprites . . . 433
37.8.2. 3D models . . . 433
37.8.3. Meshes and textures . . . 433
37.9. Low latency audio: `SoundPool` . . . 433
37.10. Physics . . . 434
37.10.1. Units: Pixels, meters and the y-axis . . . 434
37.10.2. Worlds and bodies . . . 434
37.10.3. Driving sprites from physics . . . 435
37.10.4. Collisions . . . 435
37.10.5. Joints . . . 435
37.10.6. Debugging physics . . . 435
37.11. Putting it together: Anatomy of a game . . . 435
37.12. Performance notes . . . 436
38. The Game Builder . . . 437
38.1. How it fits together . . . 437
38.1.1. The level model — `com.codename1.gaming.level` . . . 437
38.2. Getting started . . . 438
38.3. The editor at a glance . . . 438
38.4. Building a level . . . 440
38.5. Defining game rules . . . 440
38.5.1. Data — set in the Inspector . . . 440
38.5.2. Logic — written in the companion . . . 441
38.6. Game types . . . 441
38.7. Live preview . . . 442
38.8. Build, save and run . . . 443
38.9. Maven goals . . . 443
38.10. Extending . . . 444
38.10.1. Importing your own art . . . 444
39. Game Assets . . . 445
39.1. What an asset is . . . 445
39.1.1. Asset (template) versus placed element (instance) . . . 445
39.2. How assets are structured . . . 446
39.2.1. The built-in packs . . . 446
39.3. Where the art comes from . . . 447

39.3.1. Static images (TYPE_IMAGE) . . . 447
39.3.2. Sprite sheets (TYPE_SHEET) . . . 447
39.3.3. Meshes (TYPE_MESH) . . . 447
39.3.4. Imported art (the editor's Import button) . . . 447
39.3.5. Generating the starter images . . . 447
39.4. Creating your own asset . . . 448
39.4.1. Path 1: Author a pack JSON entry . . . 448
39.4.2. Path 2: Import an image in the editor . . . 449
39.5. Terrain materials . . . 449
39.6. Where to get more . . . 449
40. Events . . . 451
40.1. High-level events . . . 451
40.1.1. Chain of events . . . 451
40.1.2. Action events . . . 451
Types of action events . . . 452
Source of event . . . 452
Event consumption . . . 453
NetworkEvent . . . 453
MessageEvent and the Cross-Platform message bus . . . 454
40.1.3. DataChangeListener . . . 454
40.1.4. FocusListener . . . 455
40.1.5. ScrollListener . . . 455
40.1.6. SelectionListener . . . 456
40.1.7. StyleListener . . . 456
40.1.8. Component state change events . . . 456
40.1.9. Event dispatcher . . . 456
40.2. Low-level events . . . 457
40.2.1. Low-level event types . . . 457
40.2.2. Drag event sanitation . . . 458
40.3. BrowserNavigationCallback . . . 459
41. Device Input and Form Factors . . . 461
41.1. Rich pointer events . . . 461
41.1.1. From an ActionEvent . . . 461
41.1.2. By polling . . . 461
41.2. Two and three button mice . . . 461
41.3. Mouse wheel and trackpad scrolling . . . 462
41.4. Trackpad gestures (magnify and rotate) . . . 462
41.5. Stylus and pen support . . . 462
41.6. Foldable devices and posture . . . 463
41.7. Desktop windowing and external displays . . . 463
41.8. Apple TV remote . . . 464

42. File system, storage, network, and parsing . . . 465
42.1. JAR resources . . . 465
42.2. Storage . . . 465
42.2.1. The preferences API . . . 466
42.3. File system . . . 467
42.3.1. File paths & app home . . . 467
42.3.2. Native file chooser . . . 468
42.3.3. Storage vs. file system . . . 469
42.4. SQL . . . 470
42.5. Network manager & connection request . . . 471
42.5.1. Timeouts & retries . . . 472
42.5.2. Threading . . . 472
42.5.3. VPN detection (best effort) . . . 472
42.5.4. Arguments, headers & methods . . . 473
Arguments . . . 473
Methods . . . 473
Headers . . . 474
Server headers . . . 474
Error handling . . . 474
Error stream . . . 475
42.5.5. GZIP . . . 475
42.5.6. File upload . . . 475
42.5.7. Parsing . . . 476
Parsing CSV . . . 476
JSON . . . 476
Writing JSON — `JSONWriter` . . . 479
XML parsing . . . 480
XPath processing . . . 480
Properties files . . . 486
42.6. Debugging network connections . . . 486
42.6.1. Simpler downloads . . . 486
Downloading images . . . 487
42.7. Rest API . . . 488
42.7.1. Top-level JSON arrays — `fetchAsJsonList` . . . 488
42.7.2. Typed responses — `fetchAsMapped` and `fetchAsMappedList` . . . 488
42.7.3. Rest in practice - twilio . . . 489
42.8. Webservice wizard . . . 490
42.9. Connection request caching . . . 494
42.9.1. getCachedData() . . . 495
42.9.2. Cacheunmodified() . . . 495
42.9.3. Purgecache & purgeCacheDirectory . . . 495

42.10. Cached data service . . . 495
42.11. Externalizable objects . . . 496
42.12. UI bindings & utilities . . . 497
42.13. Logging & crash protection . . . 498
42.14. Sockets . . . 498
42.15. Properties . . . 499
42.15.1. Properties in Java . . . 500
Encapsulation . . . 500
Introspection & observability . . . 501
42.15.2. The cool stuff . . . 501
Constructors . . . 501
Seamless serialization . . . 501
Seamless SQL storage . . . 502
Preferences binding . . . 504
UI binding . . . 504
UI generation . . . 505
43. Push notifications . . . 507
43.1. Create and configure an application . . . 507
43.2. Bind the client . . . 509
43.2.1. Listener lifecycle and early messages . . . 510
43.3. Build a message . . . 511
43.4. Send from a server . . . 512
43.5. Build audiences in the Console . . . 512
43.6. Send a message from the Console . . . 513
43.7. Event automations . . . 514
43.8. Widgets and live surfaces . . . 515
43.9. Use an application-owned push server . . . 515
43.10. Testing delivery behavior . . . 517
44. Notifications and Background Execution . . . 519
44.1. Notification permission . . . 519
44.2. Local notifications . . . 520
44.2.1. Resource names for sound, image and action icons . . . 521
44.3. Notification channels (Android O+) . . . 521
44.4. Constraint-aware background work . . . 521
44.5. Foreground service (Android) . . . 522
44.6. Receiving shared content . . . 522
44.7. Push topics . . . 522
44.8. Simulator . . . 523
45. Calendar integration . . . 525
45.1. Creating an event . . . 525
45.2. Online sources . . . 526

45.3. Import and export . . . 527
45.4. Change notifications and offline synchronization . . . 527
45.5. Permissions and build hints . . . 528
46. External Surfaces: Widgets and Live Activities . . . 529
46.1. The dead-process rule . . . 530
46.2. Getting started . . . 531
46.2.1. Previewing in the simulator . . . 533
46.3. The node catalog . . . 533
46.4. Vector widgets . . . 534
46.5. Live activities . . . 536
46.6. Actions and cold start . . . 537
46.7. Updating from the background . . . 537
46.8. Per-platform notes . . . 538
46.8.1. iOS . . . 538
46.8.2. Android . . . 539
46.8.3. Desktop, Windows, and Linux . . . 539
46.9. Build hints . . . 540
46.10. Current limitations . . . 541
47. Maps . . . 543
47.1. A first map . . . 543
47.2. The MapSurface API . . . 544
47.3. Routes that follow the road . . . 545
47.4. Tile sources and styles (MapView) . . . 546
47.4.1. Connecting to real map data . . . 547
47.5. Native maps and providers . . . 548
47.5.1. Provider API keys . . . 549
47.5.2. Cross-platform web maps . . . 549
47.6. Choosing between MapView and NativeMap . . . 550
48. In-Car Experiences (CarPlay & Android Auto) . . . 551
48.1. Quick start . . . 551
48.2. Templates . . . 552
48.3. Screen stack & lifecycle . . . 553
48.4. Now-playing (audio apps) . . . 553
48.5. Build configuration . . . 554
48.6. Debugging in the simulator . . . 555
48.7. Testing on the real platforms . . . 556
48.8. Limitations . . . 556
49. Motion Sensors . . . 557
49.1. Reading a sensor . . . 557
49.2. Sensor types . . . 557
49.3. Detecting gestures . . . 558

49.4. Device orientation . . . 558
49.5. Platform notes . . . 558
49.6. Testing in the simulator . . . 558
50. Miscellaneous features . . . 559
50.1. Phone functions . . . 559
50.1.1. SMS . . . 559
50.1.2. Dialing . . . 560
50.1.3. Call detection . . . 560
50.1.4. E-Mail . . . 560
50.2. Larger text accessibility . . . 561
50.2.1. How the scale is computed per platform . . . 561
50.2.2. Previewing in the simulator . . . 562
50.3. Contacts API . . . 562
50.4. Localization & internationalization (L10N & I18N) . . . 563
50.4.1. Localization manager . . . 565
50.4.2. RTL/Bidi . . . 565
50.4.3. Localizing native iOS strings . . . 566
Example: Localizing the app name . . . 566
Example: Localization app usage description strings . . . 568
50.4.4. Localizing the app icon . . . 568
Android behavior . . . 569
iOS behavior . . . 569
Tips and troubleshooting . . . 570
50.5. Location - GPS . . . 570
50.5.1. Location in the background - geofencing . . . 571
Android background location permissions (API 30+) . . . 572
50.6. Background music playback . . . 572
50.7. Capture - photos, video, audio . . . 572
50.7.1. Capture asynchronous API . . . 576
50.8. Gallery . . . 576
50.9. Analytics integration . . . 577
50.10. Social sign-in (Facebook, Google, ...) . . . 577
50.11. Lead Component . . . 577
50.11.1. Blocking lead behavior . . . 579
50.12. Pull to refresh . . . 580
50.12.1. Modern arc-spinner pull-to-refresh . . . 580
50.13. Running 3rd party apps using display's execute . . . 581
50.14. Automatic build hint configuration . . . 582
50.15. Easy thread . . . 582
50.16. Mouse cursor . . . 583
50.17. Working with GIT . . . 583

50.17.1. cn1lib's . . . 584
50.17.2. Resource files . . . 584
50.17.3. Eclipse version . . . 585
50.17.4. IntelliJ/IDEA . . . 585
51. Video encoding & decoding (VideoIO) . . . 587
51.1. Checking for support . . . 587
51.2. Encoding a video . . . 587
51.3. Decoding a video frame by frame . . . 588
51.4. Discovering available codecs . . . 589
51.5. Platform support . . . 589
52. Performance, size & debugging . . . 591
52.1. Reducing resource file size . . . 591
52.2. Improving performance . . . 592
52.2.1. ParparVM native translation performance hints . . . 593
Method-level codegen hints . . . 593
Class-level concrete implementation hints . . . 593
Fused object layout: `@Fused` . . . 593
Optimizations you get without annotations . . . 594
Fast method-stack path . . . 595
Base64-style hot-loop guidelines . . . 595
Build configuration matters . . . 595
52.3. SIMD: Data-Parallel primitives . . . 596
52.3.1. When to use SIMD . . . 596
52.3.2. Getting an instance . . . 597
52.3.3. Allocation: Alignment and provenance . . . 597
52.3.4. A minimal End-to-End example . . . 597
52.3.5. Scratch allocations: `alloca*` . . . 597
52.3.6. Rules for `alloca*` scratch arrays . . . 598
52.3.7. Build-Time verification . . . 599
52.3.8. Simulator tracking of aligned arrays . . . 599
52.3.9. Built-In SIMD paths . . . 600
52.4. Performance monitor . . . 600
52.5. Network speed . . . 601
52.6. Debugging Codename One sources . . . 602
52.7. Device testing Framework/Unit testing . . . 602
52.8. EDT error handler and sendlog . . . 603
52.9. Kitchen sink case study . . . 604
52.9.1. Scroll performance - threads aren't magic . . . 605
53. Testing with JUnit 5 . . . 607
53.1. When to use JUnit versus AbstractTest . . . 607
53.2. Dependencies . . . 607

53.3. A minimal JUnit test . . . 608
53.4. `@CodenameOneTest` . . . 608
53.5. `@RunOnEdt` . . . 609
53.6. `@SimulatorProperty` / `@SimulatorProperties` . . . 609
53.7. `@Theme` . . . 609
53.8. `@DarkMode` . . . 610
53.9. `@LargerText` . . . 610
53.10. `@Orientation` . . . 610
53.11. `@RTL` . . . 610
53.12. Annotation resolution . . . 610
53.13. EDT semantics . . . 610
53.14. Headless behavior . . . 611
53.15. Coexistence with `cn1:test` . . . 611
53.16. Side-by-side example . . . 611
53.17. Cross-reference . . . 612
54. Advertising . . . 613
54.1. Ad provider cn1libs . . . 614
54.2. Enabling a provider . . . 615
54.3. Initialization and consent . . . 616
54.4. Banner ads . . . 616
54.5. Interstitial ads . . . 616
54.6. Rewarded and rewarded interstitial ads . . . 616
54.7. App open ads . . . 616
54.8. Native ads . . . 616
54.9. Configuring the AdMob application id . . . 617
54.10. Implementing a provider . . . 617
55. Crash Protection . . . 619
55.1. What gets stored where . . . 619
55.2. Opting in . . . 619
55.2.1. Runtime hook . . . 619
55.3. Per-platform opt-out . . . 620
55.4. Tier limits . . . 620
55.5. Privacy . . . 620
55.5.1. What's PII . . . 620
55.5.2. What gets sent . . . 621
55.5.3. Default scrubber rules . . . 621
55.6. Mapping packages to repositories . . . 621
55.7. Setup checklist . . . 622
56. Analytics . . . 623
56.1. Registering providers . . . 623
56.2. Consent and privacy . . . 623

56.2.1. Consent you already hold . . . 624
56.2.2. Client identity and erasure . . . 624
56.3. Automatic events . . . 624
56.4. Built-in providers . . . 625
56.4.1. Codename One first-party service . . . 625
56.4.2. Google Analytics 4 . . . 625
56.4.3. Matomo . . . 625
56.4.4. Firebase . . . 625
56.4.5. Logging provider . . . 626
56.5. Writing your own provider . . . 626
56.6. Migrating from AnalyticsService . . . 626
57. Commerce . . . 627
57.1. Entitlements, not SKUs . . . 627
57.2. Client usage . . . 627
57.3. How the app finds the service . . . 627
57.4. Server notifications . . . 628
57.5. Server-side validation (connect your store credentials) . . . 628
57.6. Lifecycle webhooks to your backend . . . 628
57.7. The console . . . 629
57.8. Analytics . . . 629
57.9. Privacy . . . 630
57.10. Setup checklist . . . 630
58. Monetization . . . 631
58.1. In-app purchase . . . 631
58.1.1. The SKU . . . 631
58.1.2. Types of products . . . 631
58.1.3. The "hello world" of in-app purchase . . . 631
58.1.4. Making it consumable . . . 633
58.1.5. Non-Renewable subscriptions . . . 634
58.1.6. Promotional offers (iOS) . . . 635
58.1.7. Restoring purchases and managing subscriptions . . . 635
58.1.8. The Server-Side . . . 635
The receipts API . . . 636
58.1.9. The "hello world" of Non-Renewable subscriptions . . . 636
Implementing the receipt store . . . 637
Synchronizing receipts . . . 638
Expiry dates and subscription status . . . 639
Allowing the user to purchase the subscription . . . 639
58.1.10. Subscribe() vs purchase() . . . 640
Handling purchase callbacks . . . 640
58.1.11. Screenshots . . . 640

58.1.12. Summary . . . 641
58.1.13. Auto-Renewable subscriptions . . . 642
58.1.14. Auto-Renewable vs Non-Renewable: Choosing between them . . . 642
58.1.15. Learning by example . . . 642
58.1.16. Building the IAP demo project . . . 643
Setting up the client project . . . 643
Setting up the server project . . . 643
Setting up the database . . . 644
Testing the project . . . 644
58.1.17. Looking at the source of the app . . . 647
Client side . . . 647
Server-Side . . . 648
58.1.18. The CN1-IAP-Validator library . . . 648
58.1.19. The `validateAndSaveReceipt()` method . . . 649
58.1.20. Google play setup . . . 649
Creating the app in Google play . . . 649
Testing the app . . . 652
Creating Google play receipt validation credentials . . . 652
58.1.21. iTunes connect setup . . . 655
Setting up In-App products . . . 655
Creating test accounts . . . 655
Setting up receipt verification . . . 656
59. App Review & Feedback . . . 657
59.1. Asking for a review . . . 657
59.2. The engagement scheduler . . . 657
59.3. Smart feedback split . . . 657
59.4. How the platform decides . . . 658
60. Apple Wallet Extension . . . 659
60.1. How the extension works . . . 659
60.2. Prerequisites from Apple . . . 659
60.3. Mode 1: Generate the extensions from build hints . . . 660
60.3.1. The issuer endpoint . . . 660
60.3.2. The login UI extension . . . 661
60.3.3. Customizing the generated code . . . 661
60.3.4. Provisioning profiles for cloud builds . . . 661
60.4. Mode 2: Bring your own extension . . . 661
60.5. Testing . . . 662
61. Advanced topics/under the hood . . . 663
61.1. Sending arguments to the build server . . . 663
61.2. Versioned builds . . . 693
61.2.1. Building against master . . . 693

61.3. Android permissions. . . . 694
61.3.1. Permissions under Marshmallow (Android 6+) . . . 695
Enabling permissions . . . 695
Permission prompts. . . . 695
Code changes . . . 698
Simulating prompts . . . 699
AndroidNativeUtil's checkForPermission . . . 699
61.4. On device debugging. . . . 700
61.5. Native interfaces . . . 700
61.5.1. Introduction . . . 700
Swift (iOS) and Kotlin (Android) options . . . 704
Use the Android main thread (native EDT) . . . 704
Gradle dependencies . . . 705
61.5.2. Objective-C and Swift (iOS) . . . 706
Using the iOS main thread (native EDT) . . . 707
Use Cocoapods for dependencies . . . 707
61.5.3. C (native Windows and native Linux) . . . 708
61.5.4. JavaScript . . . 709
JavaScript examples . . . 710
61.5.5. Native GUI components . . . 711
61.5.6. Type mapping & rules . . . 712
61.5.7. Android native permissions . . . 714
61.5.8. Native AndroidNativeUtil . . . 715
61.5.9. Broadcast receiver . . . 716
Listening & permissions . . . 718
61.5.10. Native code callbacks . . . 719
Accessing callbacks from Objective-C . . . 720
Accessing callbacks from JavaScript . . . 722
Callbacks of the SMS receiver . . . 722
Asynchronous callbacks & threading . . . 723
61.6. Libraries - cn1lib . . . 724
61.6.1. Why not use JAR . . . 724
61.6.2. How to use cn1libs . . . 725
61.6.3. Creating a simple cn1lib . . . 725
61.6.4. Build hints in cn1libs. . . . 726
61.7. Integrating Android 3rd party libraries & JNI . . . 730
61.8. Drag & drop . . . 730
61.9. Android Lollipop ActionBar customization. . . . 732
61.10. Intercepting URLs on iOS & Android . . . 732
61.10.1. Passing launch arguments to the app . . . 733
61.10.2. Android App-to-App contracts (Wallet/Issuer style flows). . . . 733

Why this isn't implicitly generated by AndroidGradleBuilder 734
Registering the bridge activity 734
Bridge activity responsibilities 734
Reading contract data in `start()` 736
Returning results to the wallet API 737
Foreground behavior on Android 14/15 737
61.11. Native peer components 738
61.11.1. Why does Codename One need native widgets at all 738
61.11.2. Problems with native widgets 739
61.11.3. Showing dialogs on top of peer components 739
61.11.4. Why can't you combine peer component scrolling and Codename One scrolling . 739
61.11.5. Native components in the first Form 739
61.12. Integrating 3rd party native SDKs 739
61.12.1. Step 1 : Review the FreshDesk SDKs 740
61.12.2. Step 2: Designing the Codename One public API 740
61.12.3. Step 3: The architecture and internal APIs 740
Things to notice 741
61.12.4. Step 4: Implement the public API and native interface 742
Adapting method signatures 742
Callbacks 743
Initialization 745
The resulting public API 748
The native interface 749
Connecting the public API to the native interface 749
Implementing the glue between public API and native interface 751
61.12.5. Step 5: Implementing the native interface in Android 752
61.12.6. Step 6: Bundling the native SDKs 753
The FreshDesk SDK 754
Dependencies 754
61.12.7. Step 7: Injecting Android manifest and proguard config 756
Proguard config 758
Troubleshooting Android stuff 758
61.13. Part 2: Implementing the iOS native code 758
61.13.1. Using the MobihelpNativeCallback 759
61.13.2. Bundling native iOS SDK 760
61.13.3. Troubleshooting iOS 760
61.13.4. Adding required core libraries and frameworks 761
61.14. Part 3 : Packaging as a cn1lib 761
61.15. Building your own Layout manager 761
61.15.1. Porting a Swing/AWT Layout manager 763
61.16. Port a language to Codename One 764

61.16.1. What's a JVM language . . . 764
61.16.2. How hard is it to port a JVM language to Codename One . . . 764
Step 1: Assess the language . . . 764
Step 2: Convert the runtime library into a CN1Lib . . . 769
Step 3: Hello world . . . 770
Step 4: A more complex hello world . . . 771
Step 5: Automation and integration . . . 771
61.17. Update framework . . . 772
61.17.1. How does it work . . . 772
61.17.2. What isn't covered . . . 773
62. Security . . . 775
62.1. Constant obfuscation . . . 775
62.2. Storage encryption . . . 777
62.3. Secure storage and secrets . . . 777
62.3.1. SecureStorage (the platform keychain) . . . 777
62.3.2. Secrets (fetched from the cloud, never shipped in the binary) . . . 778
62.4. Disabling screenshots . . . 778
62.5. Blocking copy & paste . . . 778
62.6. Blocking jailbreak . . . 779
62.7. Device integrity, RASP & attestation . . . 779
62.7.1. Detecting rooted / jailbroken / tampered devices (RASP) . . . 780
62.7.2. Platform attestation: Google Play Integrity & Apple App Attest . . . 780
62.7.3. Defending against accessibility-service abuse (Android) . . . 781
62.8. Strong Android certificates . . . 782
62.8.1. The bad news . . . 782
62.8.2. The good . . . 783
62.9. Certificate pinning . . . 783
62.9.1. Certificate pinning . . . 784
62.10. Cryptographic primitives . . . 784
62.10.1. Hashing . . . 785
62.10.2. Message authentication (HMAC) . . . 785
62.10.3. Secure random numbers . . . 785
62.10.4. Symmetric encryption (AES) . . . 785
62.10.5. Asymmetric encryption and digital signatures (RSA, ECDSA) . . . 785
62.10.6. JSON Web Tokens . . . 786
62.10.7. HOTP / TOTP one-time passwords . . . 786
62.10.8. Walkthrough: Adding two-factor authentication to a sign-in flow . . . 787
62.10.9. OTP input widget . . . 788
62.10.10. Algorithm identifier reference . . . 788
62.10.11. Key sizes and recommended defaults . . . 788
63. App Shield . . . 791

63.1. How it fits together . . . 791
63.2. Turning it on . . . 792
63.3. Protecting hosts . . . 792
63.3.1. Wildcards and hosts discovered at runtime . . . 793
63.4. What happens to a request . . . 794
63.4.1. The token header . . . 794
63.5. Status, and what to do about it . . . 795
63.6. Getting a token yourself . . . 796
63.7. Certificate pinning . . . 797
63.8. Runtime signals . . . 798
63.9. Transports the guard doesn’t cover . . . 799
63.10. Composing with your own network guard . . . 800
63.11. Verifying the token in your backend . . . 800
63.12. iOS App Attest specifics . . . 801
63.13. Testing it in the simulator . . . 802
63.14. When there is no engine . . . 802
64. Biometric Authentication . . . 803
64.1. Quick start: Authenticate . . . 803
64.2. Quick start: Secure storage . . . 803
64.3. Platform behaviour . . . 803
64.4. Build hints . . . 804
64.5. Configuring the prompt . . . 805
64.6. Typed error handling . . . 805
64.7. Cancelling an in-flight prompt . . . 806
64.8. Listing enrolled biometrics . . . 806
64.9. Simulator workflow . . . 806
64.10. Security notes . . . 806
65. Authentication and Identity . . . 809
65.1. Why the change . . . 809
65.2. Quick start — any OpenID Connect provider . . . 809
65.2.1. Picking a redirect URI . . . 810
iOS . . . 810
Android . . . 811
65.2.2. Saving and restoring tokens . . . 811
65.2.3. Encrypting tokens at rest . . . 811
65.3. Sign in with Apple . . . 811
65.3.1. iOS . . . 811
65.3.2. Android, JavaSE, Web . . . 812
65.3.3. One-time Apple setup . . . 812
65.4. Google Sign-In . . . 812
65.4.1. Migrating from the legacy Google integration . . . 812

65.5. Facebook Login . . . 813
65.6. Microsoft / Entra ID . . . 813
65.7. Auth0 . . . 813
65.8. Firebase Authentication . . . 813
65.9. Passkeys / WebAuthn . . . 813
65.9.1. When you actually need this class . . . 814
65.9.2. Quick start — own backend . . . 814
65.9.3. Platform requirements . . . 814
65.9.4. Build hints . . . 815
65.9.5. Auth0 passkeys . . . 815
65.9.6. Firebase passkeys . . . 815
65.9.7. Mapping WebAuthn errors . . . 815
65.10. Migrating from `Oauth2` . . . 816
65.11. A universal sign-in demo . . . 816
66. Deep-Link Routing . . . 819
66.1. Declare a route . . . 819
66.2. Path syntax . . . 819
66.3. Navigate from app code . . . 819
66.4. How it works . . . 820
66.5. iOS Universal Links . . . 820
66.6. Android App Links . . . 820
66.7. Testing . . . 820
66.7.1. iOS Simulator . . . 821
66.7.2. Android Emulator . . . 821
66.7.3. Desktop Simulator . . . 821
66.8. Back-press guard . . . 821
67. JSON / XML Object Mapping . . . 823
67.1. Annotate the model . . . 823
67.1.1. Field-level annotations . . . 823
67.1.2. Supported field types . . . 824
67.2. Round-trip . . . 824
67.3. How the plumbing works . . . 824
67.4. Custom mappers . . . 825
68. Component Binding . . . 827
68.1. Annotate the model . . . 827
68.1.1. Attribute kinds . . . 828
68.2. Bind at runtime . . . 828
68.3. Two-way bindings and the change-notification contract . . . 828
68.3.1. Setter instrumentation . . . 829
68.3.2. Limitation — cross-field setter chains . . . 829
68.4. POJOs versus Property objects . . . 829

68.5. Validation annotations . . . 829
68.6. Build-time validation . . . 830
68.7. How the plumbing works . . . 831
69. SQLite ORM . . . 833
69.1. Annotate the entity . . . 833
69.1.1. Field-level annotations . . . 833
69.2. Use the dao . . . 834
69.3. Relationships . . . 834
69.4. Validation . . . 834
69.5. How the plumbing works . . . 834
70. AI, Chat UI, and Speech . . . 837
70.1. The `com.codename1.ai` package . . . 837
70.1.1. A first chat . . . 838
70.1.2. Streaming tokens . . . 838
70.1.3. Tool calling . . . 838
70.1.4. Structured output . . . 838
70.1.5. Multi-modal messages . . . 839
70.1.6. Embeddings . . . 839
70.1.7. Image generation . . . 839
70.1.8. Conversation persistence . . . 839
70.1.9. Prompt templates . . . 839
70.1.10. Retry policy . . . 839
70.1.11. Simulator redirect for offline development . . . 839
70.1.12. Storing the API key . . . 840
70.2. The `ChatView` component . . . 840
70.2.1. Theming . . . 841
70.2.2. One-call binding to an LLM . . . 842
70.3. Speech recognition and TextToSpeech . . . 842
70.3.1. `SpeechRecognizer` . . . 842
70.3.2. `TextToSpeech` . . . 842
70.4. On-device vision . . . 843
70.4.1. Vision feature and dependency selection . . . 844
70.5. Portable LiteRT inference . . . 844
70.6. On-device language services . . . 845
70.7. Platform availability . . . 846
70.8. Migrating from the AI cn1libs . . . 846
70.8.1. Putting it all together . . . 847
71. Near-Field Communication (NFC) . . . 849
71.1. Quick start: Read an NDEF URI . . . 849
71.2. Quick start: Write an NDEF URI . . . 849
71.3. Tag-technology APIs . . . 850

71.4. Quick start: Host card emulation . . . 850
71.5. Permissions and build hints . . . 851
71.6. Simulator support . . . 851
71.7. Platform behaviour . . . 852
72. Bluetooth . . . 855
72.1. Quick start: Scan and connect . . . 855
72.2. Reading and writing characteristics . . . 857
72.3. Notifications . . . 857
72.4. Peripheral mode . . . 858
72.5. L2CAP channels . . . 859
72.6. Classic Bluetooth . . . 860
72.7. Permissions and build hints . . . 861
72.8. Simulator support . . . 862
72.8.1. Record and replay . . . 865
72.9. Testing . . . 865
72.10. Platform behaviour . . . 865
73. Health . . . 867
73.1. Quick start . . . 868
73.2. Permissions, and the one thing that will surprise you . . . 869
73.3. Reading samples . . . 870
73.4. Aggregates and time buckets . . . 871
73.5. Writing samples . . . 872
73.6. Background delivery . . . 873
73.7. Workouts . . . 874
73.8. Bluetooth health sensors . . . 875
73.9. Nutrition . . . 877
73.10. Build configuration . . . 877
73.11. Privacy and store policy . . . 878
73.12. Simulating health . . . 879
73.13. Troubleshooting . . . 879
74. Printing . . . 881
74.1. Quick start: Print a PDF . . . 881
74.2. Printing images . . . 881
74.3. Handling the result . . . 881
74.4. Permissions and the user gate . . . 882
74.5. Platform notes . . . 882
74.6. Trying it in the simulator . . . 882
75. Deeper Network Connectivity . . . 883
75.1. WiFi information . . . 883
75.1.1. Required permissions and entitlements . . . 883
75.2. WiFi scan . . . 884

75.3. WiFi connect . . . 884
75.4. Bonjour / mDNS discovery . . . 885
75.4.1. iOS requirements . . . 885
75.4.2. Android requirements . . . 885
75.4.3. Simulator behaviour . . . 886
75.5. Network type change events . . . 886
75.6. WiFi Direct (Wi-Fi P2P) . . . 887
75.7. USB host . . . 887
75.8. Releasing platform resources . . . 888
76. Signing, certificates & provisioning . . . 889
76.1. Common terms in signing & distribution . . . 889
76.1.1. What's a certificate . . . 889
76.1.2. What's provisioning . . . 889
76.1.3. What's a signing authority . . . 889
76.1.4. What's UDID . . . 889
76.1.5. Should you reuse the same certificate for all apps . . . 890
76.2. Certificate Wizard . . . 890
76.2.1. Creating the App Store Connect API key . . . 890
76.2.2. Recommended fast path: Auto Setup . . . 891
76.2.3. Mac signing and notarization . . . 893
76.2.4. Android signing . . . 893
76.2.5. Windows signing . . . 894
76.2.6. Clearing cached signing data . . . 894
76.2.7. Manual asset management . . . 894
76.2.8. Selecting devices . . . 894
76.2.9. Decisions & edge cases . . . 895
76.2.10. App IDs and provisioning profiles . . . 897
76.2.11. Installing files locally . . . 897
76.2.12. Building your app . . . 898
76.3. Advanced iOS signing . . . 898
76.4. Provisioning profile & certificates visual guide . . . 899
76.4.1. Configuring `codenameone_settings.properties` for manual signing . . . 902
76.4.2. iOS code signing failure checklist . . . 903
76.5. Android . . . 904
76.5.1. Generating an Android certificate manually . . . 905
77. Working with iOS . . . 907
77.1. Troubleshooting iOS debug build installs . . . 907
77.2. Launch screen storyboard best practices . . . 907
77.2.1. Key files . . . 907
77.2.2. Designing a flexible Layout . . . 908
77.2.3. Asset reference . . . 908

77.2.4. Testing changes . . . 909
77.3. Local notifications on iOS and Android . . . 909
77.3.1. Sending notifications. . . . 909
Example sending notification . . . 909
77.3.2. Receiving notifications . . . 910
Example receiving notification. . . . 910
77.3.3. Canceling notifications . . . 910
77.4. iOS beta testing (testflight) . . . 910
77.5. Metal renderer . . . 911
77.6. Choosing a color space for the Metal renderer . . . 911
77.7. Accessing insecure URLs . . . 912
77.8. Using Cocoapods . . . 912
77.8.1. Including multiple pods . . . 913
77.8.2. Other pod related build hints . . . 913
77.8.3. Converting PodFile to build hints . . . 913
77.9. Using Swift package manager. . . . 913
77.10. Including dynamic frameworks. . . . 914
78. On-Device Debugging (iOS) . . . 915
78.1. When to use it. . . . 915
78.2. Quick start (IntelliJ IDEA) . . . 915
78.2.1. 1. Enable the build hints. . . . 915
78.2.2. 2. Build the app as you normally do . . . 916
78.2.3. 3. Run the proxy from IntelliJ . . . 916
78.2.4. 4. Launch the app and wait for the device handshake . . . 916
78.2.5. 5. Attach the debugger from IntelliJ . . . 916
78.3. Attach from NetBeans. . . . 917
78.4. Quick start (Maven from the command line) . . . 917
78.4.1. Using jdb instead . . . 917
78.5. What works today . . . 917
78.6. Pointing the IDE at the Codename One sources . . . 918
78.7. Known limitations . . . 919
78.8. Performance . . . 919
78.9. Debugging the generated Xcode project natively . . . 919
78.10. Troubleshooting. . . . 920
78.10.1. The proxy reports "device disconnected" before the breakpoint fires. . . . 920
78.10.2. "Connection refused" when the device tries to reach the proxy . . . 920
78.10.3. jdb reports "Internal exception: Unexpected JDWP Error: 100". . . . 920
78.10.4. The app launches but the breakpoint never fires . . . 920
78.10.5. Make the app wait while the proxy and IDE attach . . . 921
79. On-Device Debugging (Android) . . . 923
79.1. When to use it. . . . 923

79.2. Quick start (IntelliJ IDEA) . . . 923
79.2.1. 1. Enable the build hint. . . . 923
79.2.2. 2. Build the APK . . . 923
79.2.3. 3. Connect the device. . . . 924
79.2.4. 4. Run the debug session from IntelliJ . . . 924
79.2.5. 5. Attach the debugger . . . 924
79.3. Wireless debugging . . . 924
79.3.1. Android 11 and newer (recommended) . . . 924
79.3.2. Android 10 and older . . . 925
79.4. Quick start (Maven from the command line) . . . 925
79.5. Useful command-line flags . . . 926
79.6. What you can step through . . . 926
79.7. Pointing the IDE at the Codename One sources . . . 927
79.8. Debugging the generated Gradle project from Android Studio . . . 927
79.9. Troubleshooting . . . 928
79.9.1. "No Android device is online" . . . 928
79.9.2. "Multiple devices online" . . . 928
79.9.3. App installs but won’t pause for the debugger . . . 928
79.9.4. Breakpoint never fires . . . 928
79.9.5. logcat shows the app exiting with `Debug.waitForDebugger`-like noise. . . . 928
79.9.6. Wireless connection drops mid-session . . . 928
80. Working with JavaScript . . . 931
80.1. Choosing the JavaScript compiler . . . 931
80.2. Troubleshooting build errors . . . 931
80.3. ZIP, WAR, or preview: What’s the difference . . . 931
80.4. Setting up a proxy for network requests . . . 934
80.4.1. Step 1: Setting up a proxy. . . . 935
80.4.2. Step 2: Configuring your application to use the proxy . . . 935
80.5. Using the CORS proxy for same origin requests. . . . 936
80.5.1. Using apache as a proxy . . . 936
80.6. Customizing the splash screen . . . 937
80.7. Debugging . . . 937
80.8. Including Third-Party JavaScript libraries . . . 938
80.8.1. Libraries vs resources. . . . 938
80.8.2. The JavaScript manifest file. . . . 938
How to NOT generate the `script` tag . . . 939
Library directories . . . 940
Including remote libraries . . . 941
Including CSS files . . . 941
Embedding variables in URLs . . . 942
80.9. Browser environment variables . . . 942

80.10. Changing the native theme . . . 944
80.11. Disabling the 'OnBeforeUnload' handler . . . 944
80.11.1. Example: Toggling the BeforeUnload prompt On/Off . . . 944
80.12. Deploying as a progressive web app . . . 944
80.12.1. Customizing the app manifest file . . . 945
80.12.2. Related applications . . . 946
80.12.3. Device/Browser support for PWAs . . . 946
80.13. Playing media and opening links . . . 947
80.13.1. Choosing where a link opens . . . 948
80.14. BrowserComponent lifecycle . . . 949
80.14.1. Dialogs and BrowserComponent . . . 949
80.14.2. Preserving BrowserComponent state . . . 950
80.14.3. Lifecycle example . . . 950
81. Working with macOS . . . 951
81.1. macOS Desktop Build Options . . . 951
81.1.1. Bundle Types . . . 951
81.1.2. Understanding Mac Certificates . . . 952
81.1.3. Obtaining Certificates . . . 952
81.1.4. Exporting Certificates as P12 . . . 954
81.1.5. Entitlements . . . 954
81.1.6. Custom Info.plist Entries . . . 955
81.2. Mac Native Build (AOT-compiled macOS App) . . . 955
81.2.1. Local Build . . . 955
81.2.2. Cloud Build . . . 955
81.2.3. IDE Shortcuts . . . 956
81.2.4. Distribution . . . 956
81.2.5. Entitlements . . . 957
81.2.6. Supported Distribution Channels . . . 957
82. App Store Submission and Deployment . . . 959
82.1. What it does, and what it doesn’t . . . 959
82.2. Prerequisites . . . 959
82.2.1. Store credentials . . . 960
82.2.2. Beta vs production . . . 960
82.3. Submitting a build from the console . . . 960
82.4. Metadata as code . . . 961
82.4.1. The metadata folder . . . 961
82.4.2. Pushing the metadata . . . 962
82.4.3. When and how it’s applied . . . 963
82.4.4. Screenshots aren’t stored long-term . . . 963
82.5. Reaching Android users beyond Google Play . . . 963
82.5.1. Distribution-channel packages . . . 963

82.6. Benefits over editing the stores by hand . . . 964
83. Desktop Integration . . . 965
83.1. Title-bar modes . . . 965
83.1.1. Enabling it on the Java SE desktop build . . . 966
83.1.2. Enabling it on the macOS (Mac Catalyst) build . . . 967
83.2. Native menu bar and command placement hints . . . 967
83.2.1. Keyboard accelerators . . . 968
83.3. Interactive scrollbars . . . 968
83.4. Theme UIIDs . . . 969
83.5. Desktop notifications . . . 969
84. Working with the native Windows port . . . 971
84.1. How it works (the technology stack) . . . 971
84.2. Building a native Windows app . . . 972
84.3. Cloud builds (the `win32` target) . . . 972
84.4. Build hints . . . 973
84.5. Optimized, with debug info in a separate file . . . 973
84.6. Code signing . . . 973
84.6.1. Signing with a PKCS#12 certificate . . . 974
84.6.2. Hardware-backed and cloud keys . . . 974
84.7. Native Windows vs. the desktop (JVM) app vs. the jar . . . 975
84.8. How it works (the technology stack) . . . 976
84.9. Runs on essentially any Linux desktop . . . 976
84.10. Building a native Linux app . . . 977
84.10.1. Build-host dependencies . . . 977
84.10.2. Architecture and cross-compiling . . . 977
84.11. Build hints . . . 978
84.12. Optimized and stripped by default . . . 978
84.13. Native Linux vs. the executable jar . . . 979
84.14. Blocking native calls and the concurrent GC . . . 979
85. Wearables (Apple Watch and Wear OS) . . . 981
85.1. Detecting the Watch Form Factor . . . 981
85.2. Designing for the Watch . . . 982
85.3. Apple Watch (watchOS) . . . 982
85.3.1. Enabling the watchOS Build . . . 982
85.3.2. watchOS Build Hints . . . 983
85.3.3. Supported and Unsupported APIs on watchOS . . . 983
85.3.4. Building and Debugging . . . 984
85.4. Android (Wear OS) . . . 984
85.4.1. Enabling the Wear OS Build . . . 984
85.4.2. Wear OS Build Hints . . . 984
85.5. Summary . . . 985

86. Television (Apple TV and Android TV) . . . 987
86.1. Detecting the TV Form Factor . . . 987
86.2. Adapting Styling with CSS Media Queries . . . 987
86.3. Building for Android TV / Google TV . . . 988
86.3.1. Android TV Build Hints . . . 988
86.4. Building for Apple TV (tvOS) . . . 988
86.4.1. tvOS Build Hints . . . 989
87. Working with Codename One sources . . . 991
87.1. Prerequisites . . . 991
87.2. Preparing the Maven workspace . . . 991
87.3. Installing the Maven archetypes . . . 991
87.4. Running tests selectively . . . 992
87.5. Using your local build in application projects . . . 992
87.6. Why build from source . . . 992
88. Skin Designer . . . 993
88.1. Stage 1 — pick a device . . . 993
88.2. Stage 2 — pick a starting source . . . 994
88.3. Stage 3 — the editor . . . 994
88.3.1. Shape tab . . . 995
88.3.2. Cutouts tab . . . 995
88.3.3. Info tab . . . 996
88.4. Stage 4 — finish and download . . . 997
88.5. File layout of a generated skin . . . 998
Appendix A: Project archetypes . . . 1001
A.1. Codename One application project archetype (cn1app-archetype) . . . 1001
A.2. Codename One library project archetype (cn1lib-archetype) . . . 1001
Appendix B: Maven goals . . . 1003
B.1. Codename One build (`build`) . . . 1003
B.1.1. Example . . . 1003
B.1.2. Properties . . . 1003
B.2. Generate legacy cn1lib (`cn1lib`) . . . 1004
B.2.1. Output location . . . 1004
B.3. Compliance check (`compliance-check`) . . . 1004
B.4. Compile CSS (`css`) . . . 1004
B.5. Generate app project (`generate-app-project`) . . . 1005
B.5.1. Usage example . . . 1005
B.5.2. Parameters . . . 1005
B.5.3. Migrating legacy Ant projects into Maven . . . 1006
B.5.4. Project templates . . . 1006
The `generate-app-project.rpf` file . . . 1006
Sample `generate-app-project.rpf` file . . . 1007

B.5.5. Sample: The bare-bones Kotlin app project 1007
B.6. Clone project (clone) 1008
B.6.1. Usage example 1008
B.6.2. Parameters 1008
B.7. Create GUI Form (create-gui-form) 1008
B.7.1. Parameters 1009
B.8. Generate cn1lib project (generate-cn1lib-project) 1009
B.8.1. Usage example 1009
B.8.2. Properties 1011
B.9. Generate desktop app wrapper (generate-desktop-app-wrapper) 1011
B.10. Generate GUI sources (generate-gui-sources) 1011
B.11. Generate native interfaces (generate-native-interfaces) 1011
B.11.1. Usage example 1011
B.12. Generate OpenAPI client (generate-openapi) 1012
B.12.1. Usage example 1013
B.12.2. Generated output 1013
B.12.3. Scope 1014
B.13. Generate gRPC client (generate-grpc) 1015
B.13.1. Usage example 1015
B.13.2. Generated output 1016
B.13.3. Wire protocol 1016
B.13.4. Scope 1017
B.14. Generate GraphQL client (generate-graphql) 1017
B.14.1. Usage example 1018
B.14.2. Generation modes 1018
B.14.3. Generated output 1019
B.14.4. Wire protocol 1019
B.14.5. Scope 1020
B.15. GUIBuilder goal (guibuilder) 1020
B.15.1. Usage 1020
B.15.2. Parameters 1020
B.16. Generate archetype (generate-archetype) 1020
B.16.1. Parameters 1021
B.16.2. Template syntax 1021
Template sections 1023
B.17. Install Codename One (install-codenameone) 1026
B.18. Install legacy Cn1lib (install-cn1lib) 1026
B.18.1. Usage example 1027
B.18.2. Removing the Cn1lib later 1027
B.18.3. Parameters 1027
B.19. Open Designer (designer) 1028

B.19.1. Usage example 1028
B.20. Prepare simulator classpath (`prepare-simulator-classpath`) 1028
B.21. Open settings (`settings`) 1028
B.22. Run tests (`test`) 1028
B.23. Update Codename One (`update`) 1029
B.23.1. Usage example 1029
Appendix C: API 1031
C.1. Limitations 1031
C.2. Kotlin API support 1031
C.3. Add-on libraries 1031
Appendix D: Codename One Settings 1033
D.1. Opening Codename One Settings 1033
D.2. Basic settings 1033
D.3. Build hints 1033
D.4. Extensions 1034
Appendix E: Rich properties file (rpf) format 1035
E.1. Example rich properties file 1035
Historical reference 1037
89. Working with UWP 1039

Preface

This developer guide comes directly from the `docs` directory of the Codename One Git repository [https://github.com/codenameone/CodenameOne/]. The documentation is written in AsciiDoc, reviewed through pull requests, and published from the main branch to several targets.

How to contribute updates

To contribute updates, clone the repository, create a feature branch, and edit the AsciiDoc files in `docs/developer-guide`. Follow the steps in the CONTRIBUTING guidelines [https://github.com/codenameone/CodenameOne/blob/master/CONTRIBUTING.md] to open a GitHub pull request (PR) instead of using the deprecated community wiki. Continuous integration lints and builds each PR automatically. A maintainer then reviews the PR before merging it. After the merge, the content appears in the web manual at https://www.codenameone.com/manual/ and in the downloadable PDF at https://www.codenameone.com/files/developer-guide.pdf [https://www.codenameone.com/files/developer-guide.pdf]. From time to time, major revisions are collected for print-on-demand distribution.

This guide focuses on tutorial and conceptual material. The complete API reference is available in the Codename One Javadoc [https://www.codenameone.com/javadoc/]. The framework source code and this manual live side by side in Git, so documentation and code improvements follow the same contribution workflow.

Authors

This document includes content from contributors and community wiki edits. If you edit pages in the guide, add your name here in alphabetical order by surname:

- Shai Almog [https://github.com/shai-almog]
- Ismael Baum [https://github.com/Isborg]
- Eric Coolman [https://twitter.com/ericcoolmandev]
- Chen Fishbein [http://github.com/chen-fishbein/]
- Steve Hannah [http://github.com/shannah/]
- Matt [https://github.com/kheops37]

Rights & Licensing

You may copy, redistribute, or print this document without prior permission from Codename One. You may not charge for the document itself, but charging for printing costs remains permitted.

If you print or reproduce sections and make changes, state that the result is a modified copy. Also link to the original source at https://www.codenameone.com/developer-guide/.

Conventions

This guide uses several notations for tips and further guidance.

When additional information breaks from the current tutorial flow, use a sidebar like this:

> **Sidebar**
>
> Dig deeper into details that don't quite fit the current flow. Sidebars cover important detours. You can skip them while reading and return to them later.

Here are common conventions for highlighting notes:



This is an important note



This is a helpful tip



This is a general informational note. It may be interesting, but it isn't crucial.



This is a warning. Pay attention to it.

`This` convention is used when referring to a button or widget to press in the UI. For example, press the button labeled `Press Here`.

Quotes appear as follows:

> This is a quote
>
> — By Author

Bold provides light emphasis for a specific word or two within a sentence.

Chapter 1. Maven Project Workflow

Codename One uses Maven as the standard way to create, run, and maintain applications. This chapter consolidates the workflow guidance that used to live in the standalone Maven manual so the developer guide provides a single source of truth for building apps, updating projects, and managing add-ons.

1.1. Introduction

Codename One uses Maven as its primary build tool. This guide aims to be the definitive source of information for this project structure.

1.1.1. Conventions

The instructions throughout this chapter provide parallel guidance for the supported development environments. Each subsection heading identifies the target tooling so you can follow the directions that match your workflow:

- **Command Line (CLI):** Focused on running Maven from a terminal. The commands use a Unix-style shell syntax; adapt the examples for Windows when necessary.
- **IntelliJ IDEA:** Step-by-step directions for working inside IntelliJ.
- **NetBeans:** Equivalent instructions tailored to NetBeans.
- **Eclipse:** Maven-based guidance for Eclipse users.

When you reach environment-specific sections later in the guide, follow the subsection corresponding to your preferred tools.

1.2. Getting started

1.2.1. Prerequisites

Codename One supports JDK 11 through 25 for running the simulator and the "Run as desktop app" target. Eclipse Temurin [https://adoptium.net] is the easiest source of a supported JDK on macOS, Windows, and Linux.

After installing the JDK, point `JAVA_HOME` at it and confirm with:

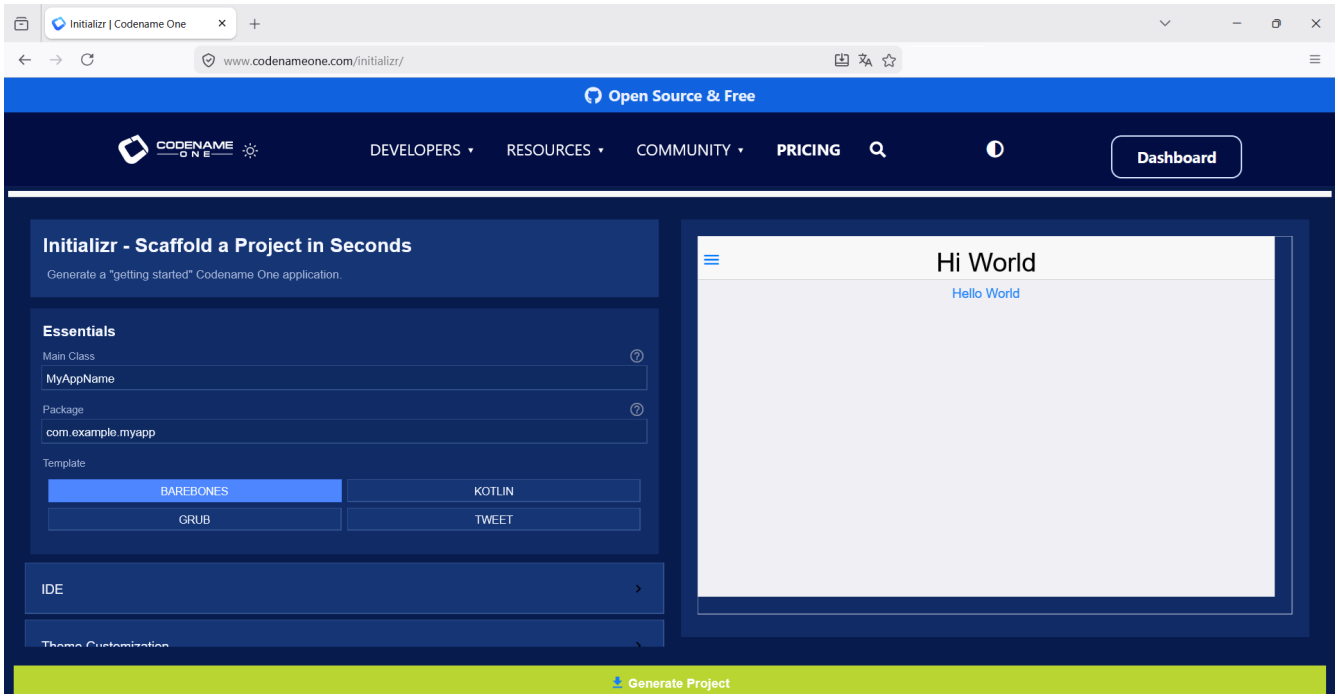
```
java -version
mvn -v
```

Both should report Java 11 or newer. The Codename One Maven plugin checks the runtime JDK version when you invoke `mvn cn1:run` or `mvn cn1:debug` and fails fast with a friendly message if it's older than 11. Build-only goals (such as `-Pexecutable-jar`) still work on older JDKs.

1.2.2. Creating a new project

Codename One initializr

The easiest way to create a new project is to use the Codename One initializr [https://www.codenameone.com/initializr].



This tool will allow you to choose from a growing selection of project templates, and download a starter project that you can open in your preferred IDE (IntelliJ IDEA, NetBeans, etc.), or build directly on the command-line using Maven.

The starter projects are based on the Section A.1, “Codename One application project archetype (cn1app-archetype)”.



The following tutorials provide step-by-step instructions for getting started with bare-bones app templates. Those tutorials are a better starting place for Codename One development than this manual, as they’re written in tutorial form.

Java Getting Started Tutorial:

See Getting Started with the Bare-bones Java App Template [https://shannah.github.io/cn1-maven-archetypes/cn1app-archetype-tutorial/getting-started.html].

Kotlin Getting Started Tutorial:

See Getting Started with the Bare-bones Kotlin App Template [https://shannah.github.io/cn1app-archetype-kotlin-template/getting-started.html].

Generating a new project from the command-line

Bare-bones Java project

If you prefer to generate your projects directly on the command-line, you can use the Section A.1, “Codename One application project archetype (cn1app-archetype)” to generate the project directly on the command-line:

```
mvn archetype:generate \
  -DarchetypeGroupId=com.codenameone \
  -DarchetypeArtifactId=cn1app-archetype \
  -DarchetypeVersion=LATEST \
  -DgroupId=YOUR_GROUP_ID \
  -DartifactId=YOUR_ARTIFACT_ID \
  -Dversion=1.0-SNAPSHOT \
  -DmainName=YOUR_MAIN_NAME \
  -DinteractiveMode=false
```

This will generate a project in the current directory. The project’s directory will have the same name as the artifact ID you specified here. For example, If your command had `-DartifactId=myapp`, then the project will be located in a newly created directory named "myapp."



If you haven’t used Maven archetypes before, this snippet may be confusing. See Introduction to Maven Archetypes [https://maven.apache.org/guides/introduction/introduction-to-archetypes.html] to get up to speed.

This command uses the Section A.1, “Codename One application project archetype (cn1app-archetype)” which has the following Maven coordinates:

```
<dependency>
  <groupId>com.codenameone</groupId>
  <artifactId>cn1app-archetype</artifactId>
  <version>LATEST</version>
  <type>maven-archetype</type>
</dependency>
```

This archetype generates a bare-bones Java project (the same one described in Getting Started with the Bare-bones Java App Template [https://shannah.github.io/cn1-maven-archetypes/cn1app-archetype-tutorial/getting-started.html]).



View the source of the cn1app-archetype project here [https://github.com/shannah/cn1-maven-archetypes/tree/master/cn1app-archetype].

You can learn more about using the archetype in the appendix.

Project templates

The bare-bones Kotlin App project template [https://shannah.github.io/cn1app-archetype-kotlin-template/getting-started.html] is an alternative starter project that uses Kotlin as the primary language instead of Java. It’s built on the cn1app-archetype at its core, but it includes some more configuration

settings and sources to change the template. You can use such templates as starter projects by using the `generate-app-project` goal of the Codename One Maven plugin.

Here is an example which generates a project based on the bare-bones Kotlin template:

```
mvn com.codenameone:codenameone-maven-plugin:7.0.210:generate-app-project \
  -DarchetypeGroupId=$archetypeGroupId \
  -DarchetypeArtifactId=$archetypeArtifactId \
  -DarchetypeVersion=$archetypeVersion \
  -DartifactId=$artifactId \
  -DgroupId=$groupId \
  -Dversion=$version \
  -DmainName=$mainName \
  -DinteractiveMode=false \
  -DsourceProject=/path/to/kotlin-example-app
```



This command is formatted for the bash prompt (for example, Linux or Mac). It will work on Windows also if you use bash. If you're on Windows and are using PowerShell or the regular command prompt, then you'll need to modify the command slightly. In particular, the entire command would need to be on a single line. (Remove the '\' at the end of each line, and merge lines together, with space between the command-line flags)

Like the `archetype:generate` goal, this will create the project in a directory named after your specified artifact ID. For example, If your command included `-DartifactId=myapp`, then the project would be in a newly created directory named "myapp."

Some notes here:

1. The `com.codenameone:codenameone-maven-plugin:7.0.210:generate-app-project` argument is the fully qualified goal name for the `generate-app-project`. This is necessary since you aren't running this goal in the context of any existing project. You should adjust the version number (`7.0.210`) to reflect the latest available Codename One version on Maven Central [https://search.maven.org/search?q=a:codenameone-maven-plugin].
2. The `archetypeGroupId`, `archetypeArtifactId`, and `archetypeVersion` parameters are the same as when using the `archetype:generate` goal, and they will (almost) always refer to the Section A.1, "Codename One application project archetype (cn1app-archetype)".
3. The `groupId`, `artifactId`, and `version` work the same as for the `archetype:generate` goal. That's, that they specify the coordinates for your newly created project.
4. The `mainName` specifies the Main class name for your app. This is the class name, and shouldn't include the full package. For example, `MyApp`, not "com.example.MyApp"
5. The `sourceProject` property is the path to the "template" project. In this case, you will assume that you've cloned the bare-bones Kotlin project template repository [https://github.com/shannah/cn1app-archetype-kotlin-template] at /path/to/kotlin-example-app.



A project template isn't much different than a regular project. The template can be

> either a legacy Ant project, or a new Maven project. In fact, this goal is the same one you would use to migrate a legacy Ant project to use the new Maven project structure.
>
> See Section 1.4, “Creating project templates” for instructions on building your own project templates.

1.2.3. Migrating an existing project

If you have an existing Codename One application project that uses the old Ant project structure, you can use the `generate-app-project` goal to migrate the project over to maven. This goal doesn't make any changes to the Ant project. It creates a new Maven project and copies over all the project sources and libraries, reorganized to fit the new project structure.

A minimal invocation of this goal would look like:

```
# Specify your the version of the codenameone-maven-plugin.
# Find the latest version at
# https://search.maven.org/search?q=a:codenameone-maven-plugin
CN1VERSION=7.0.210
mvn com.codenameone:codenameone-maven-plugin:$CN1VERSION:generate-app-project \
  -DgroupId=YOUR_GROUP_ID \
  -DartifactId=YOUR_ARTIFACT_ID \
  -DsourceProject=/path/to/your/project \
  -Dcn1Version=$CN1VERSION
```



> This command is formatted for the bash prompt (for example, Linux or Mac). It will work on Windows also if you use bash. If you're on Windows and are using PowerShell or the regular command prompt, then you'll need to modify the command slightly. In particular, the entire command would need to be on a single line. (Remove the '\' at the end of each line, and merge lines together, with space between the command-line flags)

This will generate the new project in the current directory inside a folder named after the `artifactId` parameter.

After building the project, try running it to make sure that the migration worked. For example, Assuming that your artifactId was myapp:

Command line

```
cd myapp
./run.sh
```



> On Windows it would be `run.bat` instead of `run.sh`.

If All goes well, your app should open in the Codename One simulator.

IntelliJ IDEA

Open the `myapp` folder in IntelliJ. Then press the "Run"  button in the upper right of the toolbar.

If All goes well, your app should open in the Codename One simulator.

NetBeans



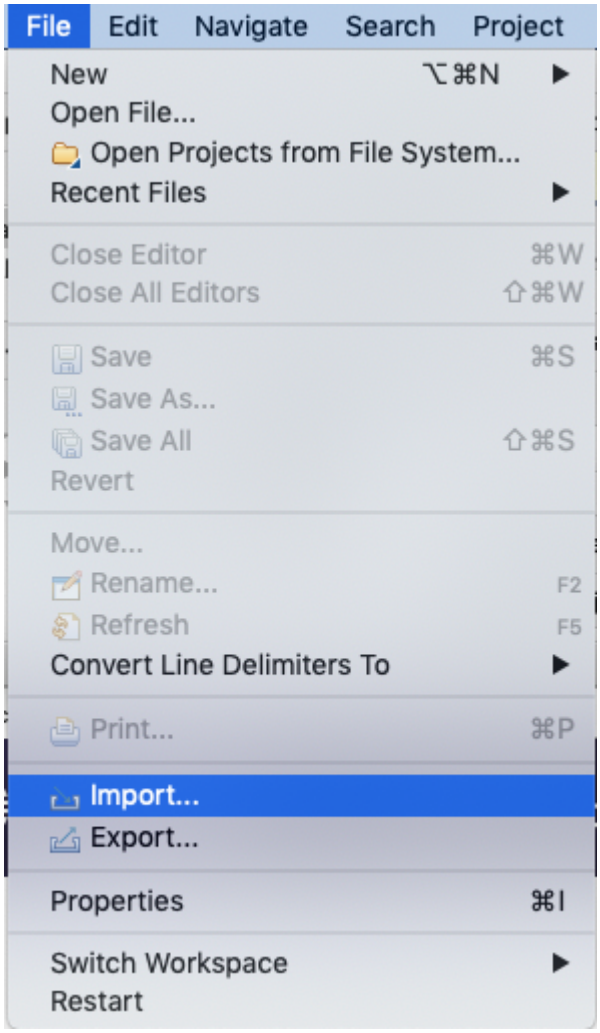
Before opening the project in NetBeans, be sure to copy the files in the *tools/netbeans* directory into the root directory. These are necessary for NetBeans to run, build, and debug the project.

Open the `myapp` folder as a project in NetBeans. Then press the "Run"  button on the toolbar.

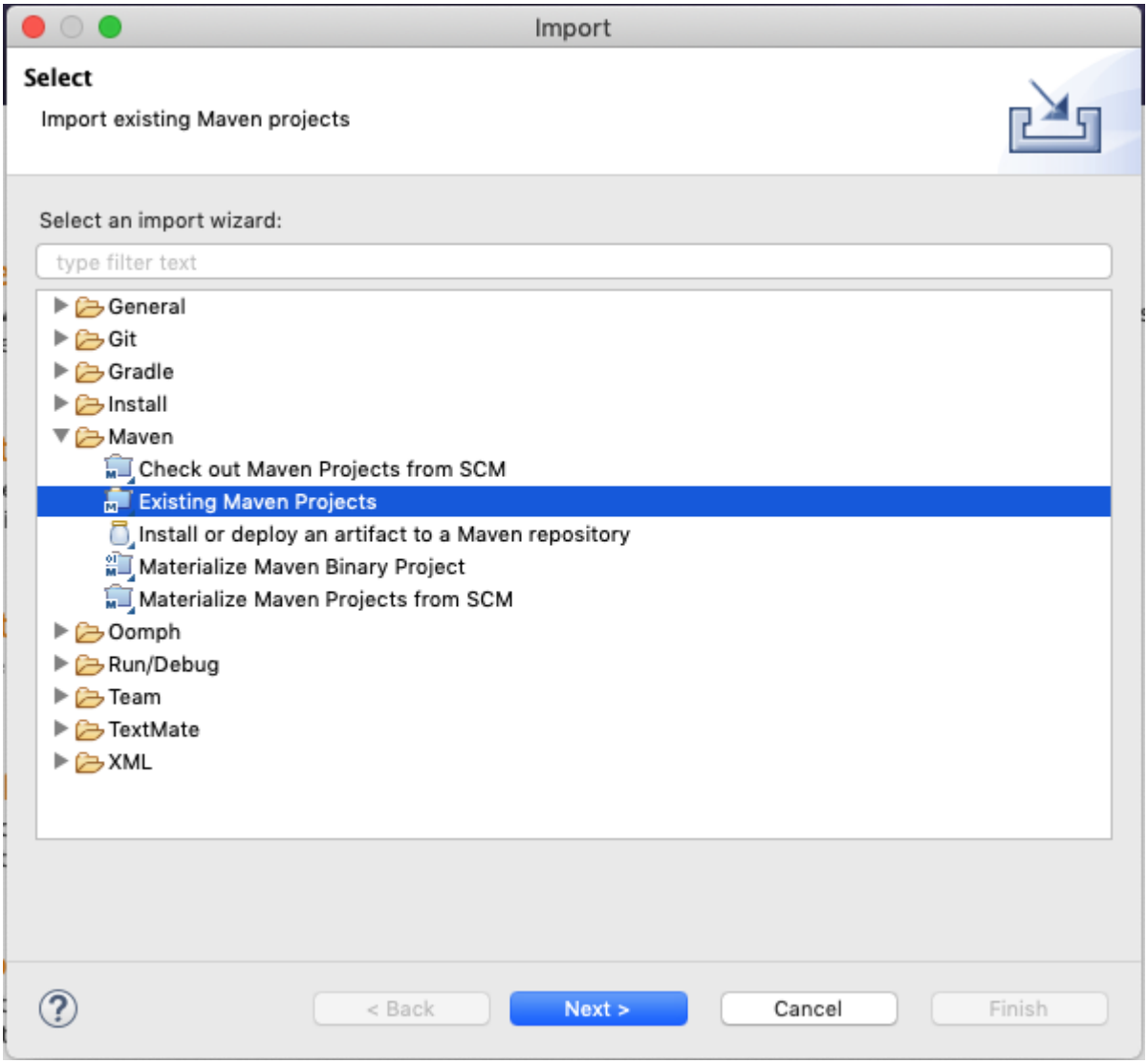
If all goes well it should open in the Codename One simulator.

Eclipse IDE

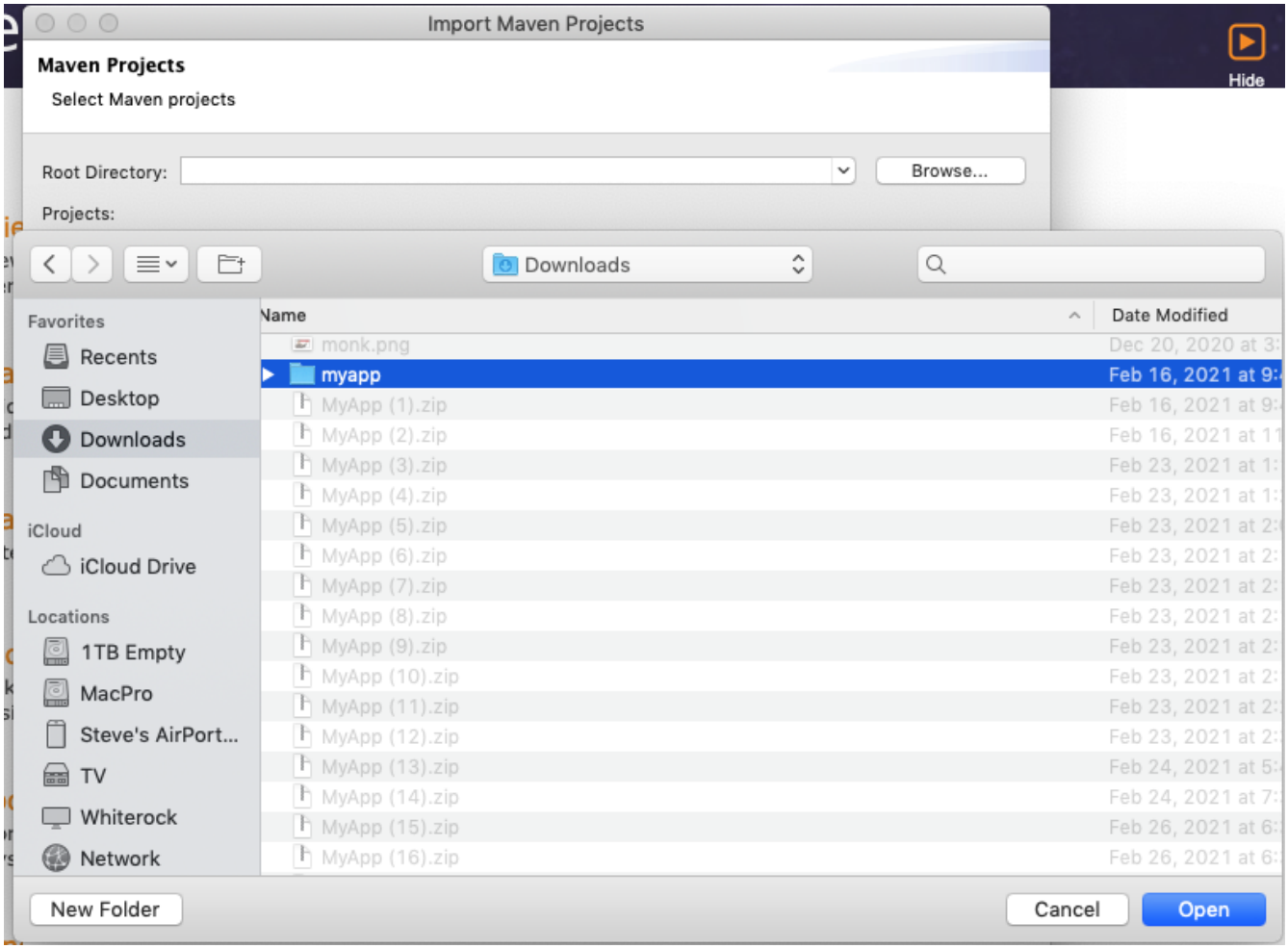
Open Eclipse, and select "File" > "Import..."



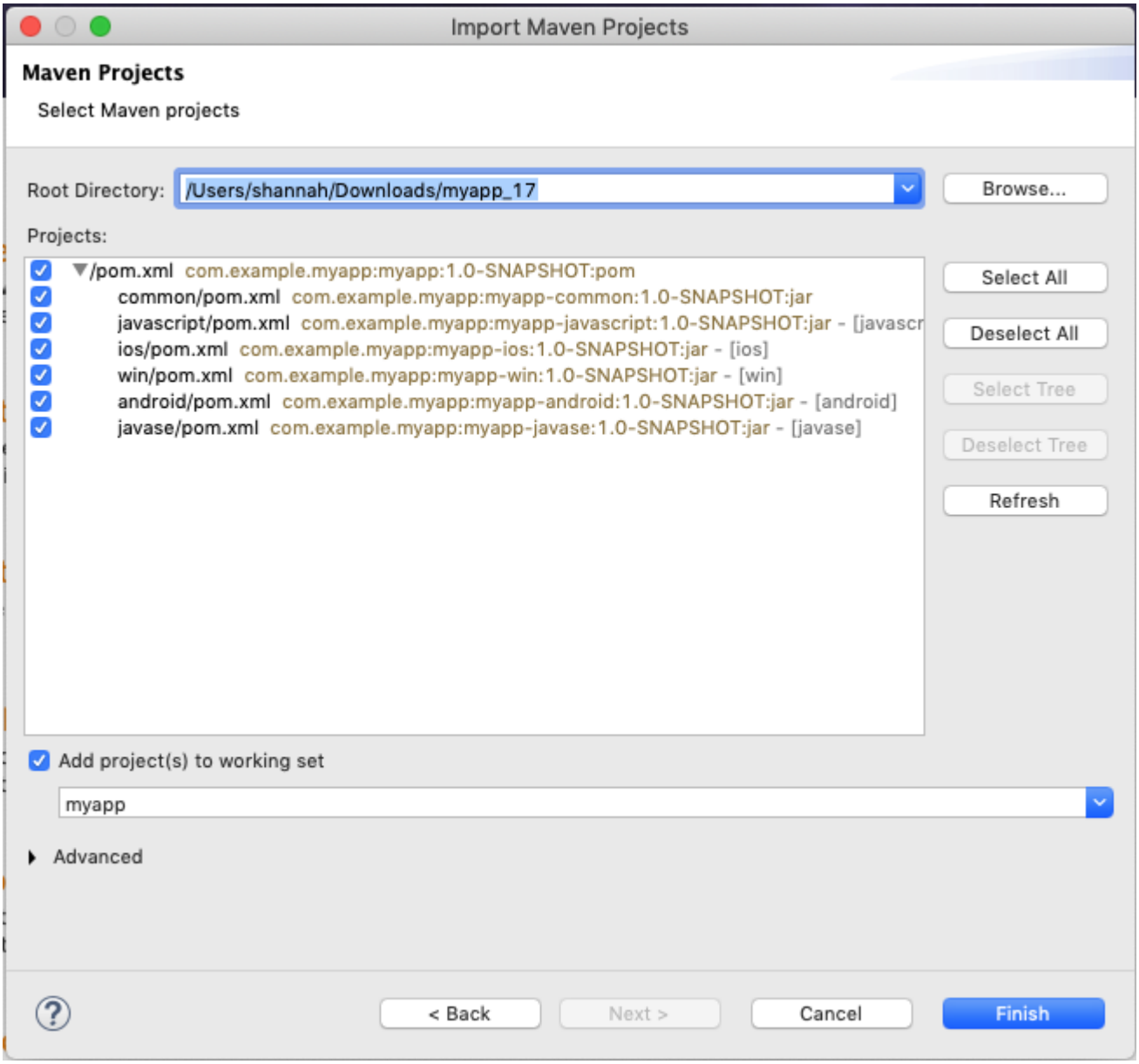
In the *Import* dialog, expand *Maven*, select *Existing Maven Projects*, and press *Next*.



In the next panel, press the *Browse* button, and, in the file dialog, select the "myapp" directory, and press *Next*.



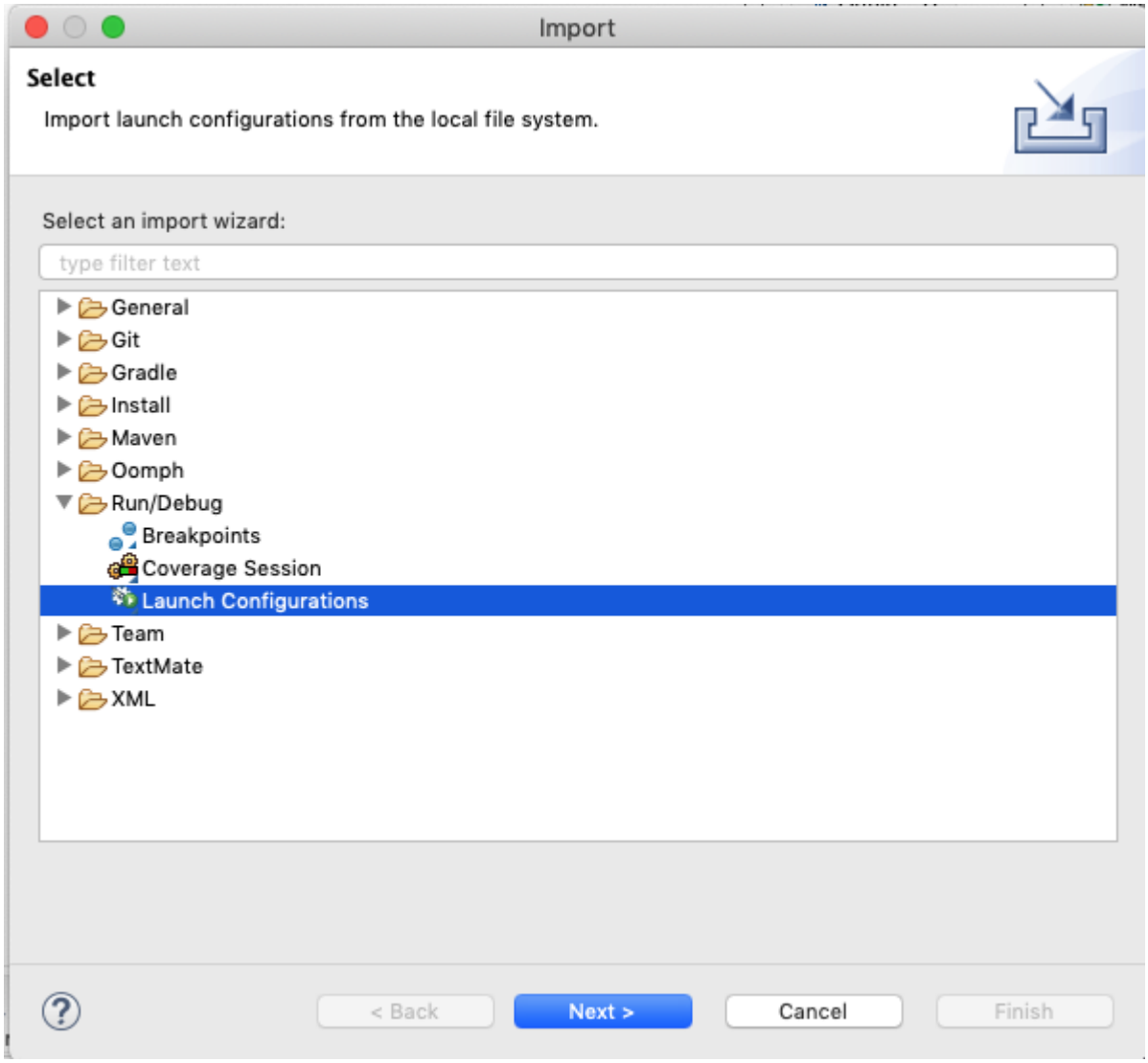
The next panel should look like the one below. Make sure all the projects are "checked," and press *Finish*.



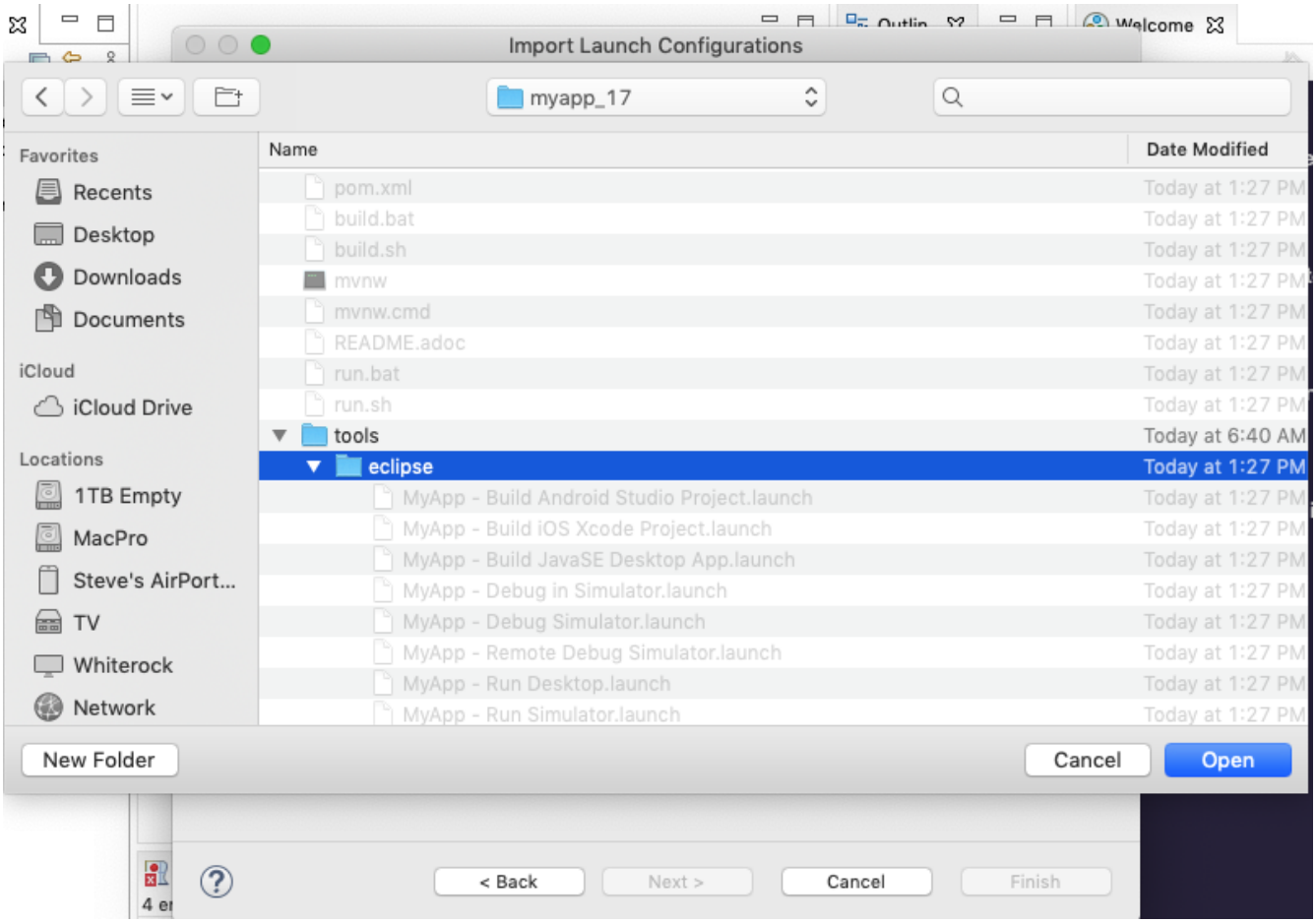
Almost there, but not quite…

Next you need to import the Eclipse launch configurations located inside the *tools/eclipse* directory.

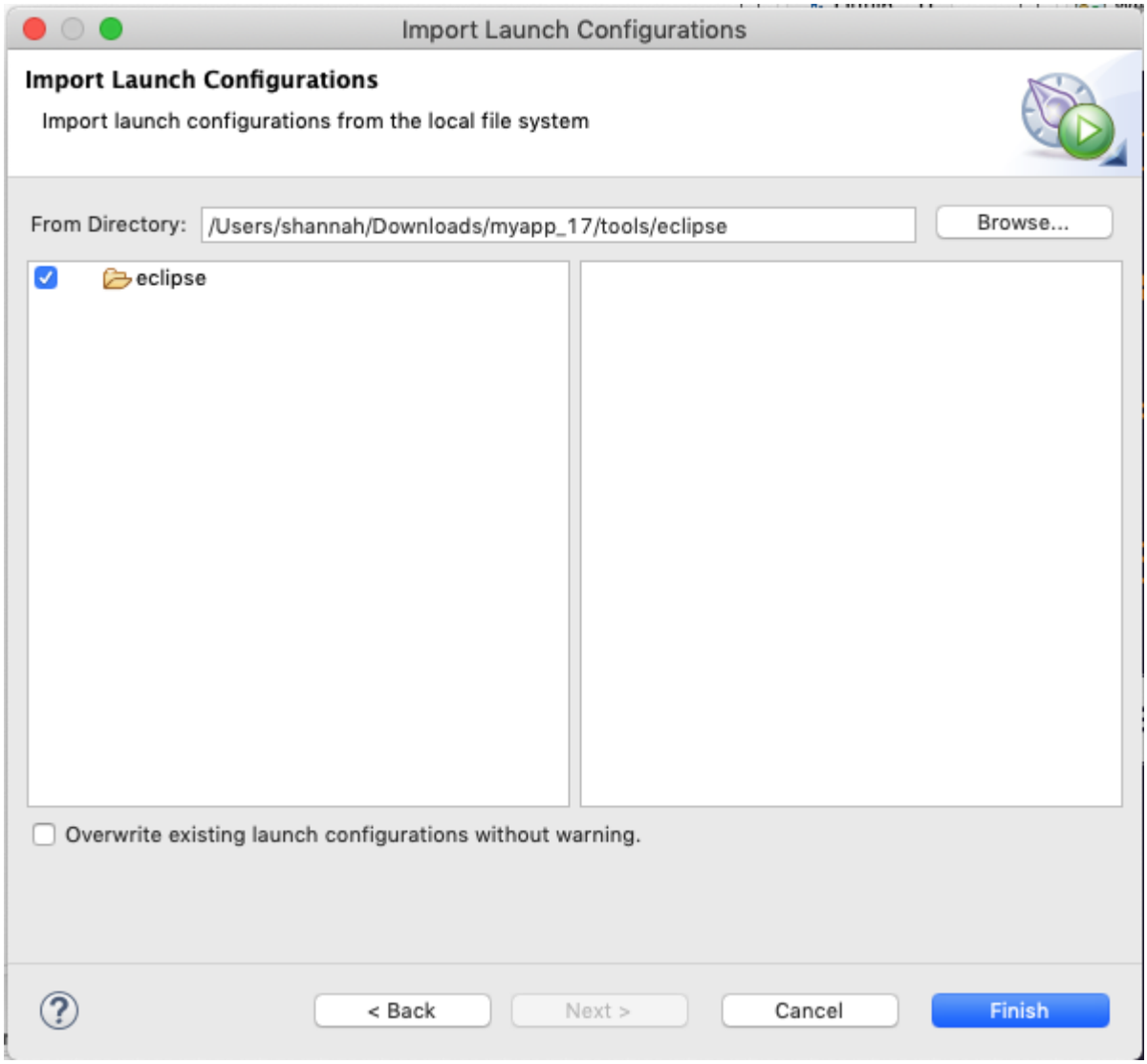
Select *File* > *Import…* again, but this time, in the *Import* dialog, select *Run/Debug* > *Launch Configurations* and click *Next*.



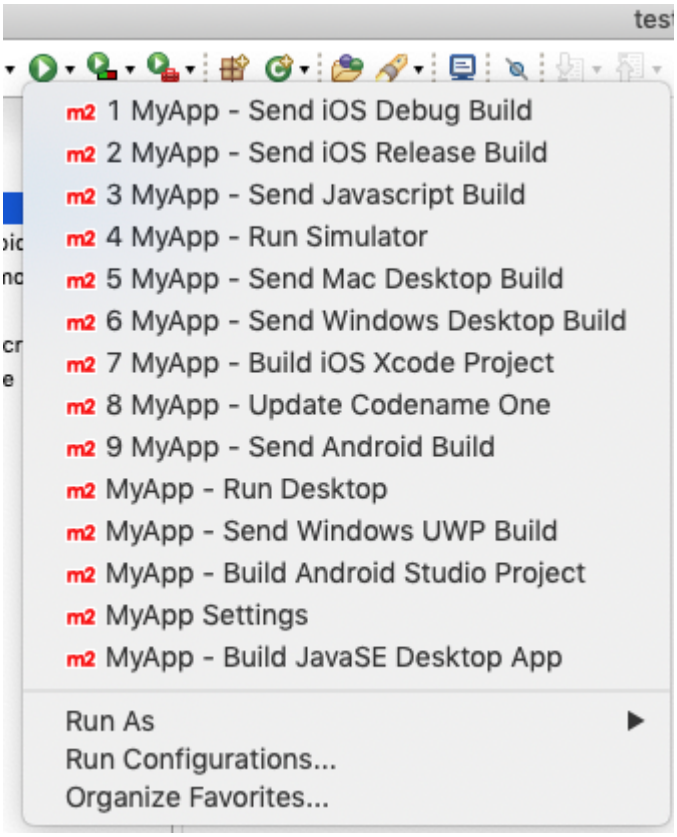
In the next panel, press *Browse…* then select the *tools/eclipse* directory.



Then check the *eclipse* option, and press *Finish*



The "Run" button menu should now include options for all the major build targets. You can see them by pressing on the *Run* button in the toolbar:



Select the *MyApp - Run Simulator* option from this menu.

If all goes well it should open in the Codename One simulator.

Example: Migrating kitchen sink app

Consider a concrete example, now. Download the KitchenSink Ant project from here [https://github.com/codenameone/KitchenSink/archive/refs/tags/v1.0-cn7.0.11.zip] and extract it.

The following is a bash script that uses curl to download this project as a zip file, and then converts it to a fully functional Maven project:

```
CN1_VERSION=7.0.210
curl -L https://github.com/codenameone/KitchenSink/archive/v1.0-cn7.0.11.zip > master.zip
unzip master.zip
rm master.zip
mvn com.codenameone:codenameone-maven-plugin:${CN1_VERSION}:generate-app-project \
  -DarchetypeGroupId=com.codename1 \
  -DarchetypeArtifactId=cn1app-archetype \
  -DarchetypeVersion=${CN1_VERSION} \
  -DartifactId=kitchensink \
  -DgroupId=com.example \
  -Dversion=1.0-SNAPSHOT \
  -DinteractiveMode=false \
  -DsourceProject=KitchenSink-1.0-cn7.0.11
```



This command is formatted for the bash prompt (for example, Linux or Mac). It will work on Windows also if you use bash. If you're on Windows and are using

> PowerShell or the regular command prompt, then you'll need to modify the command slightly. In particular, the entire command would need to be on a single line. (Remove the '\' at the end of each line, and merge lines together, with space between the command-line flags)

This will generate the maven project in a directory named "kitchensink" in the current working directory because of the `-DartifactId=kitchensink` directory.

1.2.4. Adding project dependencies



For the easiest and recommended approach to adding dependencies to your project, skip to Section 1.2.4.5, "Managing add-ons in Codename One Settings".

One of the reasons to use Maven as the build tool is because it makes the management of project dependencies **almost** trivial. If the library you want to add is on Maven central, then you can copy and paste its `<dependency>` snippet into your pom.xml file and you're good to go. Maven does the rest.



See Introduction to the Dependency Mechanism [https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html] on the Maven website for a gentle, but comprehensive introduction to Maven dependencies.

With Codename One projects, there are a few caveats (see Section 1.2.4.4, "The compliance check"), and a few added niceties that make it easier to find and install add-on libraries in your project (see Section 1.2.4.5, "Managing add-ons in Codename One Settings").

Which `pom.xml` to add the `<dependency>` snippet to

Suppose you have a Maven `<dependency>` snippet that you've copied from Maven central, and it's burning a hole in your clipboard while you're trying to figure out where to paste it into your project. Codename One application projects, being multi-module projects, have more than one `pom.xml` file; One per module.

Question: Which pom.xml file should you paste the snippet into?

Answer: common/pom.xml (almost always).

The "common" module is where all your Codename One application resides. It houses your Java and Kotlin files, your CSS files, your GUI builder files, your Codename One configuration files (that is, `codenameone_settings.properties`). Pretty much everything. The things you'd place in the other modules (for example, `Java SE`, `ios`, etc.) are your platform-specific native interface implementations; And in many applications you won't need any of that.

When adding dependencies into your app, you'll therefore almost always place them inside the pom.xml file for the "common" module.



You can add dependencies without editing XML by launching Codename One Settings with `mvn cn1:settings`. See Section 1.2.4.5, "Managing add-ons in

> Codename One Settings”.

When to use the "other" pom.xml Files

The instructions say that you **almost** always add dependencies in the common/pom.xml file. What are the other modules' pom.xml files for, and when do you need to change them, or add dependencies to them?

Here’s an overview:

%PROJECT_ROOT%/pom.xml

The root pom.xml file is the parent module of all other modules. Anything you add here will be inherited by all the modules. It can be helpful to use `<dependencyManagement>` and `<pluginManagement>` sections in this file to merge versions for dependencies and plugins project-wide. This is also a good place to add project meta-data like `<developers>`, `<scm>`.

Java SE/pom.xml

Any dependencies that are only required for native implementations on the Java SE platform can be added here. Dependencies added to this project aren’t subject to the compliance check.

Additionally, this module handles the build toolchain for the Java SE platform. This includes Mac and Windows Desktop builds, as well as Java SE desktop builds. If you want to customize the build workflow for any of these targets, you would do so by adding plugin executions in this pom.xml file.

android, ios, win, and JavaScript

These modules don’t use Maven for their dependencies (Android may deserve a small asterisk here, but that’s complicated), so the primary thing you’d want to **change** in these pom.xml files are the build toolchain for those targets. For example, You might add plugin executions for your CI workflow on builds targeting these particular platforms.

Example: Adding Google Maps dependency via Maven central

Add the GoogleMaps library to your app as a maven dependency.

As described in the GoogleMaps cn1lib README [https://github.com/codenameone/codenameone-google-maps#maven-dependency], the dependency snippet is:

```
<dependency>
  <groupId>com.codenameone</groupId>
  <artifactId>googlemaps-lib</artifactId>
  <version>1.0.1</version>
  <type>pom</type>
</dependency>
```

You should, but, look on Maven central [https://search.maven.org/artifact/com.codenameone/googlemaps-lib]

to see what the latest version number is, and substitute that version into the `<version>` tag of the snippet.

Copy and paste this snippet into the `<dependencies>` section of your common/pom.xml file. And save it.



The common/pom.xml file has **a lot** of existing configuration in it, and it may not be clear, on first glance, where the `<dependencies>` tag is located. A simple "find" for `<dependencies>` may also lead you astray, since there are a few `<profile>` tags which also include `<dependencies>` sections.

The **correct** `<dependencies>` section, is located near the top of the file. You can identify it because it will include the following comment:

```
<!-- INJECT DEPENDENCIES -->
```

This is a special marker that's used by some Codename One tooling to help it locate the optimal place to inject dependencies.

Don't REMOVE THIS COMMENT. Just add your dependency snippet somewhere before or after it.

Compatibility with Codename One

You can paste any Maven dependency snippet you like into your project, but libraries that haven't been specifically developed for Codename One might not be compatible. See Appendix C, *API*. If you're unsure whether a library is compatible, you could add the dependency and try to use it in your app. If it isn't compatible, it will fail when you try to build the app, during the compliance check.

The easiest way to find Codename One libraries is to use the Extensions view in Codename One Settings. These libraries were built specifically for Codename One. Settings warns before installing catalog entries known to target older Codename One releases.

The compliance check

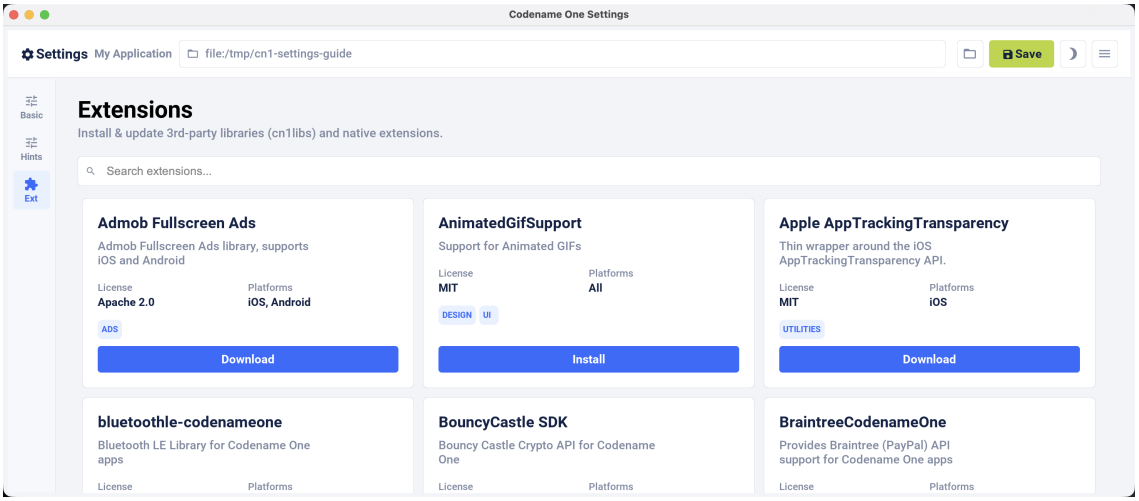
All application code in the common module of your Codename One project must be compatible with Codename One. This includes all dependencies. When you build your project, it will perform a compliance check to ensure that no code uses unsupported APIs. (See Appendix C, *API*).

If the compliance check fails (that is, the app uses unsupported APIs), the build will fail. The error log should provide some clues about where the offending code resides.

Managing add-ons in Codename One Settings

Launch Codename One Settings from the project root, then select **Extensions** in the left navigation rail:

```
mvn cn1:settings
```



As an example, install the "Google Maps" library.

Type `Maps` in the search field and locate `Codename One Google Native`, the Google Maps library.

Click **Install** or **Download**, depending on how the catalog entry is distributed.

Settings shows progress while it installs the extension. For an installed extension, click the same action area and confirm **Uninstall** to remove it.

How Codename One Settings handles Maven dependencies

Many extensions listed in Settings are deployed as cn1lib bundles. Others are deployed on Maven Central and **could** be installed by adding a snippet to `pom.xml` (as described in Section 1.2.4.2, “Example: Adding Google Maps dependency via Maven central”).

Settings shields you from the details of how it installs extensions. For extensions deployed on Maven Central, it adds the dependency directly to your project’s `common/pom.xml`. For extensions distributed as cn1lib bundles, it uses the `install-cn1lib` Maven goal.

You shouldn’t need to worry about this, as it happens seamlessly. If you’re curious, you can look at the `<dependencies>` section of your common/pom.xml file to see the added `<dependency>` tag after you install an extension.

If the Settings window opens blank

Settings is a Codename One app hosted in a native window, so a window that opens but shows nothing can come from the window layer, the display scale, the theme, or font loading. Capture the render state and attach it to a bug report rather than guessing:

```
mvn cn1:settings -Dsettings.diagnostics=cn1-settings-diagnostics.txt
```

The file records the OS and JDK, the HiDPI scale the JVM reports, the window and canvas geometry,

the size Codename One laid the form out at, and the colours and font metrics resolved for the theme. Add `-Dsettings.screenshot=shot.png` to also write the window contents; a second `shot.onscreen.png` captures the pixels actually on screen, which can differ from the offscreen paint. The launch log is at `~/.codenameoneSettings/settings.log`.

Installing legacy cn1libs

The recommended approach for installing add-ons is to use Codename One Settings or add the Maven dependency to `common/pom.xml`. In some situations neither method is available. For example, you might have a legacy cn1lib file that isn't listed in the extension catalog or deployed to Maven Central.

In cases like this you can use the `install-cn1lib` Maven goal to install it as follows:

```
mvn cn1:install-cn1lib -Dfile=/path/to/yourlibrary.cn1lib
```

1.3. Updating Codename One

Codename One releases new versions weekly on Maven central. It's recommended that you stay up to date with the latest version as much as possible to ensure compatibility with the Codename One build server, which is always running the latest version.

You can use the update goal to update both the Codename One libraries, and the Codename One dependencies in your project.

For example:

```
mvn cn:update
```

CLI

Or you can use the `run.sh/run.bat` script to run this goal as follows:

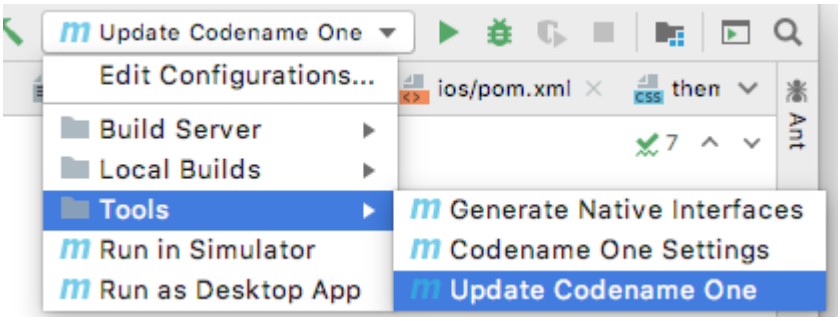
```
./run.sh update
```



Use `run.bat update` instead on Windows

IntelliJ

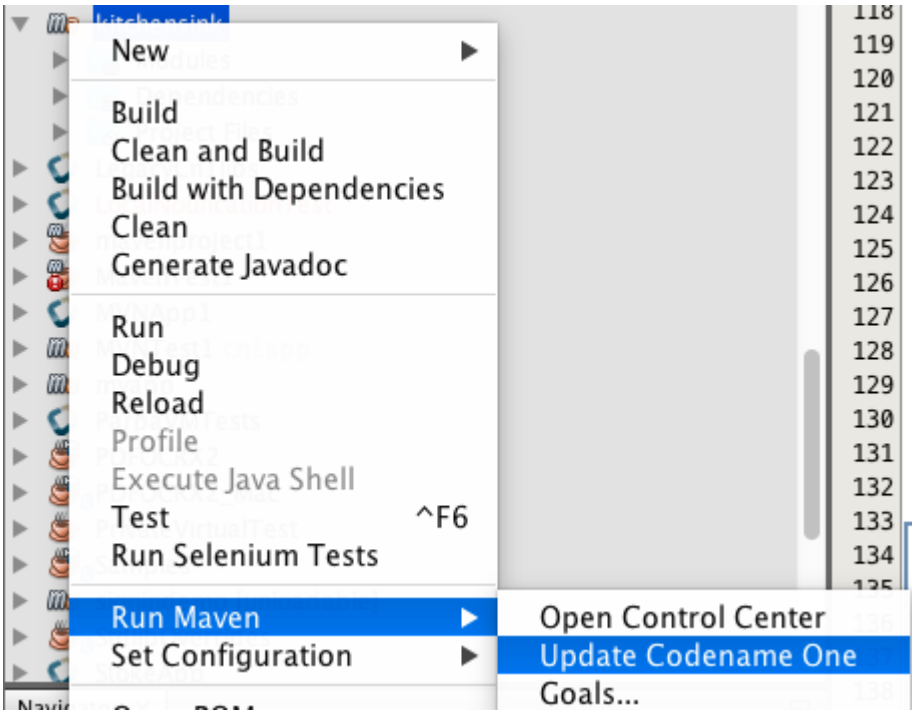
Or you can click on the "Configuration" menu, and select "Tools" > "Update Codename One" as shown here:



Then press the ▶ button.

NetBeans

Or you can right-click on the project in the project inspector, and select "Run Maven" > "Update Codename One" as shown here:



Then press the button.

Manually updating the pom.xml file

You can also update your Codename One dependencies manually by modifying the `cn1.version` and `cn1.plugin.version` properties defined in your project's `pom.xml` file.

For example, Open the `pom.xml` file, and look for the following:

```
<cn1.plugin.version>7.0.210</cn1.plugin.version>
<cn1.version>7.0.210</cn1.version>
```

Change these values to reflect the latest version of the `codenameone-maven-plugin` found here [https://search.maven.org/artifact/com.codenameone/codenameone-maven-plugin].



Updating the `cn1.plugin.version` and `cn1.version` properties manually will update the Maven dependencies for your project but it won't update the other Codename One tools such as the GUI builder, and the Build Server Client, which are managed outside of Maven. You should use the Section B.23, "Update Codename One (`update`)" as described at the beginning of this section to perform a "full" update.

1.4. Creating project templates

A project template is a Codename One application project that can be used as a starting point for building a Codename One application. Codename One initializr [https://www.codenameone.com/initializr] uses project templates to generate starter projects for Codename One applications. You can also use the Section B.5, "Generate app project (`generate-app-project`)" goal to generate starter projects from templates directly in Maven.

Any Codename One project can be converted into a project template.

1.4.1. Converting a Codename One application project into a project template

If you have an existing maven Codename One application project, you can convert it into a project template by adding a file named `generate-app-project.rpf` in the root directory of the project.

The contents of this file should look like:

```
template.mainName=$YOUR_PROJECT_MAIN_NAME
template.packageName=$YOUR_PROJECT_PACKAGE_NAME

[dependencies]
====
... YOUR PROJECT MAVEN DEPENDENCIES ...
====

[parentDependencies]
====
... YOUR PARENT PROJECT MAVEN DEPENDENCIES ...
====
```

Where you make the following substitutions:

$YOUR_PROJECT_MAIN_NAME

This should be the value of the `codename1.mainName` property in the project's `codenameone_settings.properties` file.

$YOUR_PROJECT_PACKAGE_NAME

This should be the value of the `codename1.packageName` property in the project's `codenameone_settings.properties` file.

... YOUR PROJECT MAVEN DEPENDENCIES...

Paste any maven dependencies that the project requires into this section. *These will be injected into the* `<dependencies>` *section of the common/pom.xml file*.

... YOUR PARENT PROJECT MAVEN DEPENDENCIES...

Paste any maven dependencies that the parent project requires into this section. *These will be injected into the* `<dependencies>` *section of the pom.xml file*.

See Section B.5.4.2, "Sample `generate-app-project.rpf` file" for a more concrete example of the `generate-app-project.rpf`.

1.4.2. Test your project template

You can test your project template by using it as the `sourceProject` parameter for the `generate-app-project` goal. See Section B.5, "Generate app project (`generate-app-project`)".

1.4.3. Add your template to Codename One intializr

If you have a project template that you want to share with the community, please file an issue in the Codename One issue tracker [https://github.com/codenameone/CodenameOne/issues] with a link to a GitHub Repository of your project template, and request to have it added Codename One initializr [https://www.codenameone.com/initializr].

1.5. Codename One libraries

A Codename One Library (cn1lib) is a module that can be distributed and added to Codename One applications to add functionality. It can be distributed as a self-contained bundle (a file with the.cn1lib extension), or deployed on Maven central to be included in application projects as a pom dependency.

A cn1lib may contain any of the following:

1. Cross-platform Java classes.
2. Native code that targets specific platforms.
3. Build hints, which will affect how projects will be built that include this library. These can contain things like Gradle dependencies on Android, Cocoapods dependencies on iOS, and other hints to affect the build-server process.
4. CSS files.

`.cn1lib` vs `.jar`

You may be wondering why the .cn1lib format is even necessary. Why not just distribute libraries as .jar files? The .cn1lib format offers several advantages over the plain .jar format:

1. **cn1libs can contain platform-specific native sources** that make use of native APIs on the various platforms. For example, they can contain Objective-C code which will be compiled on the build server when deploying on iOS.

2. **Codename One library projects perform a compliance check** at the time that the library is compiled to ensure it only uses supported Codename One APIs. This provides a sort of "certification" that the library will be compatible with Codename One application projects.
3. **Codename One libraries can include CSS files and build hints** which will be appended to the build hints of application projects when they're built.

All that said, you **can** still distribute libraries as plain old jars and include them in your Maven Codename One projects as **jar** dependencies. Codename One application projects will perform an additional compliance check to ensure that the jar is compatible, and the build will fail if it uses APIs that aren't available in Codename One.



Codename One supports a subset of JavaSE8, as well as addition APIs for accessing device functionality and building beautiful user interfaces. See the Codename One javadocs [https://www.codenameone.com/javadoc/] for a definitive list of supported APIs.

1.5.1. Creating a library project

Use the `cn1lib-archetype` for generating a new Codename One library project as follows:

Command line

```
mvn archetype:generate \
  -DarchetypeArtifactId=cn1lib-archetype \
  -DarchetypeGroupId=com.codenameone \
  -DarchetypeVersion=LATEST \
  -DgroupId=com.example.mylib \
  -DartifactId=mylib \
  -Dversion=1.0-SNAPSHOT \
  -DinteractiveMode=false
```



This command is formatted for the bash prompt (for example, Linux or Mac). It will work on Windows also if you use bash. If you're on Windows and are using PowerShell or the regular command prompt, then you'll need to modify the command slightly. In particular, the entire command would need to be on a single line. (Remove the '\' at the end of each line, and merge lines together, with space between the command-line flags)

In the above snippet you would change the `groupId`, `artifactId`, and `version` properties to reflect your project settings.



You can run the `archetype:generate` goal with as many or few properties as you like, and it will prompt you to enter any properties that are required. For example, You could enter:

```
mvn archetype:generate
```

And then follow the prompts. Or you could enter:

```
mvn archetype:generate -DarchetypeGroupId=com.codenameone \
  -DarchetypeArtifactId=cn1lib-archetype
```



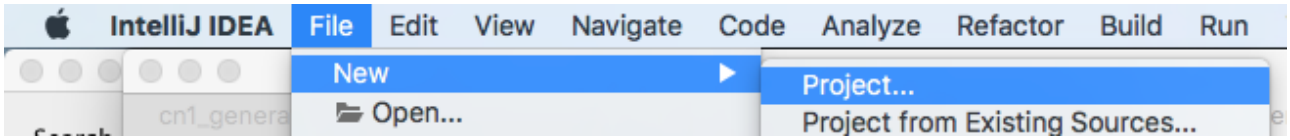
This command is formatted for the bash prompt (for example, Linux or Mac). It will work on Windows also if you use bash. If you're on Windows and are using PowerShell or the regular command prompt, then you'll need to modify the command slightly. In particular, the entire command would need to be on a single line. (Remove the '\' at the end of each line, and merge lines together, with space between the command-line flags)

And follow the prompts. This will, result in fewer prompts because you've already specified the archetype to use.

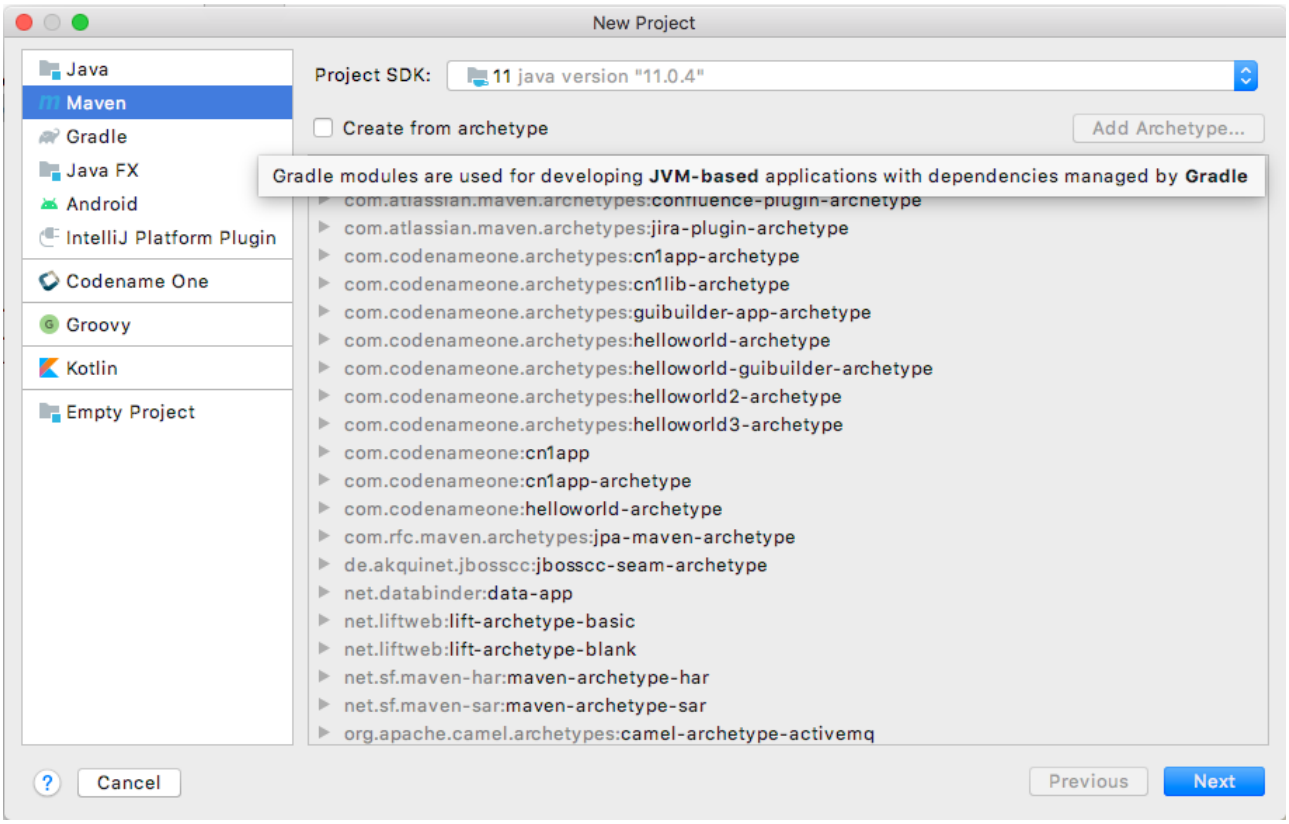
This will create a new project for you in the current directory, in a newly created directory named after the `artifactId` that you entered.

IntelliJ IDEA

1. Select "File" > "New Project"...



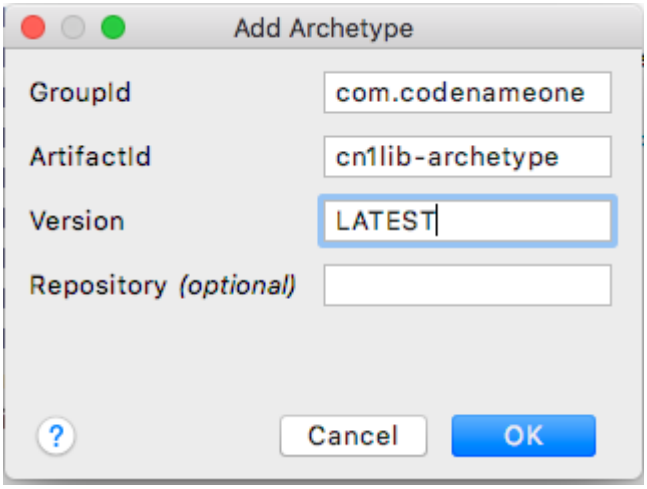
2. Select "Maven" in the left menu.



3. Check the "Create from Archetype" checkbox. 

. This should allow you to choose from of archetypes that are already known to IntelliJ.

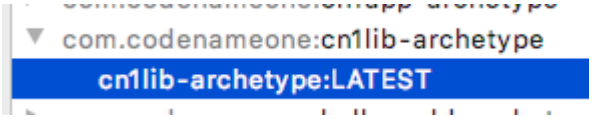
4. If you don't see an option for `com.codenameone:cn1lib-archetype`, then IntelliJ doesn't know about it yet. If, but you **do** see this option, you can skip to the next step. Press the "Add Archetype..." button. This will display a dialog for you to enter the details of the archetype.



Fill in this dialog as shown in the above image. Specifically `groupId=com.codenameone`, `artifactId="cn1lib-archetype,"` and `version="LATEST"`

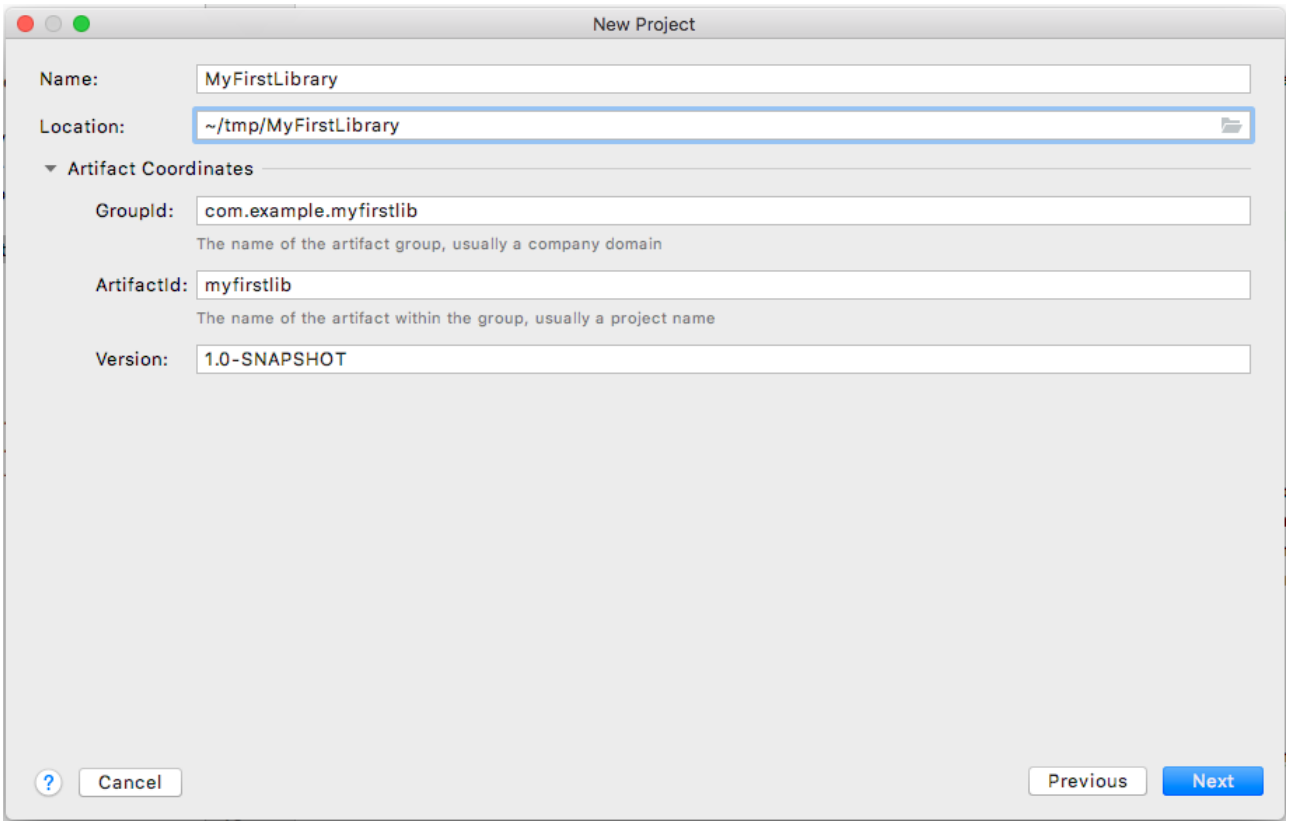
Then press `OK`.

5. Select the option that says "com.codenameone:cn1lib-archetype"



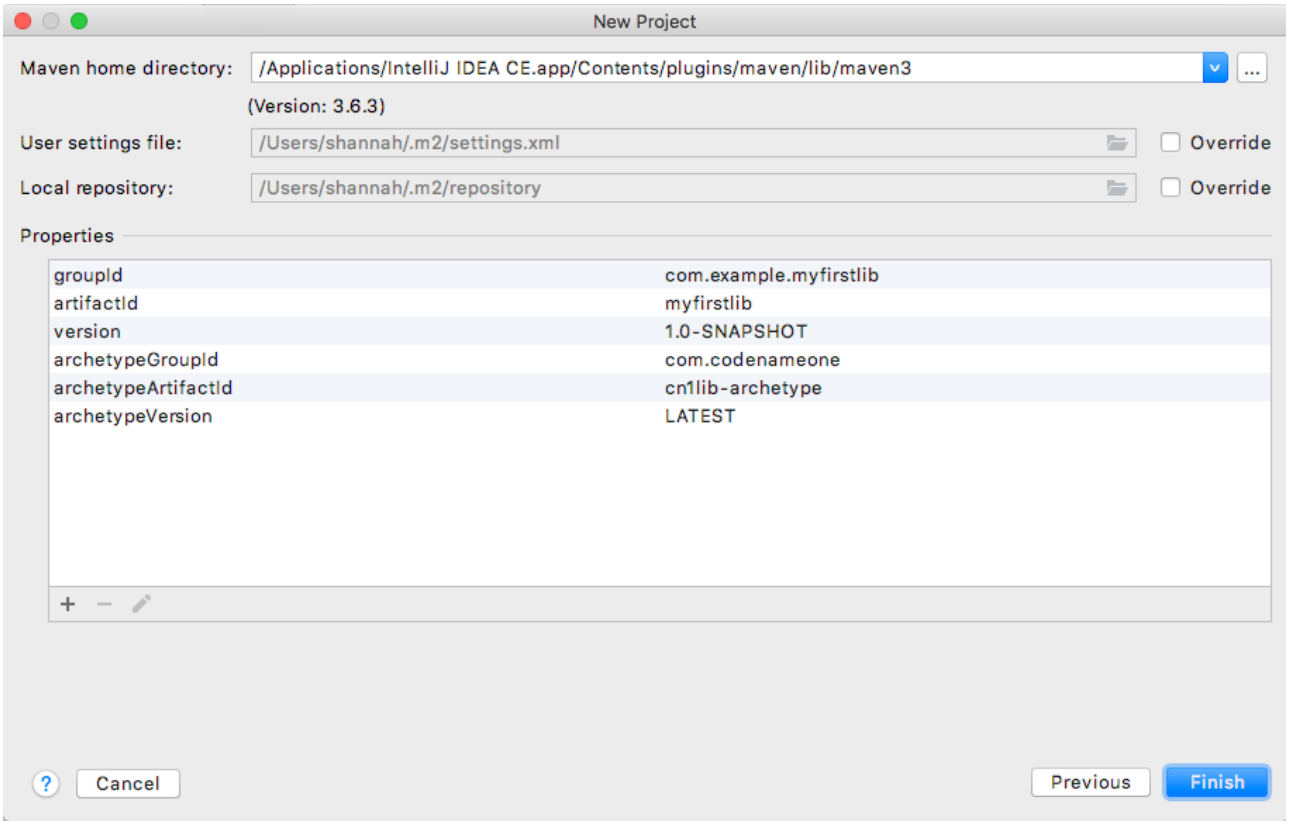
Then press "Next"

6. This will display a form where you can enter the details of your project such as its location (where you want to create the project folder), the name, the artifact ID, and the groupID. Fill in this form as you see fit.



Then click "Next"

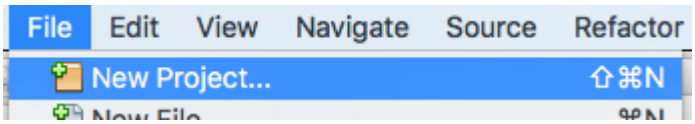
7. The final form in this wizard summarizes the project details and gives you an opportunity to add more properties to pass to the `archetype:generate` goal. In your case you don't need to add any more properties. If the information looks correct, you can press `Next`.



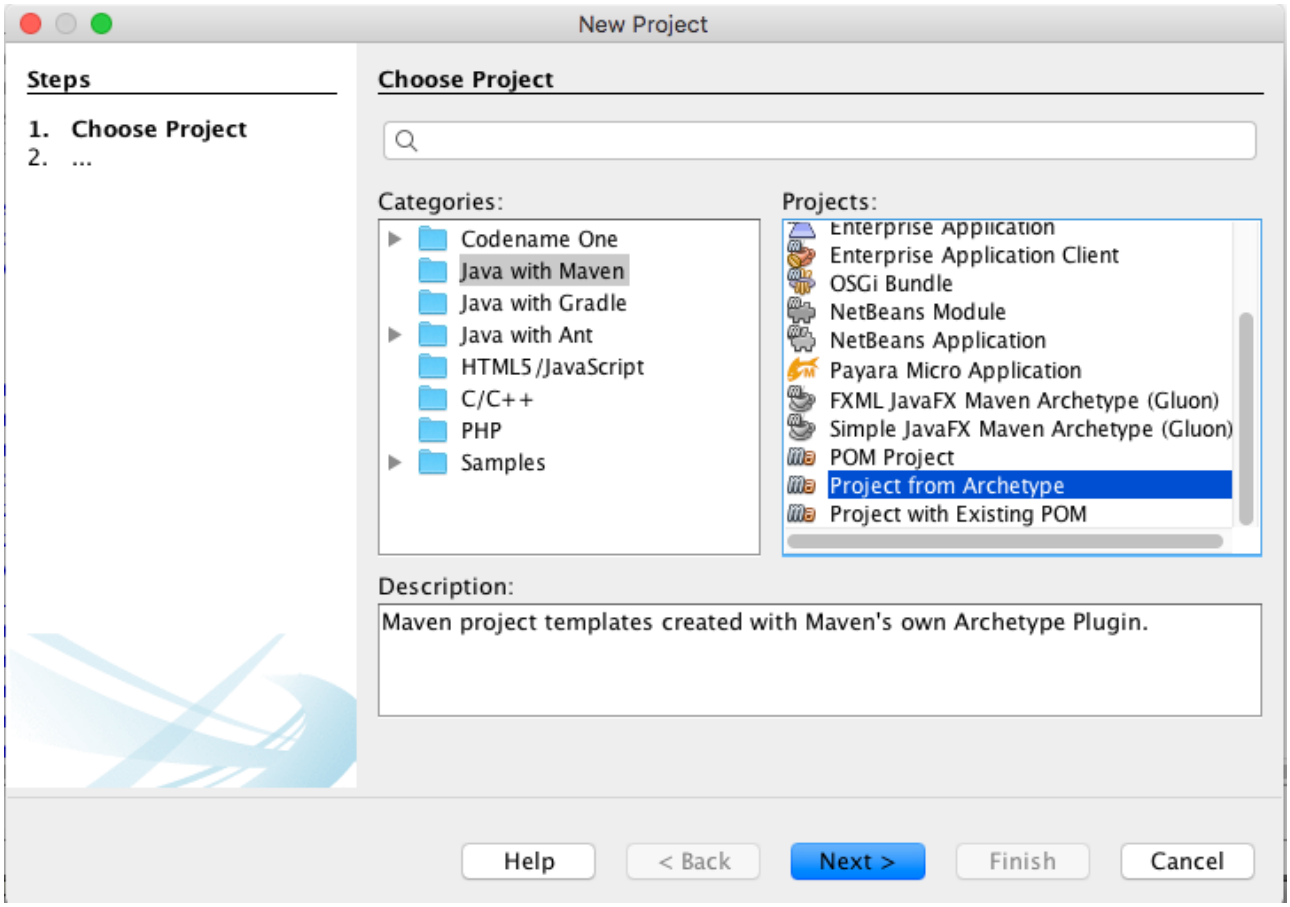
At this point you will be prompted to open the project.

NetBeans

1. Select "File" > "New Project..."

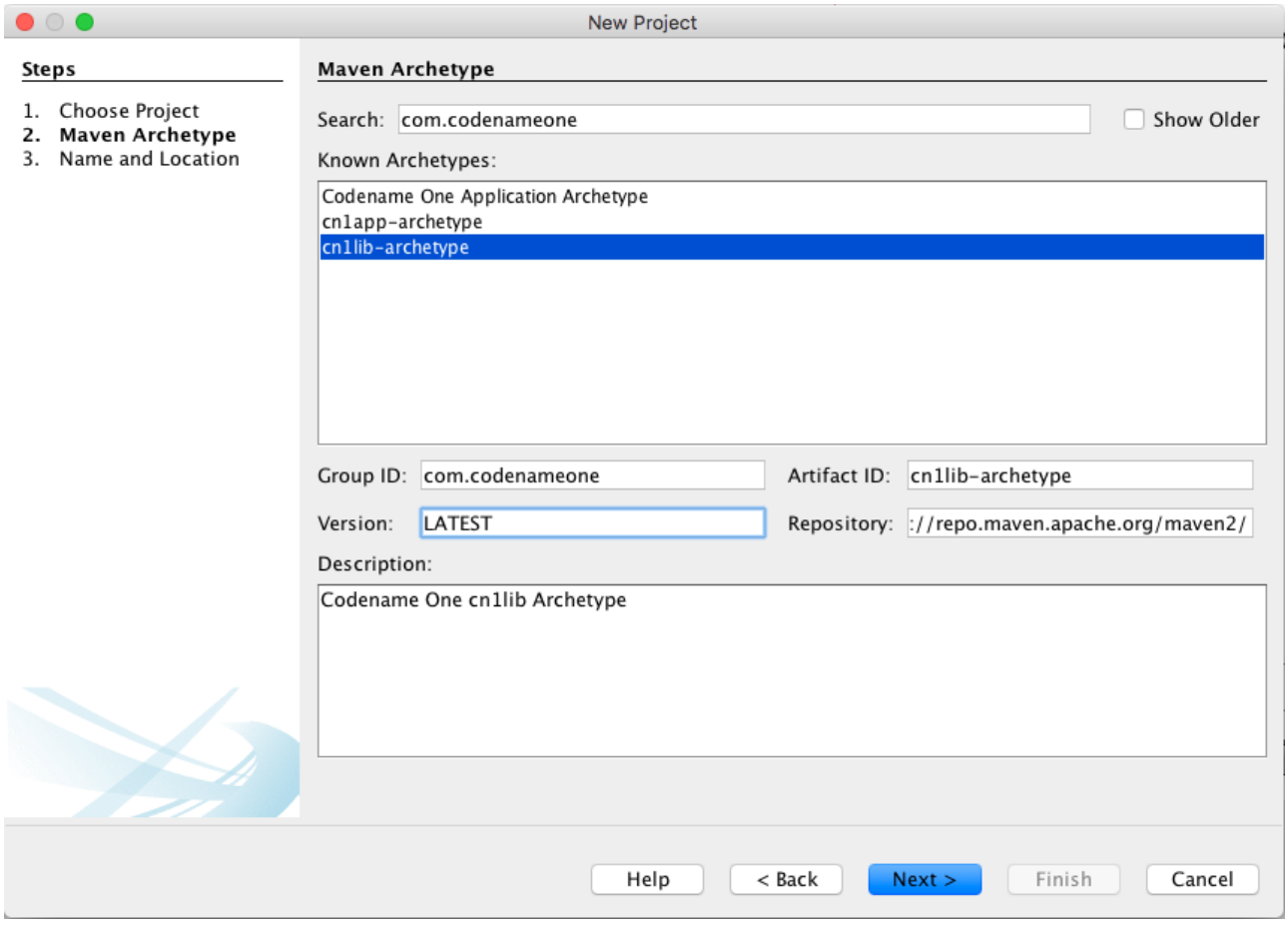


2. In the "New Project" dialog, select "Java with Maven" in the left panel, and "Project from Archetype" in the right panel, as shown below.



Then press "Next"

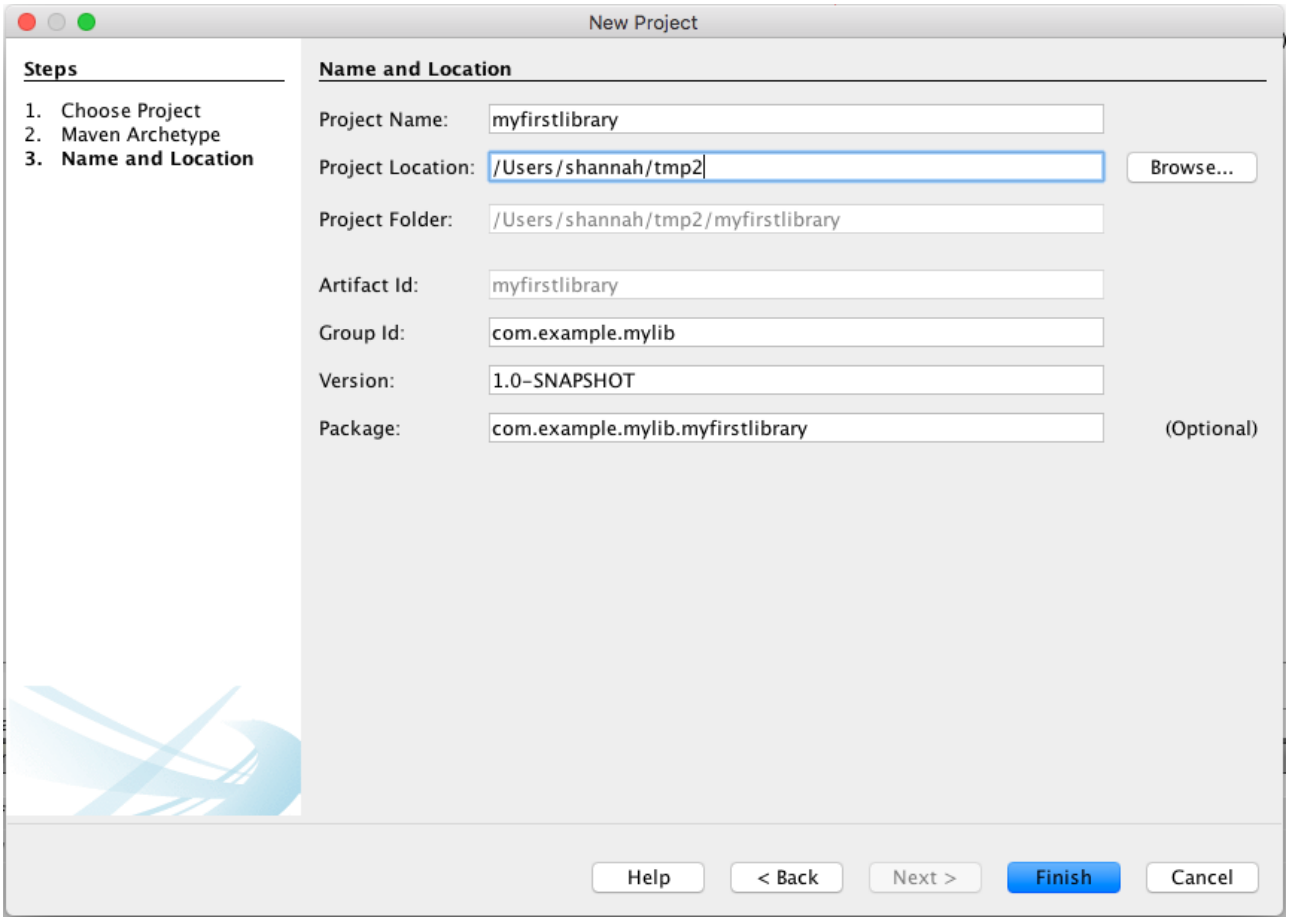
3. This will bring you to the "Maven Archetype" dialog as shown below:



Enter "com.codenameone" or "cn1lib-archetype" into the search field. Then select "cn1lib-archetype" in the "Known archetypes:" panel. This will prefill the **Group ID**, **Artifact ID** and **Version** fields for you. You may want to change **Version** to LATEST to ensure that it tries to use the latest available version of the archetype.

Then click "Next"

4. This will bring you to the "Name and Location" panel of the wizard.



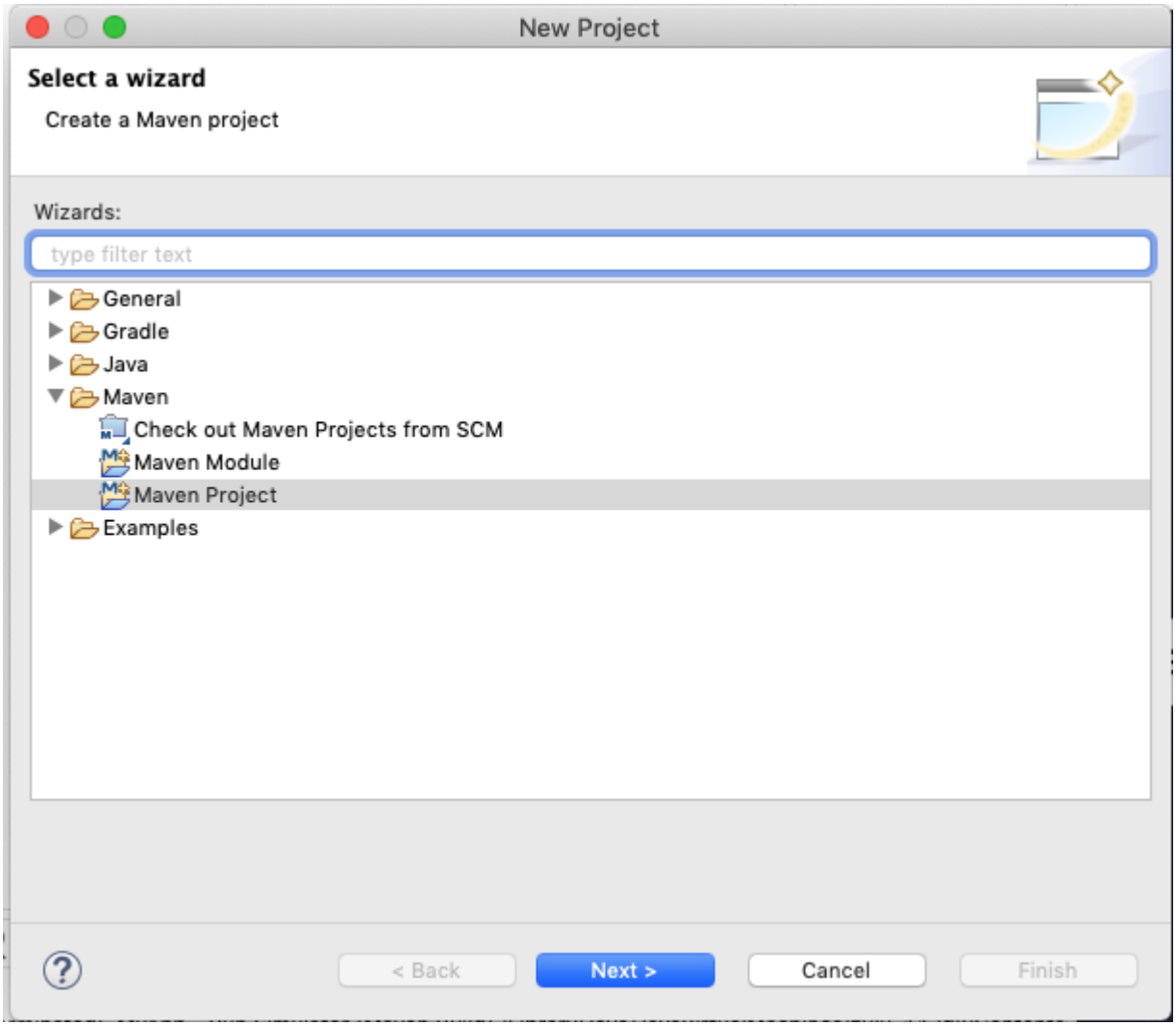
Enter in the project name (which you'll be forced to use as the artifact ID also), project location, groupId, version, and package. The "Package" is unimportant here as it isn't used anywhere in the project.

Once you've entered the information to your liking press the "Finish" button.

This will create a new library project for you at the location you specified.

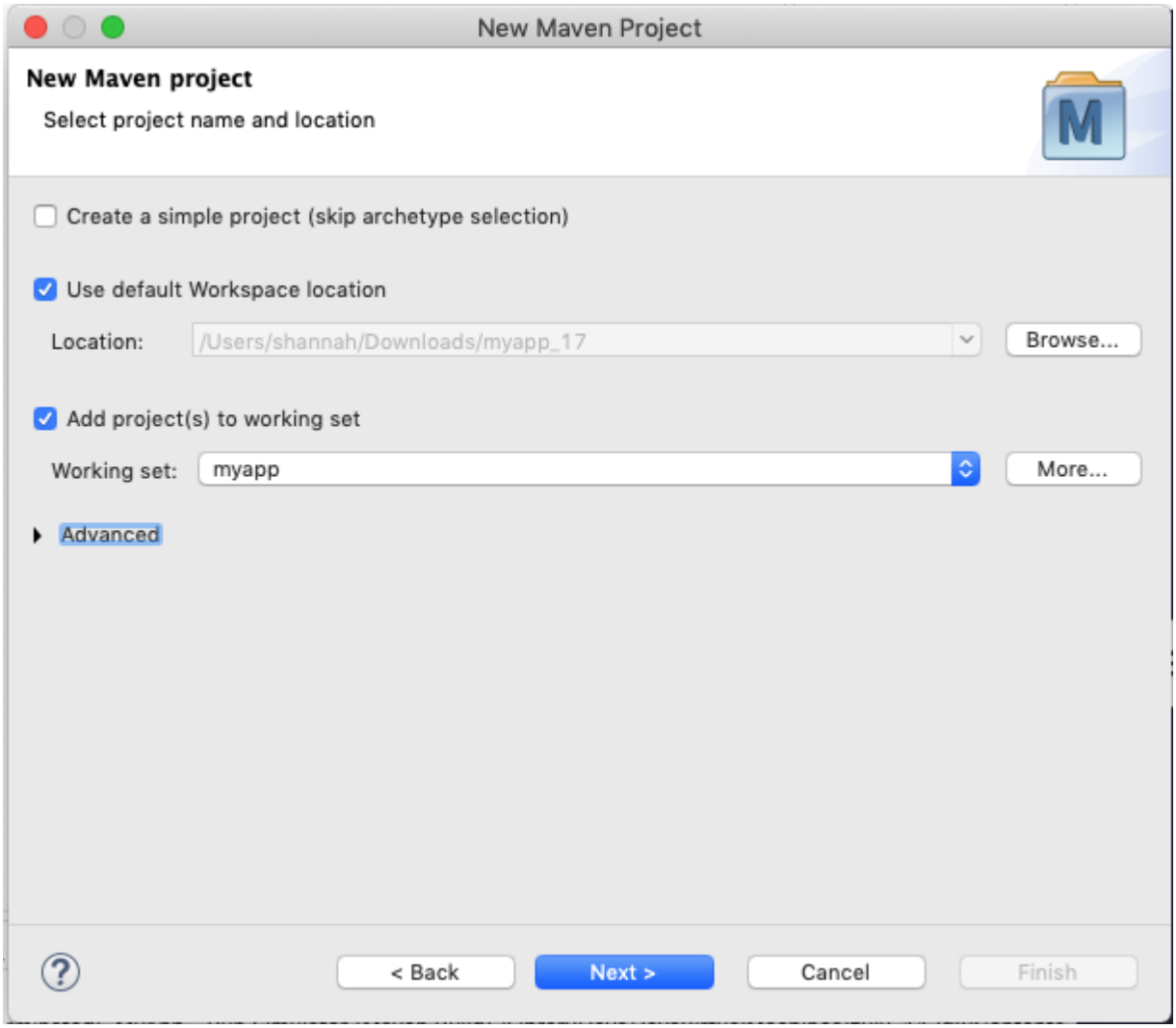
Eclipse IDE

1. Select "File" > "New Project..."
2. In the *New Project* dialog, expand the *Maven* item, and select *Maven Project*

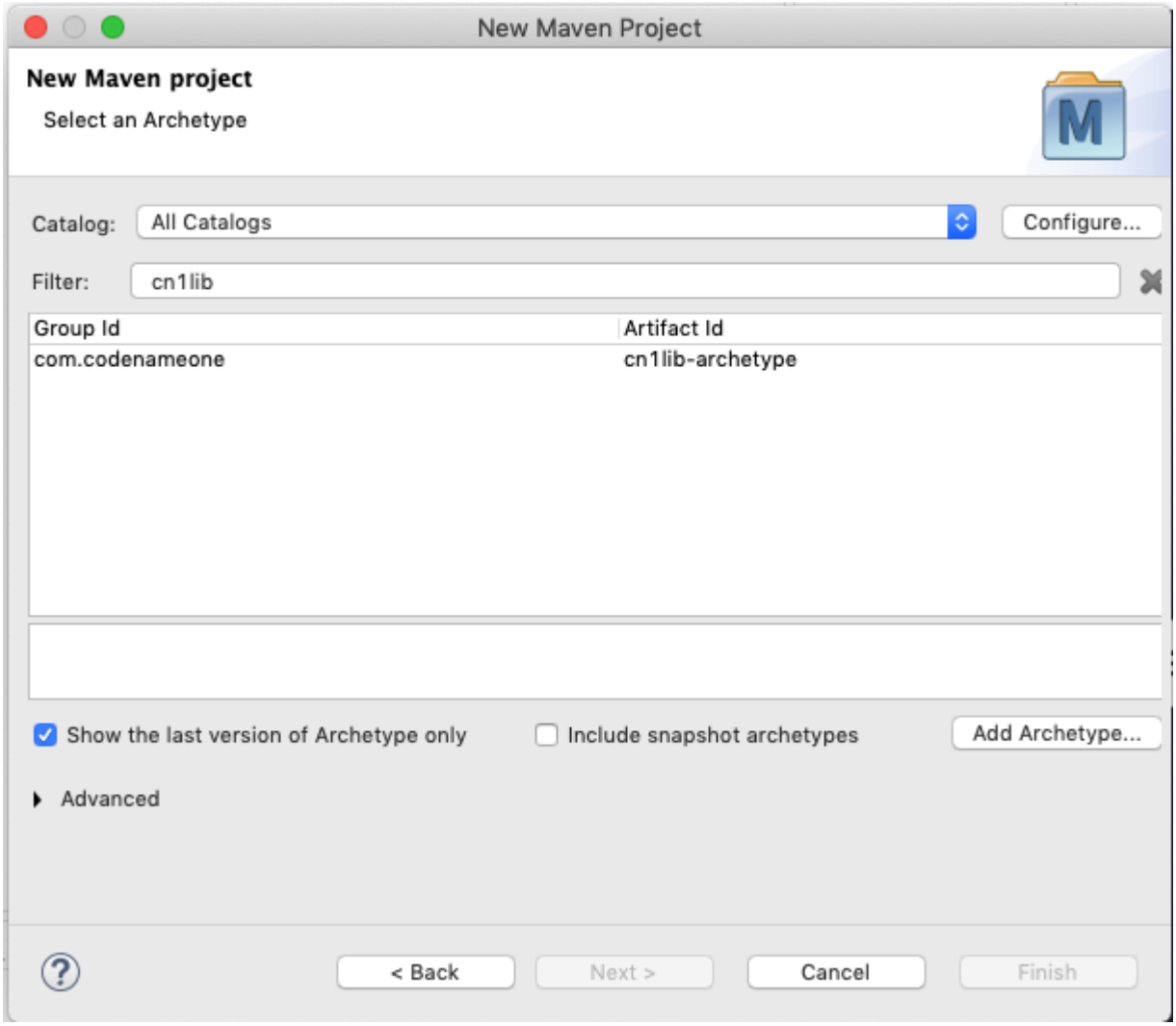


Then press "Next"

3. The next panel will look like the below image. The default settings on this panel should be fine. Press *Next*

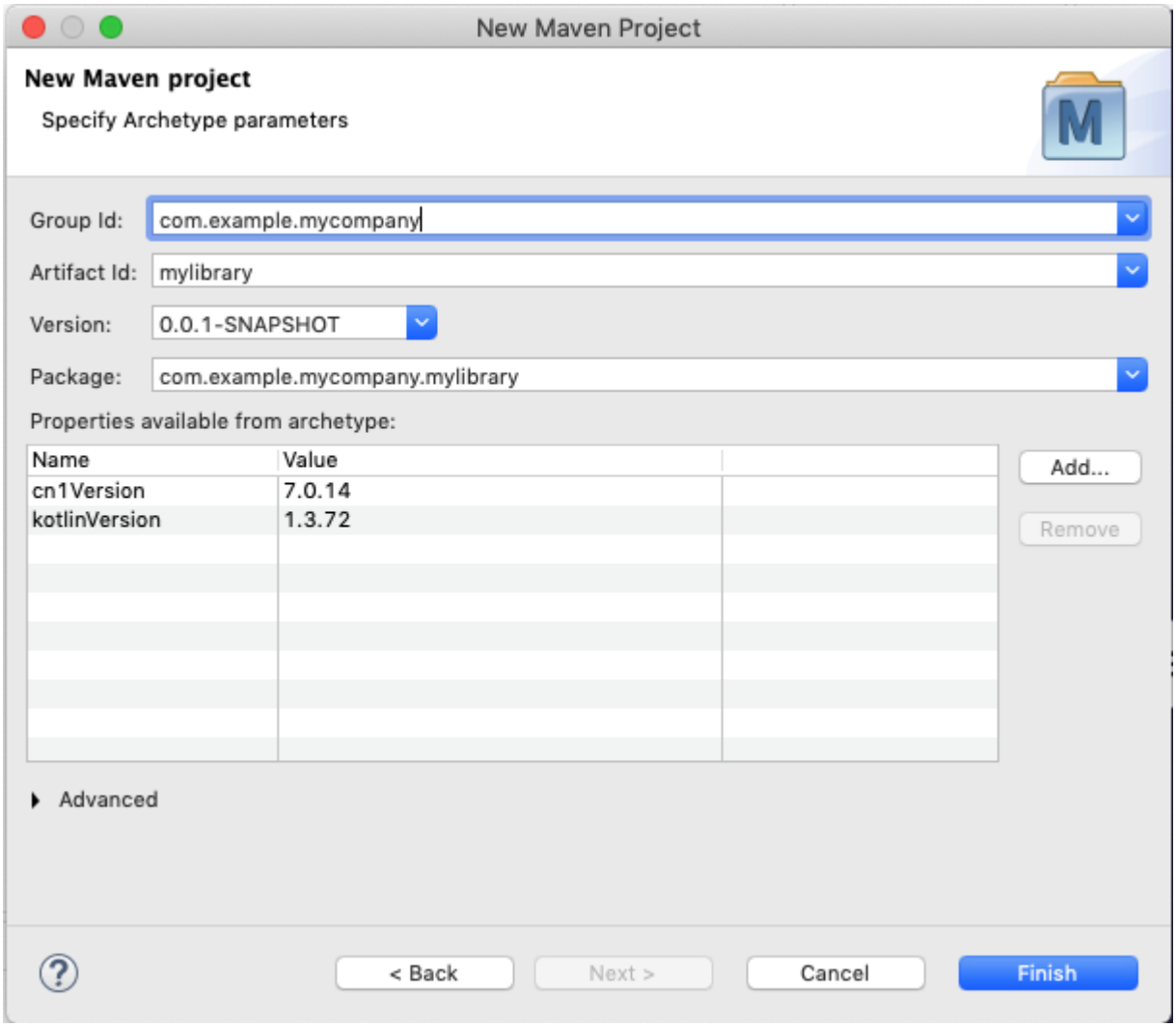


4. In the next panel, enter "cn1lib" in the *Filter* field. After a moment the *cn1lib-archetype* should appear in the area below as shown here:



Select that option, and press *Next*

5. The next panel, allows you to enter your project details, such as group ID, and artifact ID. Your project information here and then press *Finish*.



This will create a new library project for you at the location you specified.

Project structure

Take a look at the project that was created. It's a multi-module Maven project with the following modules:

common

The module where you'll add all your cross-platform code and CSS, and build hint configuration. This module is in the "common" directory of the main project.

Java SE

The module where you can implement native interfaces for the Java SE platform. This module is in the "Java SE" directory of the main project.

ios

The module where you can implement native interfaces for the iOS platform. This module is in the "ios" directory of the main project.

android

The module where you can implement native interfaces for the Android platform. This module is in the "android" directory of the main project.

JavaScript

The module where you can implement native interfaces for the JavaScript platform. This module is in the "JavaScript" directory of the main project.

lib

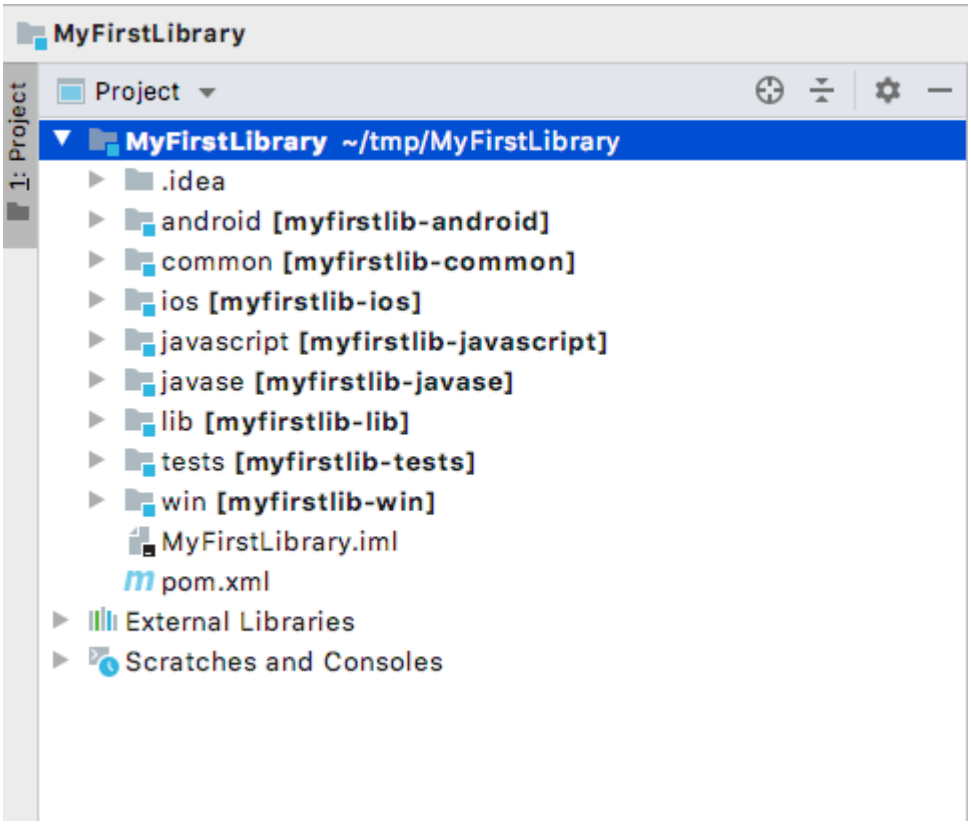
The library module which includes all the other modules as dependencies, and can be used as a pom dependency in Codename One application projects that wish to use this library. This module is in the "lib" directory of the main project.

tests

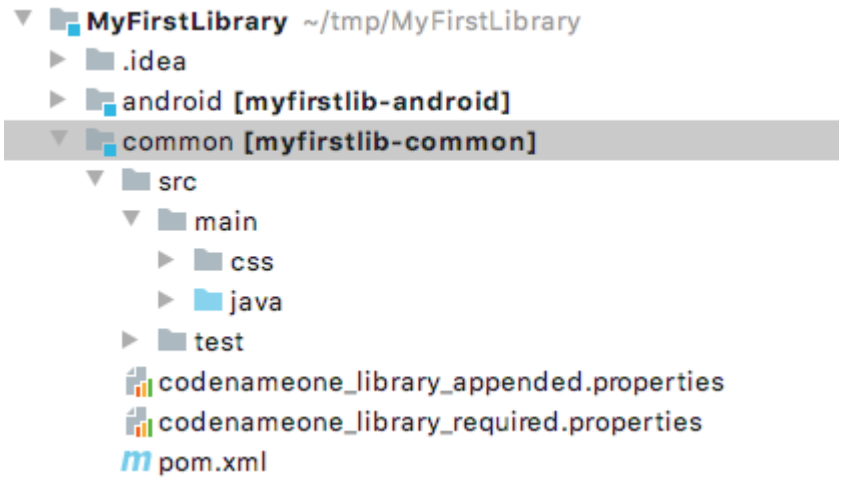
An application project for writing unit tests against your library. This module is in the "tests" directory of the main project.

IntelliJ IDEA

The project inspector will look like:



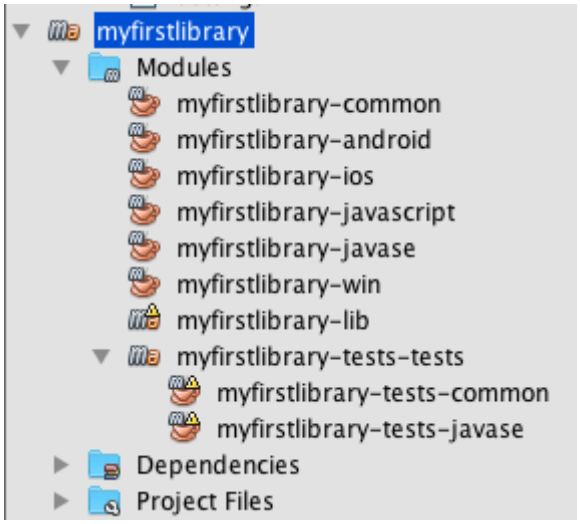
This top-level view of the module structure may seem daunting. Most of your development will occur inside the "common" module. If you expand that module it will look more familiar to developers who have used the old Ant project structure:



Your cross-platform Java source would go in the `common/src/main/java` directory. Your CSS files go in the `common/src/main/css` directory.

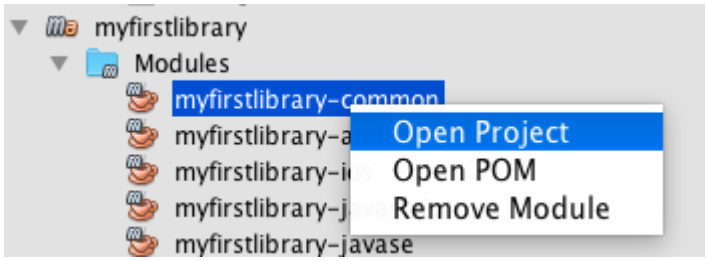
NetBeans

The project inspector will look like:

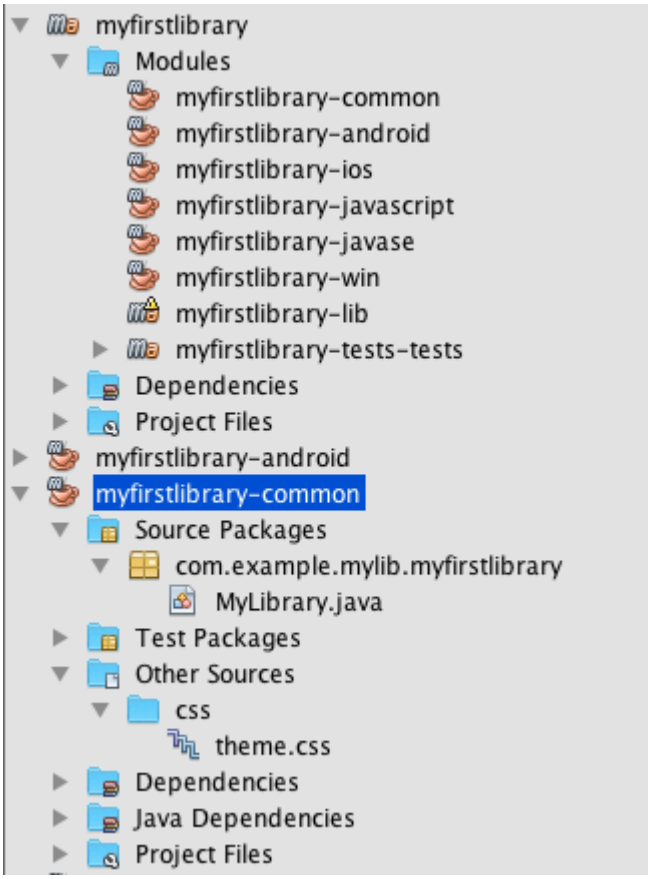


This top-level view of the modules doesn't provide a clear view of the project landscape, but, since 99% of your development will occur inside the `common` submodule. Open that "common" sub-module project as well and take a peek.

Right-click on the "Common" sub-module, and select "Open Project" as shown below:

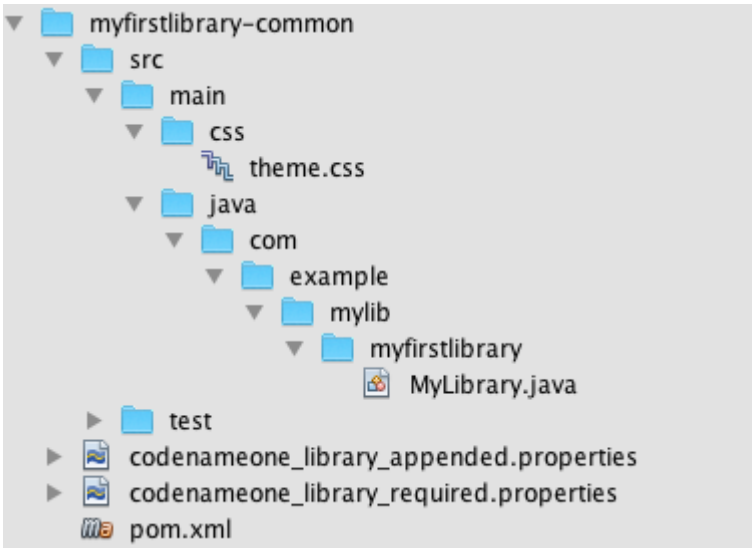


With the common subproject open, the project inspector will look like:



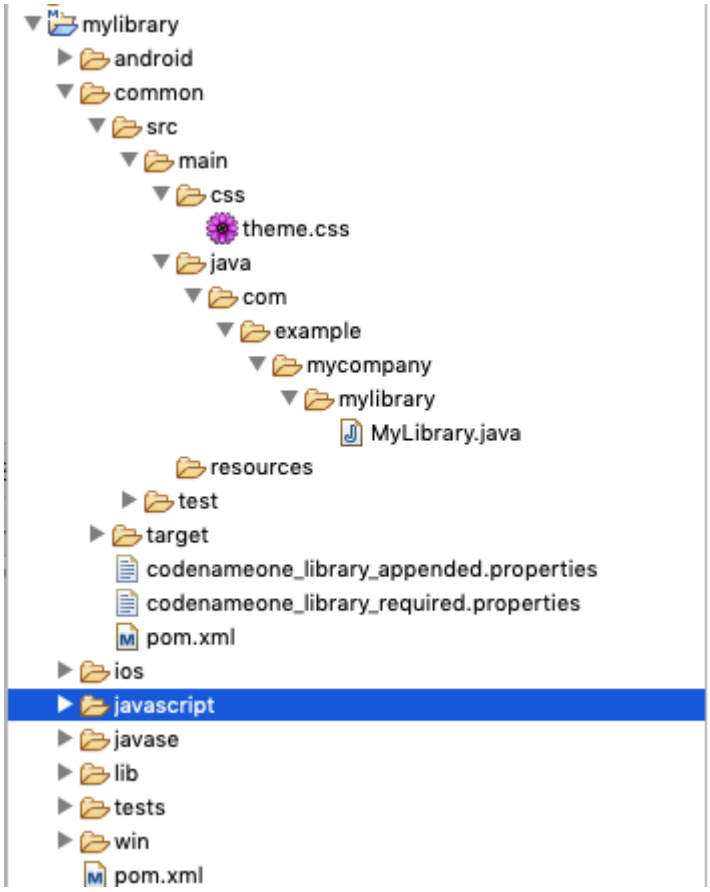
In this screenshot, "Source Packages" and "Other Sources/css" are expanded to highlight where your Java source files and CSS source files will be located.

The project inspector hides a few important files, but, so here is a screenshot of the File inspector for the common project:



Eclipse IDE

The package explorer will look like:



In this screenshot, the *common/src/main/css* and *common/src/main/java* directories are expanded since this is where most of your module source will go.

Command line

If you do a file listing on the project directory, it shows the following:

```
Steves-Mac-Pro:MyFirstLibrary shannah$ find .
.
./tests
./tests/pom.xml
./tests/javase
./tests/javase/pom.xml
./tests/common
./tests/common/codenameone_settings.properties
./tests/common/pom.xml
./tests/common/nbactions.xml
./tests/common/src
./tests/common/src/test
./tests/common/src/test/java
./tests/common/src/test/java/com
./tests/common/src/test/java/com/example
./tests/common/src/test/java/com/example/myfirstlib
./tests/common/src/test/java/com/example/myfirstlib/MyFirstTest.java
./tests/common/src/main
./tests/common/src/main/css
./tests/common/src/main/css/theme.css
```

```
./tests/common/src/main/java
./tests/common/src/main/java/com
./tests/common/src/main/java/com/example
./tests/common/src/main/java/com/example/myfirstlib
./tests/common/src/main/java/com/example/myfirstlib/LibraryTests.java
./tests/cn1libs
./tests/.mvn
./tests/.mvn/jvm.config
./pom.xml
./javase
./javase/pom.xml
./javase/src
./javase/src/main
./javase/src/main/java
./javase/src/main/java/com
./javase/src/main/java/com/example
./javase/src/main/java/com/example/myfirstlib
./ios
./ios/pom.xml
./ios/src
./ios/src/main
./ios/src/main/objectivec
./common
./common/codenameone_library_required.properties
./common/pom.xml
./common/codenameone_library_appended.properties
./common/src
./common/src/test
./common/src/test/java
./common/src/test/java/com
./common/src/test/java/com/example
./common/src/test/java/com/example/myfirstlib
./common/src/test/java/com/example/myfirstlib/MyLibraryTest.java
./common/src/main
./common/src/main/css
./common/src/main/css/theme.css
./common/src/main/java
./common/src/main/java/com
./common/src/main/java/com/example
./common/src/main/java/com/example/myfirstlib
./common/src/main/java/com/example/myfirstlib/MyLibrary.java
./android
./android/pom.xml
./android/src
./android/src/main
./android/src/main/java
./android/src/main/java/com
./android/src/main/java/com/example
./android/src/main/java/com/example/myfirstlib
./lib
./lib/pom.xml
```

```
./MyFirstLibrary.iml
./javascript
./javascript/pom.xml
./javascript/src
./javascript/src/main
./javascript/src/main/javascript
./.idea
./.idea/encodings.xml
./.idea/jarRepositories.xml
./.idea/.gitignore
./.idea/workspace.xml
./.idea/misc.xml
./.idea/compiler.xml
```

This may seem daunting at first, but it's important to realize that 99% of the time, you'll be working in the "common" module - most of the other stuff is boilerplate.

Important files

A few key files in this project that you'll be using more than the others.

pom.xml

The maven configuration file of the root module is where you will set project-wide properties such as the `cn1.version` property, which specifies the version of the Codename One libraries that the module should be compiled against. Periodically, you'll want to update the `cn1.version` property to point to the latest version.

When/if you decide to deploy your module to Maven central, you'll need to add more deployment-related settings in this file.

common/pom.xml

The maven configuration file for the "common" module, which will contain most of your cn1lib's source code, CSS files, and properties files. If your library depends on other libraries or jar files, you'll be adding them as dependencies in this file, and not the root pom.xml file.

common/codenameone_library_appended.properties

This file is where you can specify properties that should be merged with the codenameone_settings.properties of application projects that include this library as a dependency. This is where you would add, for example, Gradle dependencies required for the Android builds, or CocoaPods dependencies that are required for iOS builds.

common/codenameone_library_required.properties

This file allows you to specific build hints that **must** be present in application projects that include this library. If this library requires a particular android build tools version, or a specific Java version, then those requirements should be specified in this file.

Important directories

As mentioned earlier, 99% of all your development will likely occur inside the "common" module.

The other modules are for native implementations of Native interfaces.

common/src/main/java

This is where your cross-platform Java source files will be placed.

common/src/main/css

If your library uses CSS, this is where all CSS-related files will be placed.

common/src/main/resources

Other non-java resources that you want to have included in the classpath.

Building the library

Command line

To build the library, run the "install" goal on the root module as follows:

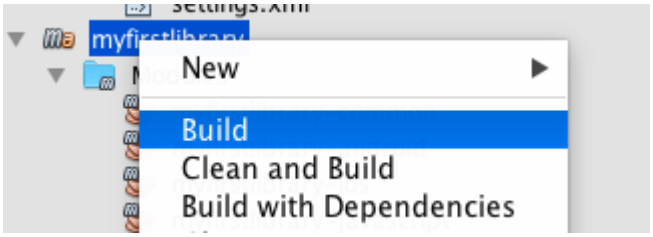
```
mvn install
```

IntelliJ IDEA

Press the "build"  button on the toolbar.

NetBeans

Right-click on the "root" module in the project explorer and select `Build`.



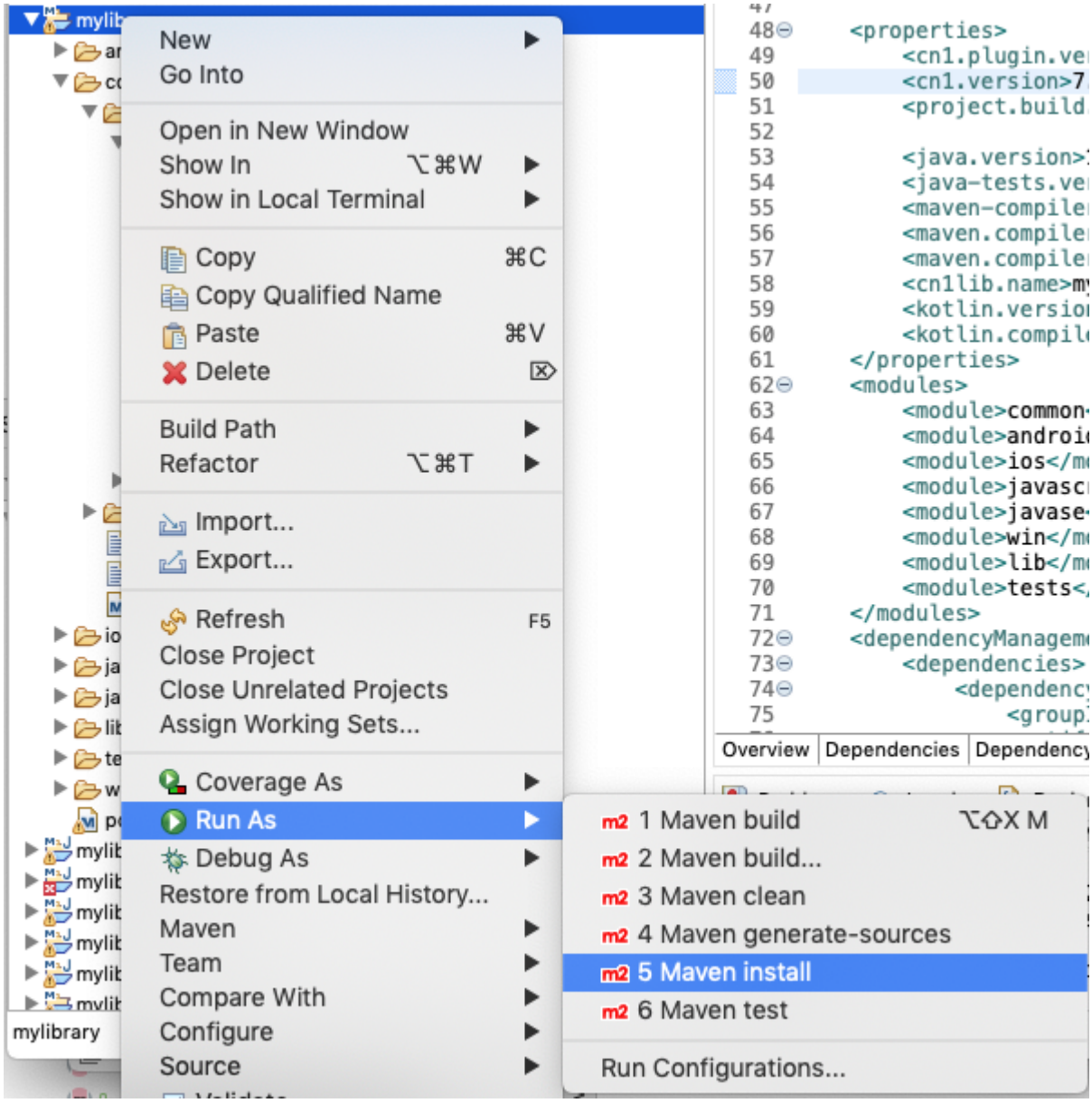


You must build the root module and not one of the submodules.

Or you could have selected the "root" module in the project explorer and pressed the "build"  button on the toolbar.

Eclipse IDE

Right-click on the "root" module in the project explorer and select *Run as* > *Maven Install*





If the build fails for any reason, check to make sure that your project is using the latest version of the Codename One plugin. You can do this by opening the *pom.xml* file, and changing the `cn1.version` and `cn1.plugin.version` properties to reference the latest version. Check for the latest version here [https://search.maven.org/artifact/com.codenameone/codenameone].

Building the Legacy.cnlib file

When using the Maven build tool, you no longer require the.cn1lib file at all. Your library projects can be handled entirely via Maven's dependency mechanism. The preferred way to distribute your libraries is on Maven central, and the preferred way to add a library to an application is via a Maven "pom" dependency.

That being said, you may still want to distribute your library as a.cn1lib file for the sake of users who are still using Ant as their build tool. For that reason, when you build a library project, the cn1lib is automatically built as well. After running a build, you can look in the common/target directory and find your.cn1lib file ready to be distributed.

Editing Java Code

To get acquainted with your project, add a "Hello World" Java class that you want to make available as part of your cn1lib.

Add a new class inside the "common/src/main/java" directory with package `com.example`, and name `HelloWorld`. Enter the following contents into the class:

```
public class HelloWorld {
    public static void helloWorld() {
        System.out.println("Hello world");
    }
}
```

Now build the library again. (See Section 1.5.1.6, “Building the library”).

Using the library in an application project

Now that you’ve built your library and added a Java class, try adding it as a dependency in an application project. If you haven’t yet created an application project, do that now. See Section 1.2.2, “Creating a new project” for instructions on creating a new application project.

Open the common/pom.xml file of your application project.



Make sure you’re editing the common/pom.xml file of the **application project** and not the library project.

This file may look a little hairy as there is a lot of configuration in there. You will be looking for the `<dependencies>` section.

The common/pom.xml file will have more than one `<dependencies>` tag, as it includes some profiles handling things like Kotlin support. There will be one particular `<dependencies>` tag that includes a comment like

```
<!-- INJECT DEPENDENCIES -->
```

You should add your dependencies before this comment.

For the sake of this example, suppose your library was set up with the following coordinates:

groupId:	`com.example`
artifactId:	`mylib`
version:	`1.0-SNAPSHOT`

In this case you would add the following XML snippet to the `<dependencies>` section of your application’s common/pom.xml file:

```
<dependency>
    <groupId>com.example</groupId>
    <artifactId>mylib-lib</artifactId>
    <version>1.0-SNAPSHOT</version>
    <type>pom</type>
</dependency>
```



Notice that you appended "-lib" to the `artifactId`. This is because you're including the "lib" module of your library project as the dependency, and not the root module. Also the `<type>pom</type>` is important as it indicates that this is a pom dependency - not a regular jar dependency.

Now try it out. Try adding the following code to your application project's main class (or anywhere in the application project, for that matter):

And build the project. The project should build OK, and if you run it, you should see that the `helloWorld()` method works as designed.

1.5.2. Adding menus to the simulator

A cn1lib can contribute menu items to the Codename One simulator's menu bar. This is the same affordance the framework itself uses for the Skins / Native Theme / Simulate menus — opened up so library authors can expose backend-specific actions (for example, "Add a simulated peripheral," "Inject a push notification," or "Switch backend") without users having to write any Swing code or instrument their app.

Every menu item is also reachable from CN1 UnitTests (or any app code) through `CN.execute("namespace:itemN")` — the JavaSE port's URL execute is overloaded to recognize a registered hook url and dispatch it on the EDT instead of opening it as a browser URL. On Android, iOS, JavaScript and other production targets no hooks are registered, so a hook-style URL falls through to the normal native execute and (almost always) becomes a no-op; tests should pair `CN.execute` with `CN.canExecute` for that reason. Hooks can also be declared with no menu label, which makes them callable from tests but invisible in the menu — useful for state-priming actions a human wouldn't click.

The contract

Each cn1lib ships a properties file at a well-known classpath location. The simulator scans every jar on its classpath for this resource and merges the results, so multiple cn1libs coexist cleanly:

```
# META-INF/codenameone/simulator-hooks.properties
name=Bluetooth
namespace=bluetooth   # optional; defaults to slugified `name`

# Each itemN is the action; the matching labelN is the menu text.
# Items are positional — the loader reads item1, item2, item3, ... and
# stops at the first missing index. Don't skip numbers.
item1=com.example.bt.simulator.Hooks#toggleAdapter
```

```
label1=Toggle adapter on/off

item2=com.example.bt.simulator.Hooks#addDemoPeripheral
label2=Add demo peripheral

# Label omitted → API-only hook. Callable from tests, invisible in menu.
item3=com.example.bt.simulator.Hooks#primeReadFailure
```

Required keys:

`name`

: Menu title shown in the simulator's menu bar. One menu per properties file.

`itemN`

: A `fully.qualified.ClassName#staticMethodName` reference for the Nth menu item. The method must be `public static void` and take no arguments. Items are numbered from 1 upward; the loader stops at the first missing `itemN`, so don't leave gaps.

Optional keys:

`namespace`

: Identifier used for the `CN.execute` lookup (for example, `bluetooth` for a URL like `bluetooth:item1`). Defaults to lowercased, ASCII-slugified `name` (`Push Notifications!` → `push-notifications`). Set this explicitly when you want a different identifier from the display name.

`labelN`

: Display text for the matching `itemN`. Omit entirely to make the hook API-only — registered with `CN.execute` but hidden from the menu.

No groups, no submenus, no priority — flat by design. If you need ordering relative to another cn1lib, you can't have it: discovery order wins, and that's intentional so the contract stays small and the future simulator UX can re-render this metadata however it likes.

The action method

The simulator dispatches every action on the Codename One EDT through `Display.callSerially`, so your method can call `Display.getInstance()`, `Form.show()`, `Dialog.show()`, `ToastBar.showInfoMessage()` and any other CN1 API. Reflection uses the same classloader that loaded `Display`, so cn1lib internals (including package-private classes) resolve normally:

The `Display.isInitialized()` guard is a useful pattern when the same static methods are also called from JUnit tests that don't run inside a live CN1 simulator — the state mutation runs in both contexts, only the UI feedback is skipped.

Worked example using cn1-bluetooth

The `cn1-bluetooth` cn1lib ships a JavaSE port with two backends: a scriptable in-memory simulator and a real-hardware backend that talks to a native helper (CoreBluetooth on macOS, BlueZ on Linux, WinRT on Windows). Both are useful in different stages of development, and the menu lets

the user choose between them and exercise the simulator without writing any test scaffolding.

Its `simulator-hooks.properties` looks like this:

```
name=Bluetooth
namespace=bluetooth

item1=com.codename1.bluetoothle.BluetoothSimulatorHooks#toggleAdapter
label1=Toggle adapter on/off

item2=com.codename1.bluetoothle.BluetoothSimulatorHooks#addDemoPeripheral
label2=Add demo peripheral

item3=com.codename1.bluetoothle.BluetoothSimulatorHooks#switchToNativeBle
label3=Switch backend → native BLE (real hardware)

# API-only: used by the test suite but never displayed in the menu
item4=com.codename1.bluetoothle.BluetoothSimulatorHooks#primeReadFailure
```

When the user runs an app that depends on `cn1-bluetooth`, the simulator's menu bar gets a **Bluetooth** menu with the three labeled items. Clicking **Add demo peripheral** drops a peripheral into the in-memory simulator that the running app can then scan for, connect to, and exchange data with — without any real hardware. `item4` is callable from tests via `CN.execute("bluetooth:item4")` but never shows up in the menu.

Calling hooks from CN1 unit tests

CN1 unit tests (`AbstractTest` subclasses run via `mvn cn1:test`) compile under the same restrictions as the rest of the app — no reflection, no JavaSE-only imports. Drive a hook the same way you'd execute any URL — `CN.execute` recognizes registered hook urls and dispatches them on the EDT:

```
public class BluetoothDemoTest extends AbstractTest {
    @Override
    public boolean runTest() throws Exception {
        // Skip cleanly off-simulator: a real device has no hook registered
        // and CN.canExecute will not return TRUE.
        if (!Boolean.TRUE.equals(CN.canExecute("bluetooth:item2"))) {
            return true;
        }
        // Seed the simulator — same effect as clicking "Add demo peripheral".
        CN.execute("bluetooth:item2");
        // nullnow drive the public Bluetooth API as usual.
        return true;
    }
}
```

A common pattern: ship label-bearing hooks for actions a developer might want to fire manually (toggle adapter, inject notification), and ship label-less hooks for test-only state setup

(`primeReadFailure`, `seedFixture`) that would just clutter the menu.

What's intentionally not exposed

- **Swing types.** `JMenu`, `JMenuItem`, `KeyStroke` and friends don't appear in the contract. The simulator UX may change shape (toolbar, command palette, sidebar) and cn1libs shouldn't have to follow.
- **Submenus, separators, priority.** The metadata is a flat list with no hierarchy. If you want grouping, ship multiple `simulator-hooks.properties` files in separate jars — each becomes its own menu.
- **Long-running work on the EDT.** Hook methods run on the CN1 EDT; if you need to do I/O, fire-and-forget a `new Thread(⋯)` from the method or use `CN.invokeAndBlock` so you don't block the UI.

1.5.3. Distributing your library

The recommended way to distribute your library is on Maven central. That way users will be able to install your library by copying and pasting a familiar `<dependency>` snippet into their pom.xml file.

Chapter 2. Introduction

Codename One is a Write Once Run Anywhere mobile development platform for Java/Kotlin developers. It fits into modern Maven-capable IDEs such as IntelliJ IDEA, NetBeans, VS Code, and Eclipse, and you can also drive it entirely from the command line.

Codename One's mission statement is:

> Unify the complex and fragmented task of mobile device programming into a single set of tools, APIs and services. As a result create a more manageable approach to mobile application development without sacrificing the power/control given to developers.

This brings that old "Write Once Run Anywhere" (WORA) Java mantra to mobile devices without dumbing it down to the lowest common denominator.

The things that make Codename One stand out from other tools in this field are:

- Write Once Run Anywhere support with no special hardware requirements and 100% code reuse
- Compiles Java/Kotlin into native code for iOS, Android and even JavaScript/PWA
- Open Source and Free with commercial backing/support
- Easy to use with 100% portable Drag GUI builder
- Full access to underlying native OS capabilities using the native OS programming language (for example: Objective-C) without compromising portability
- Provides full control over every pixel on the screen
- Lets you use native widgets (views) and mix them with Codename One components within the same hierarchy (heavyweight/lightweight mixing)
- Supports seamless Continuous Integration out of the box

Codename One can trace its roots to the open source LWUIT project started at Sun Microsystem in 2007 by Chen Fishbein (co-founder of Codename One). It has been under constant development for over a decade.

2.1. Build cloud

The build cloud approach to mobile development is one of the things that makes Codename One stand out. iOS native development requires a Mac with Xcode. Windows native development requires a Windows machine. To make matters worse, Apple, Google, and Microsoft change their tools.

That makes it hard to keep up.

When you develop an app in Codename One, use the built-in simulator for running and debugging. When you want to build a native app, use the build cloud, where Macs build the native iOS apps

and Windows machines build the native Windows apps. This works seamlessly and makes Codename One apps native because the native platform compiles them. For example, iOS builds use Macs running Xcode, the native Apple tool, to build the app.



Codename One doesn't send source code to the build cloud, compiled bytecode!

2.1.1. Why build servers

The build servers can build native iOS apps without a Mac and native Windows apps without a Windows machine. They remove the need to install and update complex toolchains and simplify the process of building a native app to a right click.

Even though the build servers streamline delivery, Codename One also supports fully local builds. You can install the toolchain on your own hardware and follow the workflows in Chapter 1, *Maven Project Workflow* and Chapter 87, *Working with Codename One sources* to compile, package, and test apps without leaving your desktop environment.

For example, because native iOS applications require a Mac with a recent version of Xcode, Codename One maintains such machines in the cloud. When developers send an iOS build, that Mac generates C source code using ParparVM [https://github.com/codenameone/CodenameOne/tree/master/vm], then compiles the C source code using Xcode and signs the resulting binary using Xcode. You can install the binary on your device or build a distribution binary for the App Store. Since C code is generated, your app is also future-proof against changes from Apple. You can also inject Objective-C native code into the app while keeping it 100% portable thanks to Codename One's native interfaces capability.

Subscribers can receive the C source code back using the include-sources feature of Codename One and use those sources for benchmarking and debugging on devices.

The same is true for most other platforms. On Android and Java SE, Codename One runs the standard Java code, unchanged.

Java Language Level Support

Historically, Java 8 syntax was supported through Retrolambda [https://github.com/orfjackal/retrolambda] on the Codename One servers. It translated bytecode to Java 5 syntax using a server-based processor built on Retroweaver and custom code. This architecture was transparent to developers because the build servers abstract most of the painful differences between devices.

Today Codename One supports Java 25 level bytecode in ParparVM [https://github.com/codenameone/CodenameOne/tree/master/vm] and Android supports Java 17 level bytecode natively. This lets Codename One support Java 17 seamlessly, and newer versions enable Java 25 level support through client side bytecode mutation.

2.1.2. How does Codename One work

Codename One uses a SaaS-based approach so the information in this appendix might (and probably will) change in the future to accommodate improved architectures. This appendix includes this information for reference, you don't need to understand this to follow the content of the book...

Since Android is already based on Java, Codename One is already native to Android and works with the Android VM (ART/Dalvik).

On iOS, Codename One built and open-sourced ParparVM, which is a conservative VM. ParparVM features a concurrent, non-blocking GC and is written entirely in Java/C. ParparVM is a transpiler that generates C source code matching the given Java bytecode. This means that an Xcode project is generated and compiled on the build servers. It's as if you hand-coded a native app and is thus future-proof against changes that Apple introduces. For example, Apple migrated to 64-bit and later introduced bitcode support to iOS. ParparVM needed no modifications to meet those changes.



Codename One translates the bytecode to C, which is faster than Swift/Objective-C. The port code that invokes iOS APIs is hand coded in Objective-C

Codename One earlier offered a UWP (Universal Windows Platform) target based on iKVM. That target was discontinued in release 7.0.229 and is preserved as historical context in older documentation and blog posts.

JavaScript build targets use TeaVM to do the translation statically. TeaVM supports threading using JavaScript by breaking the app down in a rather elaborate way. To support the complex UI Codename One uses the HTML5 Canvas API which allows absolute flexibility for building applications.

For desktop builds Codename One uses `javapackager`, since both Macs and Windows machines are available in the cloud, the platform-specific nature of `javapackager` isn't a problem.

Lightweight architecture

What makes Codename One stand out is the approach it takes to UI: "lightweight architecture."

Lightweight architecture is the "not so secrete sauce" to Codename One's portability. Essentially it means all the components/widgets in Codename One are written in Java. Thus their behavior is consistent across all platforms and they're fully customizable from the developer code as they don't rely on OS internal semantics. This allows developers to preview the application accurately in the simulators and GUI builders.

One of the big accomplishments in Codename One is its unique ability to embed "heavyweight" widgets into place among the "lightweights." This is crucial for apps such as Uber where the cars and widgets on top are implemented as Codename One components yet below them, you have the native map component.

Codename One achieves fast performance by drawing using the native gaming APIs of most platforms for example: OpenGL ES on iOS. The core technologies behind Codename One are all open source including most of the stuff developed by Codename One itself, for example: ParparVM

but also the full library, platform ports, designer tool, device skins etc.

Lightweight Architecture Origin

Lightweight components date back to Smalltalk frameworks, this notion was popularized in the Java world by Swing. Swing was the main source of inspiration to Codename One's predecessor LWUIT. Many frameworks took this approach over the years including JavaFX and most recently Ionic in the JavaScript world.

Why ParparVM

On iOS, Codename One uses ParparVM [https://github.com/codenameone/CodenameOne/tree/master/vm] which translates Java bytecode to C code and boasts a non-blocking GC as well as 64 bit/bitcode support. This VM is fully open source in the Codename One git repository [https://github.com/codenameone/CodenameOne/]. In the past Codename One used XMLVM [http://www.xmlvm.org/] to generate native code similarly, but the XMLVM solution was too generic for the needs of Codename One. ParparVM [https://github.com/codenameone/CodenameOne/tree/master/vm] boasts a unique architecture of translating code to C (similarly to XMLVM), because of that Codename One is the only solution of its kind that can **guarantee** future iOS compatibility since the officially supported iOS toolchain is always used instead of undocumented behaviors.



XMLVM could guarantee that as well, but it's no longer maintained and lacked the API layer support

The key advantages of ParparVM over other approaches are:

- **Truly Native:** since code is translated to C rather than directly to ARM or LLVM code, the app is "more native." It uses the official tools and approaches from Apple and can gain from their advancements. For example: bitcode changes or profiling toolchain.
- **Smaller Class Library:** ParparVM includes a small segment of the full JavaAPI's resulting in final binaries that are smaller than the alternatives by orders of size. This maps directly to performance and memory overhead.
- **Simple and Extensible:** to work with ParparVM you need a basic understanding of C. Unlike other tools that require deep understanding of ARM assembly and LLVM bitcode. This is crucial for the fast-moving world of mobile development, as Apple changes things left and right, you need a more agile VM.

Historical windows ports

Codename One experimented with many Windows ports over the years, including the legacy Windows Phone target and a later UWP (Universal Windows Platform) target based on iKVM.



The UWP target was discontinued in release 7.0.229 and is no longer part of the supported build targets. Older references to it in historical material are retained for archival context.

JavaScript port

The JavaScript port of Codename One uses ParparVM to translate Java bytecode into JavaScript. Cloud builds retain the original TeaVM-based builder [http://teavm.org:] as a compatibility fallback, selected with the public `javascript.port=teavm` build hint.

The JavaScript port allows unmodified Codename One applications to run within a desktop or mobile browser. The port itself is based on the HTML5 Canvas API, which provides a pixel-perfect implementation of the Codename One API.



The JavaScript port is available to all Codename One users.

Desktop and Android

The other ports of Codename One use the VMs available on the host machines and environments to execute the runtime. Historically, Retrolambda [https://github.com/orfjackal/retrolambda] was used to provide Java 8 language features portably.

The Android port uses the native Android tools, including the Gradle build environment, in the latest versions.

The desktop port creates a standard Java SE application, which is packaged with the JRE and an installer.



The Desktop port is available to pro grade subscribers of Codename One

2.2. History



Figure 1. LWUIT App Screenshot circa 2007

Codename One was started by Chen Fishbein and Shai Almog who authored the Open Source LWUIT project at Sun Microsystems (circa 2007). The LWUIT project aimed to solve the fragmentation within J2ME/Blackberry devices by creating a higher standard of user interface than the common baseline at the time. LWUIT received critical acclaim and traction within many industries but was limited by the declining feature phone market. It was forked by many companies including Nokia. It was used as the base standard for DTV in Brazil. Another fork has brought a LWUIT into high-end cars from Toyota and other companies. This fork later adapted Codename One as well.

In 2012 Shai and Chen formed Codename One as they left Oracle. The project has taken many of the basic concepts developed within the LWUIT project and adapted them to the smartphone world which is still experiencing similar issues to the device fragmentation of the old J2ME phones.

2.3. Core concepts of mobile programming

Before you proceed, this section explains some universal core concepts of mobile programming that might not be intuitive. These are universal concepts that apply to mobile programming regardless of the tools you're using.

You can skip this section if you feel you're familiar enough with the core problems/issues in mobile app development.

2.3.1. Density

Density is also known as DPI (Dots Per Inch) or PPI (pixels or points per inch). Density is confusing, unintuitive and might collide with common sense. For example, an iPhone 7 plus has a resolution of `1080x1920` pixels and a PPI of `401` for a 5-inch screen. But, an iPad 4 has `1536x2048` pixels with a PPI of `264` on a `9.7` inch screen... Smaller devices can have higher resolutions!

As the following figure shows, if a Pixel 2 XL had pixels the size of an iPad, it would have been twice the size of that iPad. While in reality, it's half the height of the iPad!

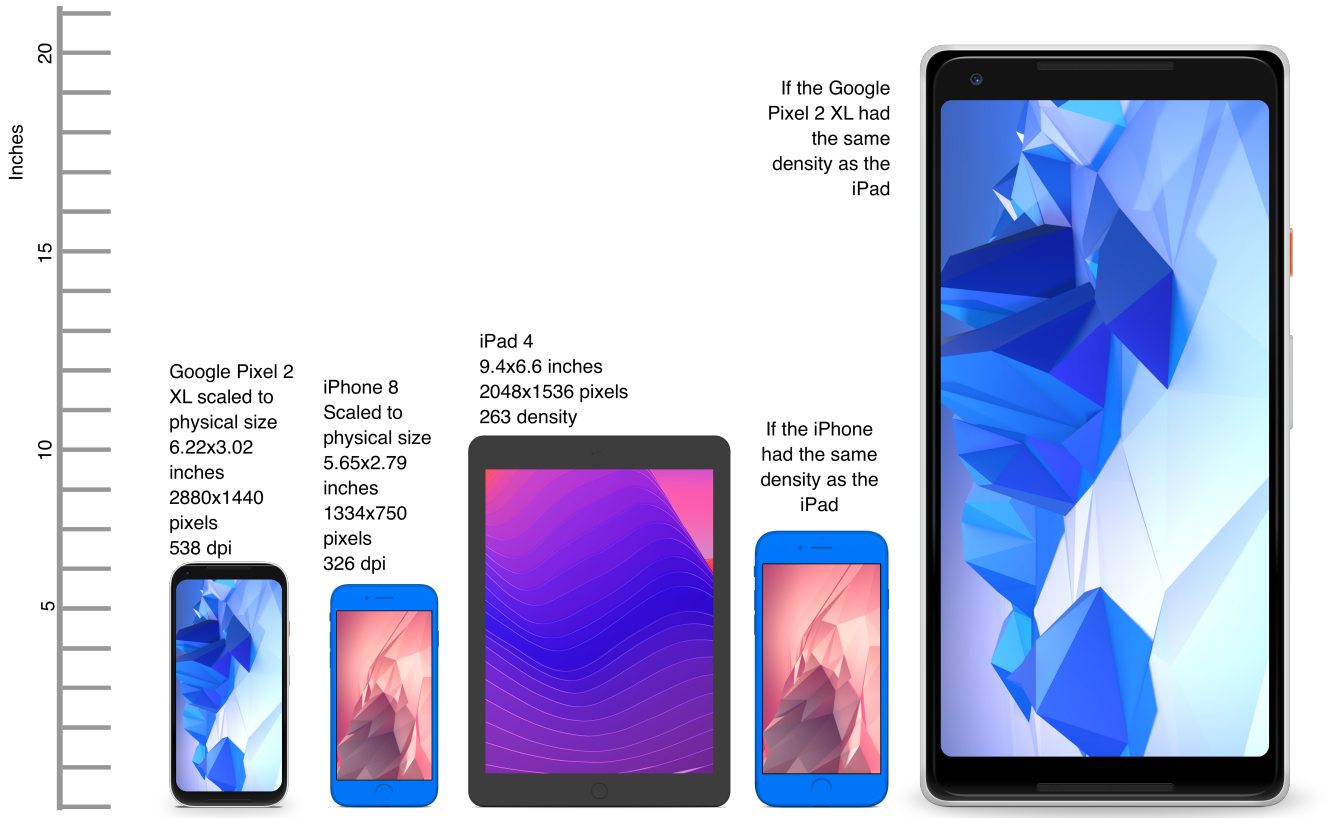


Figure 2. Device Density vs. Resolution

Differences in density can be extreme. A second generation iPad has 132 PPI, where modern phones have PPI that's over 600 dpi.

Low resolution images on high PPI devices will look either small or pixelated. High resolution images on low PPI devices will look huge, overscaled (artifacts) and will consume too much memory.

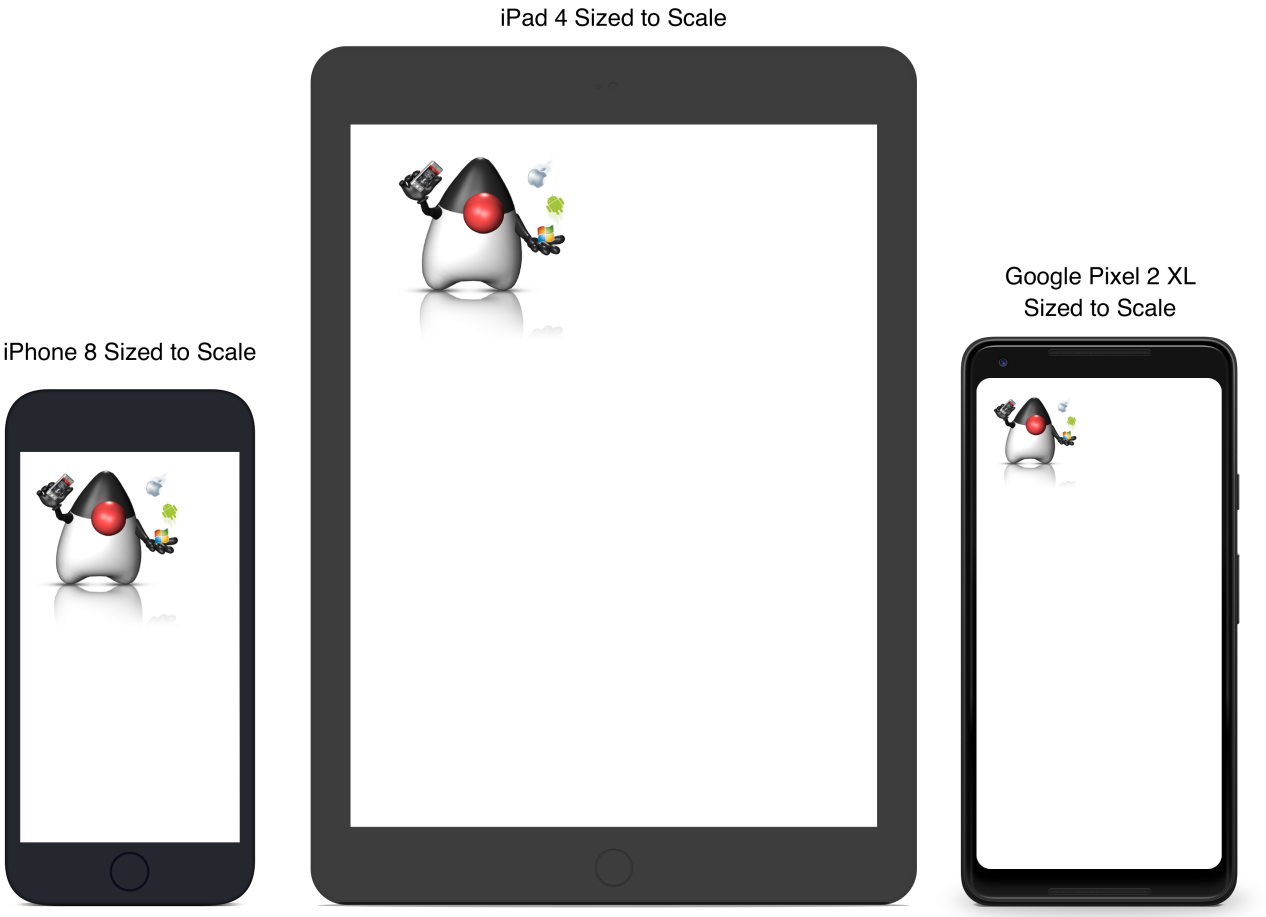


Figure 3. How the Same Image Looks in Different Devices

The exact same image will look different on each device, sometimes to a comical effect. One of the solutions for this problem is multi-images. All OS's support the ability to define different images for various densities. Chapter 2 discusses multi-images later in detail.

This also highlights the need for working with measurements other than pixels. Codename One supports millimeters (or dips) as a unit of measurement. This is highly convenient and is a better representation of size when dealing with mobile devices.

However, there is a bigger conceptual issue involved. You need to build a UI that adapts to the wide differences in form factors. You might have fewer pixels on an iPad, but because of its physical size, you would expect the app to cram more information into that space so the app won't feel like a blown-up phone application. Many strategies to address that, but one of the first steps is in the layout managers.

Chapter 2 discusses the layout managers in depth, but the core concept is that they decide where a UI element is placed based on generic logic. That way, the user interface can adapt automatically to the huge variance in display size and density.

2.3.2. Touch interface

The fact that mobile devices use a touch interface today isn't news... However, the implications of that aren't obvious to some developers.

UI elements need to be finger sized and spaced. Otherwise, you risk the "fat finger" effect. That

means spacing should be in millimeters and not in pixels due to device density.

Scrolling poses another challenge in touch-based interfaces. In desktop applications, it's common to nest scrollable items. For example, in touch interfaces, the scrolling gesture doesn't allow such a nuance. Furthermore, scrolling on both the horizontal and vertical axis (side scrolling) can be inconvenient in touch-based interfaces.

2.3.3. Device fragmentation

Some developers single out this wide range of resolutions and densities as "device fragmentation." While it does contribute to development complexity, it isn't a challenging problem to overcome.

Densities aren't the cause of device fragmentation. Device fragmentation is caused by many OS versions with different behaviors. This is clear on Android and relates to the slow rollout of Android vendor versions compared to Google rollout. For example, 7 months after the Android 8 (Oreo) release in 2018, it was still available on 1.1% of the devices. The damning statistic is that 12% of the devices in mid 2018 run Android 4.4 Kitkat released in 2013!

This makes QA difficult as the disparity between these versions is pretty big. These numbers will be out of date by the time you read this, but the core problem remains. It's hard to get all device manufacturers aligned, so this problem will probably remain in the foreseeable future despite everything.

2.3.4. Performance

Besides the obvious need for performance and smooth animation within a mobile app, there are a couple of performance related issues that might not be intuitive to new developers: size and power.

App size

Apps are installed and managed through stores. This poses some restrictions about what an app can do. However, it also creates a huge opportunity. Stores manage automatic update and to some degree the marketing/monetization of the app.

A good mobile app is updated once a month and sometimes even once a week. Since the app downloads automatically from the store, this can be a huge benefit:

- Existing users are reminded of the app and get new features instantly
- New users notice the app featured on a "what's new" list

If an app is big, it might not update over a cellular network connection. Google and Apple have restrictions on automatic updates over cellular networks to preserve battery life and data plans. A large app might negatively impact users perception of the app and trigger uninstalls for example, when a phone is low on available space.

Power drain

Desktop developers think about power usage within their apps. In mobile development, this is a crucial concept. Modern device OSes have tools that highlight misbehaving applications, and this can lead to bad reviews.

Code that loops forever while waiting for input will block the CPU from sleeping and drain the battery.

Worse. Mobile OSes kill applications that drain the battery. If the app is draining the battery and is minimized, (for example, during an incoming call) the app could be killed. This will impact app performance and usability.

2.3.5. Sandbox and permissions

Apps installed on the device are "sandboxed" to a specific area so they won't harm the device or its functionality. The filesystem of mobile applications is restricted so one application can't access the files of another application. Things that most developers take for granted on the desktop such as a "file picker" or accessing the image folder don't work on devices!

This means that when your application works on a file, it belongs to your application. To share the file with a different application, you need to ask the operating system to do that for you.

Furthermore, some features require a "permission" prompt and sometimes require special flags in system files. Apps need to request permission to use sensitive capabilities, for example, Camera, Contacts etc.
Historically, Android developers declared required permissions for an app and the user was prompted with permissions during installation. Android 6 adopted the approach used by iOS of prompting the user for permission when accessing a feature.

This means that in runtime a user might revoke a permission. A good example for an Uber app is the location permission. If a user revokes that permission, the app might lose its location.

2.4. Installing Codename One



Codename One supports JDK 11 through 25 for running the simulator and the desktop ("Run as desktop app") target. Eclipse Temurin (https://adoptium.net) is the easiest way to install a supported JDK.

Codename One projects are built with Maven. Typical Maven targets such as `package`, `clean` and `install` work out of the box, but the Codename One integrations that ship with each IDE provide dedicated Run and Build actions for a smoother workflow.

To create a new Codename One project visit https://www.codenameone.com/initializr/ and generate a starter project, or run the Codename One Application Project Archetype (`cn1app-archetype`) directly on the command line:

```
mvn archetype:generate \
  -DarchetypeGroupId=com.codenameone \
  -DarchetypeArtifactId=cn1app-archetype \
  -DarchetypeVersion=LATEST \
  -DgroupId=YOUR_GROUP_ID \
  -DartifactId=YOUR_ARTIFACT_ID \
  -Dversion=1.0-SNAPSHOT \
```

```
  -DmainName=YOUR_MAIN_NAME \
  -DinteractiveMode=false
```

This command generates a project in the current directory. The folder name matches the `artifactId` value. For example, specifying `-DartifactId=myapp` produces a project inside a new `myapp` directory.

Import the generated Maven project into your preferred IDE and use the Codename One Run in Simulator task from the IDE toolbar or Run/Debug buttons:

- **IntelliJ IDEA:** use **File** > **Open** on the project directory, then choose the Codename One Run in Simulator action from the toolbar or standard Run/Debug controls.
- **NetBeans:** use **File** > **Open Project**, select the generated Maven project, and rely on the Codename One toolbar actions to run and debug the simulator.
- **VS Code:** install the Java and Codename One extensions, open the folder, and trigger the Run in Simulator task from the command palette or the Run/Debug buttons.
- **Eclipse:** use **File** > **Import** > **Existing Maven Projects**, then use the Codename One launch shortcuts provided by the plugin for simulator and build tasks.
- **Command line:** invoke Maven goals directly whenever you need to integrate with CI/CD pipelines or scripting.

For deeper coverage of the Maven goals, project structure, and automation tasks, continue with Chapter 1, *Maven Project Workflow*.



Arbitrary Maven dependencies probably won't work for Codename One. Many dependencies assume a full JDK which Codename One can't provide and they often assume functionality that might not be available for example: reflection, Spring, etc.

Legacy onboarding resources

Legacy Ant-based project instructions remain available for teams maintaining older codebases. New projects should follow the Maven workflows described in this guide.

2.4.1. Important notes for new projects

Before you get to the code there are few important things you need to understand about Codename One applications.

- **App Name** - This is the name of the app and the main class, it's important to get this right as it's hard to change this value later
- **Package Name** - it's **crucial** you get this value right. Besides the difficulty of changing this after the fact, once an app is submitted to iTunes/Google Play with a specific package name this can't be changed! See the sidebar `Picking a Package Name`.
- **Theme** - Codename One ships with various built-in themes; for simplicity recommend `Native` as it's a minimal starting point

Picking a Package Name

Apple, Google and Microsoft identify applications based on their package names. If you use a domain that you don't own it's possible that someone else will use that domain and collide with you. In fact some developers left the default `com.mycompany` domain in place all the way into production sometimes.

This can cause difficulties when submitting to Apple, Google, or Microsoft. Submitting to one of them is no guarantee of success when submitting to another.

To come up with the right package name use a reverse domain notation. If your website is `goodstuff.co.uk` your package name should start with `uk.co.goodstuff`. Follow these guidelines for package names:

- **Lower Case** - some OSes are case-sensitive and handling a mistake in case is painful. The Java convention is lower case; stick to that
- **Avoid Dash and Underscore** - You can't use a dash character (-) for a package name in Java. Underscore (_) doesn't work for iOS. If you want more than one word just use a deeper package e.g.: `com.mydomain.deeper.meaningful.name`
- **Obey Java Rules** - A package name can't start with a number so you can't use `com.mydomain.1sler`. You should avoid using Java keywords like `this`, `if` etc.
- **Avoid Top Level** - instead of using `uk.co.goodstuff` use `uk.co.goodstuff.myapp`. That would allow you to have more than one app on a domain

2.4.2. Runtime

Once Maven is set up you can run the `HelloWorld` application by selecting the Codename One Run in Simulator task from the IDE run menu. The Codename One simulator launches and you can use its menus to control and inspect details related to the device. You can rotate it, determine its location in the world, monitor networking calls etc.

With the `Skins` menu you can download device skins to see how your app will look on different devices.



Some skins are bigger than the screen size, uncheck the `Scrollable` flag in the `Simulator` menu to handle them more effectively

Use your IDEs Debug button with the Run in Simulator task to launch the simulator under the debugger.

Simulator vs. Emulator

Codename One ships with a simulator similarly to the iOS toolchain which also has a simulator. Android ships with an emulator. Emulators go further. They create a virtual machine that's compatible with the device CPU and then boot the full mobile OS within that environment. This provides a proper runtime environment but is **slow**.

Simulators rely on the fact that OSes are similar and so they leave the low-level details in place and just map the API behavior. Since Codename One relies on Java it can start simulating on top of the virtual machine on the desktop. That provides several advantages including fast development cycles and full support for all the development tools/debuggers you can use on the desktop.

Emulators make sense for developers who want to build OS level services for example, screensavers or low-level services. Standard applications are better served by simulators.

2.4.3. The source code of the hello world app

After clicking finish in the new project wizard you have a `HelloWorld` project with a few default settings. The following sections break the class down to small pieces and explain each piece starting with the enclosing class:

Listing 1. HelloWorld Class

```
public class HelloWorld { ①
    private Form current; ②
    private Resources theme; ③

    // null class methods null
}
```

① This is the main class, it's the entry point to the app, notice it doesn't have a `main` method but rather callback which this guide covers soon

② Forms are the "top level" UI element in Codename One. Only one `Form` is shown at a time and everything you see on the screen is a child of that `Form`

③ Every app has a theme, it determines how everything within the application looks for example: colors, fonts etc.

Next consider the first lifecycle method `init(Object)`. The Application Lifecycle Sidebar discusses the lifecycle in depth:

`init(Object)` works as a constructor to some degree. Recommend avoiding the constructor for the main class and placing logic in the init method instead. This isn't crucial but recommend it since the constructor might happen too early in the application lifecycle.

In a cold start `init(Object)` is invoked followed by the `start()` method. For example, `start()` can be invoked more than once if an app is minimized and restored, see the sidebar Application Lifecycle:



Figure 4. Title and Label in the UI

Some complex ideas appear within this short snippet that this chapter addresses later when talking about layout. The gist of it's that you create and show a `Form`. `Form` is the top level UI element, it takes over the whole screen. You can add UI elements to that `Form` object, in this case the `Label`. You use the `BoxLayout` to arrange the elements within the `Form` from top to the bottom vertically.

Application Lifecycle

A few years ago Romain Guy (a senior Google Android engineer) was on stage at the Google IO conference. He asked for a show of hands of people who understand the `Activity` lifecycle (`Activity` is like a Codename One main class). He then proceeded to jokingly call the audience members who lifted their hands "liars" claiming that after all his years in Google he still doesn't understand it...

Lifecycle seems simple on the surface but hides a lot of nuance. Android's lifecycle is ridiculously complex. Codename One tries to simplify this and also make it portable. Sometimes complexity leaks out and the nuances can be difficult to deal with.

Explained an application has three states:

- **Foreground** - it's running and in the foreground which means the user can physically interact with the app
- **Suspended** - the app isn't in the foreground, it's either paused or has a background process running
- **Not Running** - the app was never launched, was killed or crashed

The lifecycle is the process of transitioning between these 3 states and the callbacks invoked when such a transition occurs. The first time you launch the app you start from a "Cold Start" (Not Running State) but on later launches the app is started from the "Warm Start" (Suspended State).

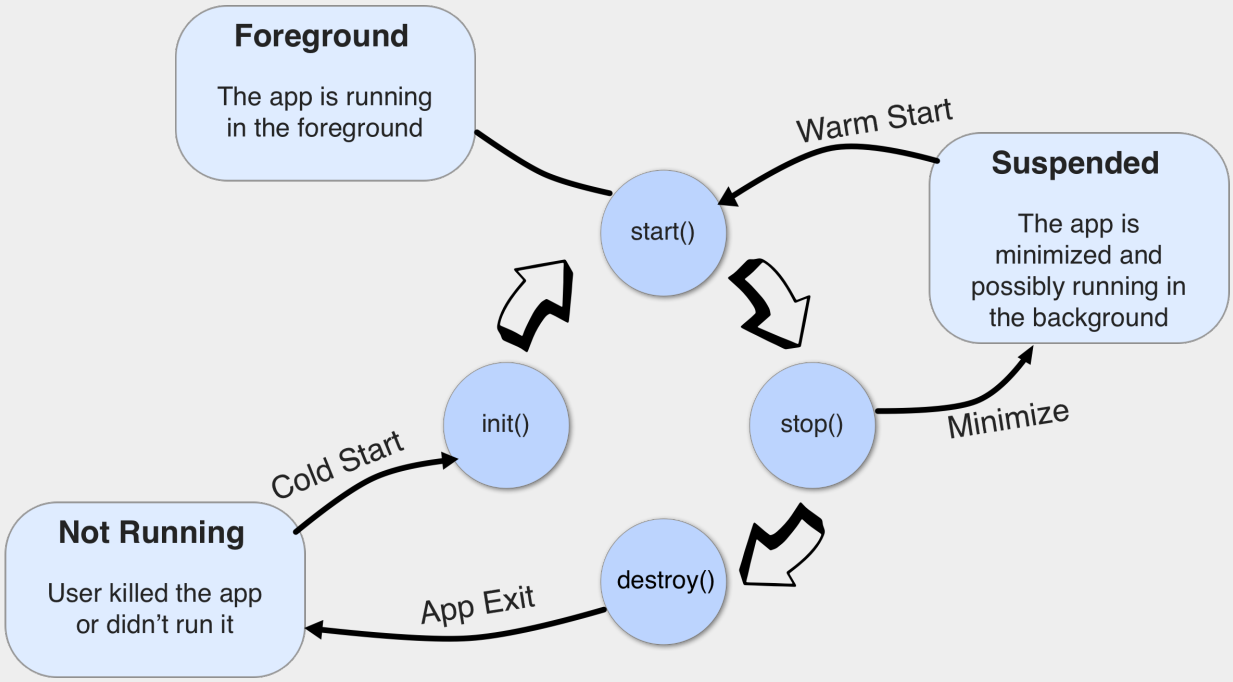


Figure 5. Codename One Application Lifecycle

Codename One has four standard callback methods in the lifecycle API:

- `init(Object)` - is invoked when the app is first launched from a *Not Running* state.
- `start()` - is invoked for two separate cases. After `start()` is finished the app transitions to the *Foreground* state.
  - Following `init(Object)` in case of a cold start. Cold start refers to starting the app from a *Not Running* state.
  - When the app is restored from *Suspended* state. In this case `init(Object)` isn't invoked
- `stop()` - is invoked when the app is minimized for example, when switching to a different app. After `stop()` is finished the app transitions to the *Suspended* state.
- `destroy()` - is invoked when the app is destroyed for example, killed by a user in the task manager. After `destroy()` is finished the app is no longer running hence it's in the *Not Running* state.



`destroy()` is optional there is no guarantee that it will be invoked. It should be used only as a last resort

Now that you have a general sense of the lifecycle lets look at the last two lifecycle methods:

That's it. You should now have a general sense of the code. It's time to run on the device.

2.4.4. Building and deploying on devices

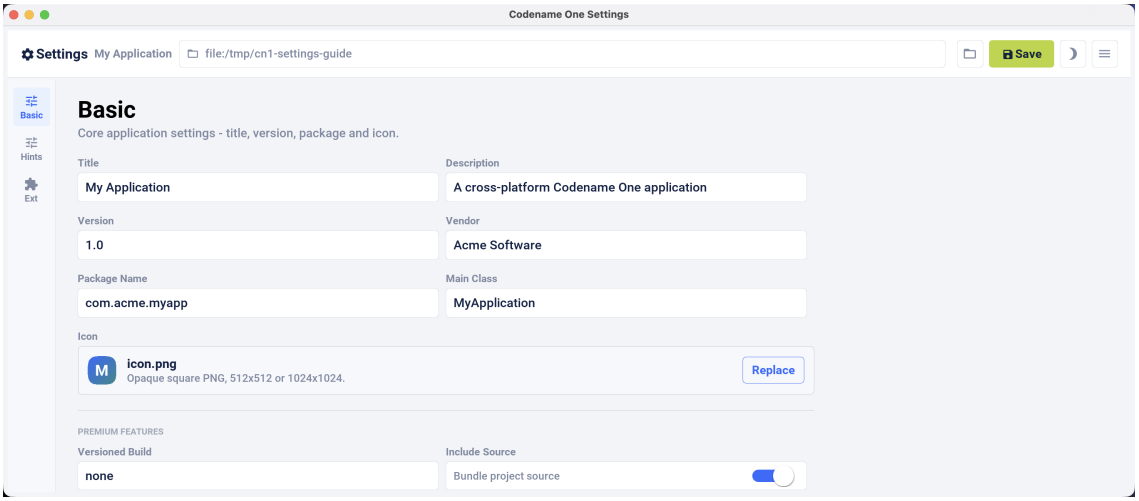


Figure 6. Codename One Settings

Use the standalone Codename One Settings editor to configure the application title, version, package, icon, build hints, and extensions. Launch it from the Maven project root:

```
mvn cn1:settings
```

Use the standalone Certificate Wizard for signing assets. Monitor Codename One Cloud builds and subscription information on the Codename One website.

Signing/Certificates

All the modern mobile platforms require signed applications but they all take radically different approaches when implementing it.

Signing is a process that marks your final application for the device with a special value. This value (signature) is a value that you can generate based on the content of the application and your certificate. Effectively it guarantees the app came from you. This blocks a 3rd party from signing their apps and posing as you to the App Store or to the user. It's a crucial security layer.

A certificate is the tool you use for signing. Think of it as a mathematical rubber stamp that generates a different value each time. Unlike a rubber stamp a signature can't be forged!

Signing on Android



Backup your Android certificate and save its password!

If you lose your Android certificate you won't be able to update your app

Android uses a self-signed certificate approach. You can generate a certificate by describing who you're and picking a password!

Anyone can do that. For example, once a certificate is published it can't be replaced...

If this wasn't the case someone else could push an "upgrade" to your app. Once an app is submitted with a certificate to Google Play this app can't be updated with any other certificate.

With that in mind generating an Android certificate is trivial.



The following chart illustrates a process that's identical on all IDEs

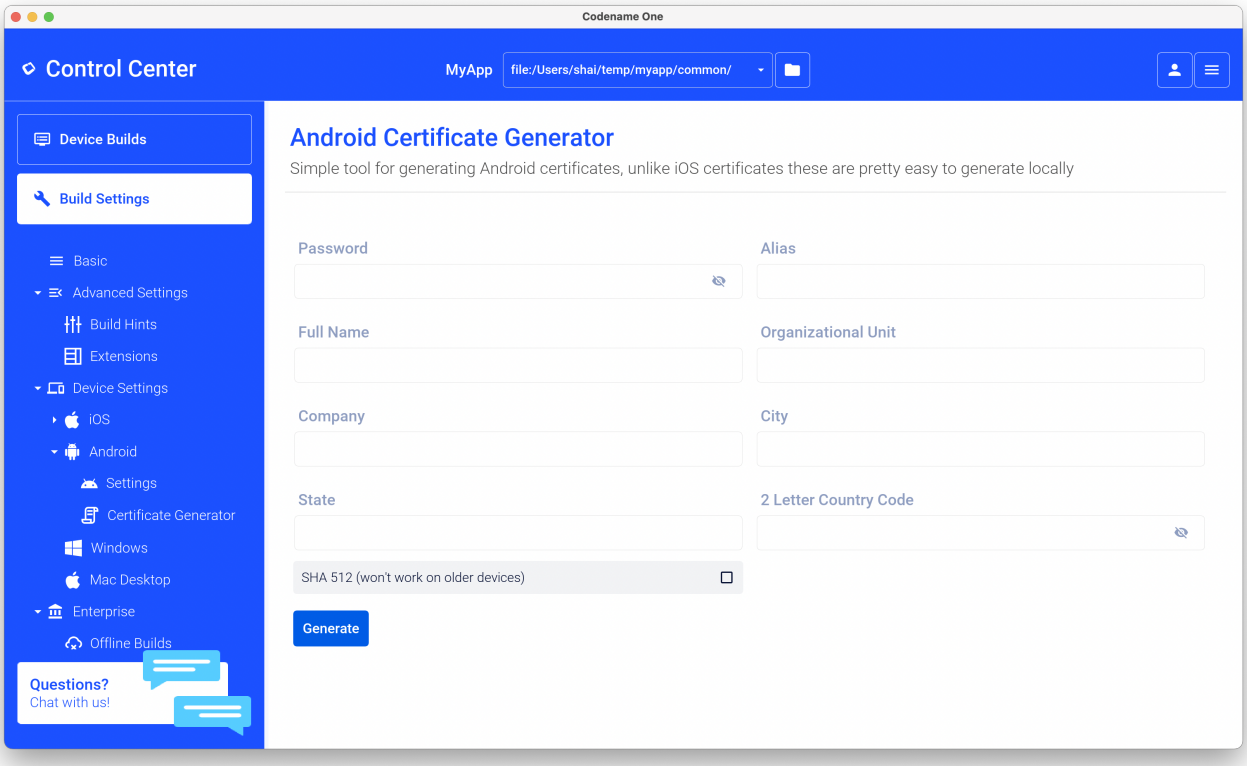


Figure 7. Process of Certificate Generation for Android



Your certificate will generate into the file `Keychain.ks` *in your home directory*

Make sure to back that up and the password as losing these can have dire consequences

Should You Use a Different Certificate for Each App?

In theory yes. In practice it's a pain... Keeping multiple certificates and managing them is a pain so you often just use one.

The drawback of this approach occurs when you're building an app for someone else or want to sell the app. Giving away your certificate is akin to giving away your house keys. It makes sense to have separate certificates for each app.

Signing and provisioning iOS

Code signing for iOS relies on Apple as the certificate authority. This is something that doesn't exist on Android. iOS also requires provisioning as part of the certificate process and separates the process for development/release.

First, the good news:

- Losing an iOS certificate is no big deal - in fact you revoke them often with no impact on shipping apps

- Codename One has a wizard that hides most of the pain related to iOS signing

In iOS Apple issues the certificates for your applications. That way the certificate is trusted by Apple and is assigned to your Apple iOS developer account. One important caveat applies: You need an iOS Developer Account and Apple charges a 99USD Annual fee for that.



The 99USD price and need have been around since the introduction of the iOS developer program for 10 years at the time of this writing. It might change at some point though

Apple also requires a "provisioning profile" which is a special file bound to your certificate and app. This file describes some details about the app to the iOS installation process. One of the details it includes during development is the list of permitted devices.

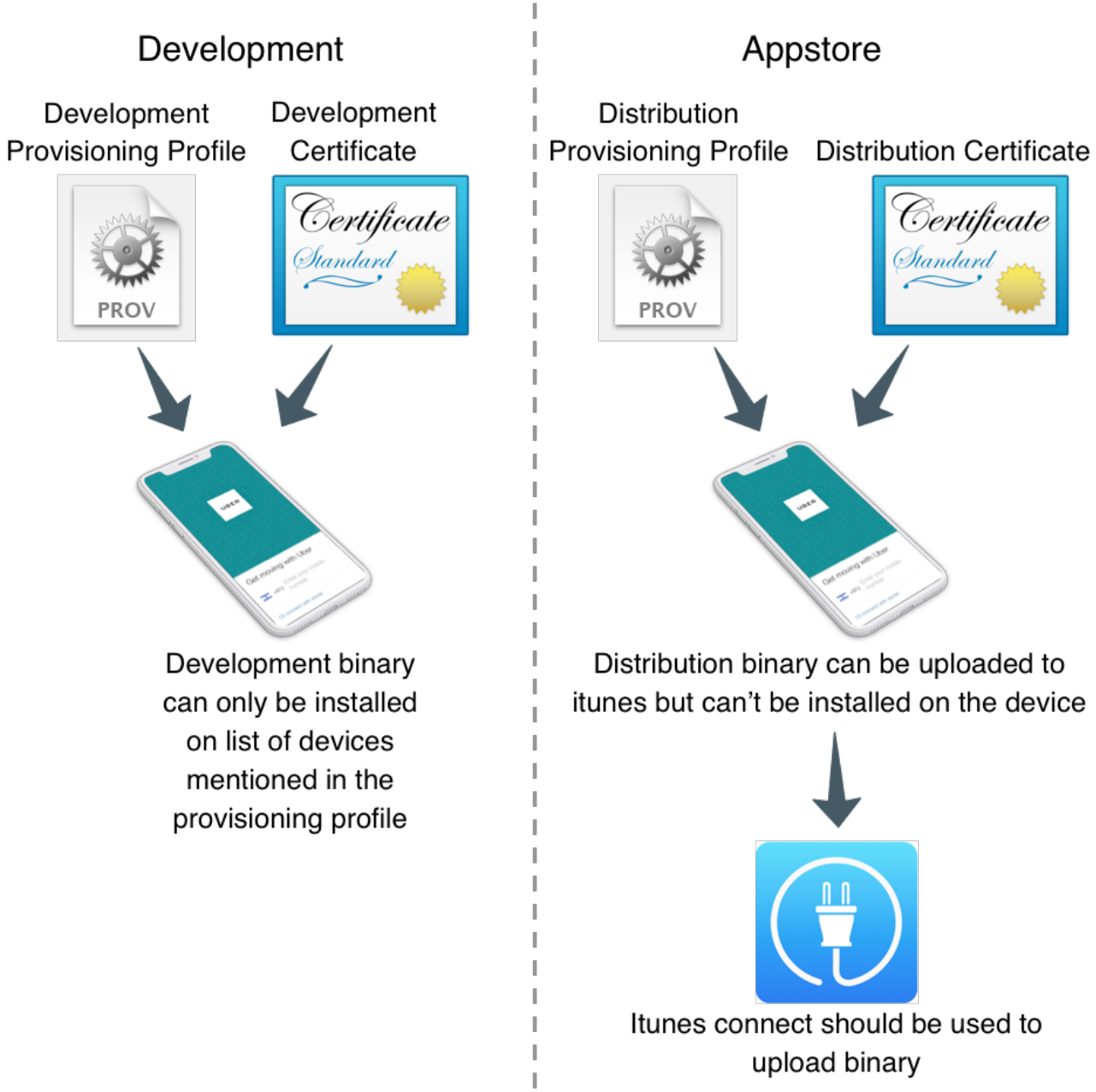


Figure 8. The Four Files Required for iOS Signing and Provisioning

You need 4 files for signing. Two certificates and two provisioning profiles:

1. **Production:** The production certificate/provisioning pair is used for builds that are uploaded to iTunes
2. **Development:** The development certificate/provisioning is used to install on your development devices

The standalone Certificate Wizard can automatically create these files and configure the current Maven project.

Run the wizard from the project directory:

```
mvn cn1:certificatewizard
```

After storing an App Store Connect API key, click **Auto Setup** in the toolbar. The wizard reads the project package and display name, creates or reuses the Bundle ID with push enabled, creates/reuses development and App Store signing assets, and writes the debug/release signing settings back into `codenameone_settings.properties`.

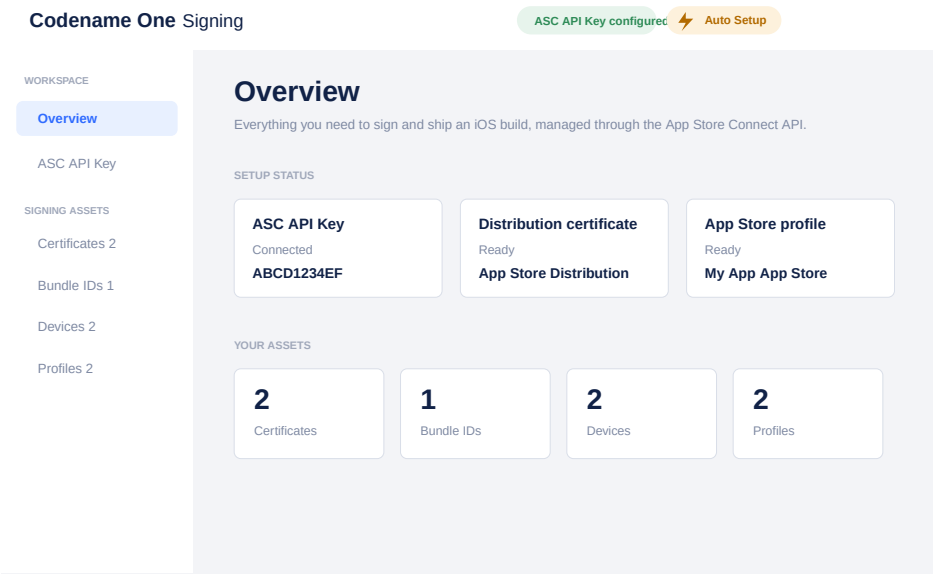


Figure 9. Certificate wizard overview with mocked signing assets

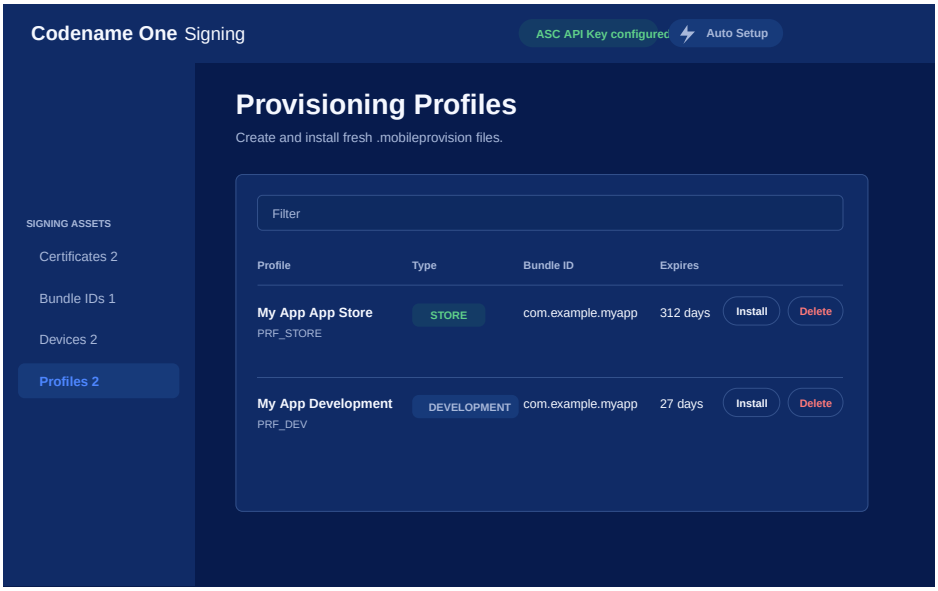


Figure 10. Certificate wizard profiles screen with mocked values



If you have more than one project you should use the same iOS P12 certificate files in all the projects and regenerate the provisioning. In this situation the Certificate Wizard asks you if you want to revoke the existing certificate which you shouldn't revoke in such a case. You can update the provisioning profile in Apple's iOS developer website.

One important aspect of provisioning on iOS is the device list in the provisioning step. Apple allows you to install the app on 100 devices during development. This blocks developers from skipping the App Store altogether. It's important you list the correct UDID for the device in the list otherwise install will fail.



Many apps and tools offer the UDID of the device, but they aren't necessarily reliable and might give a fake number!

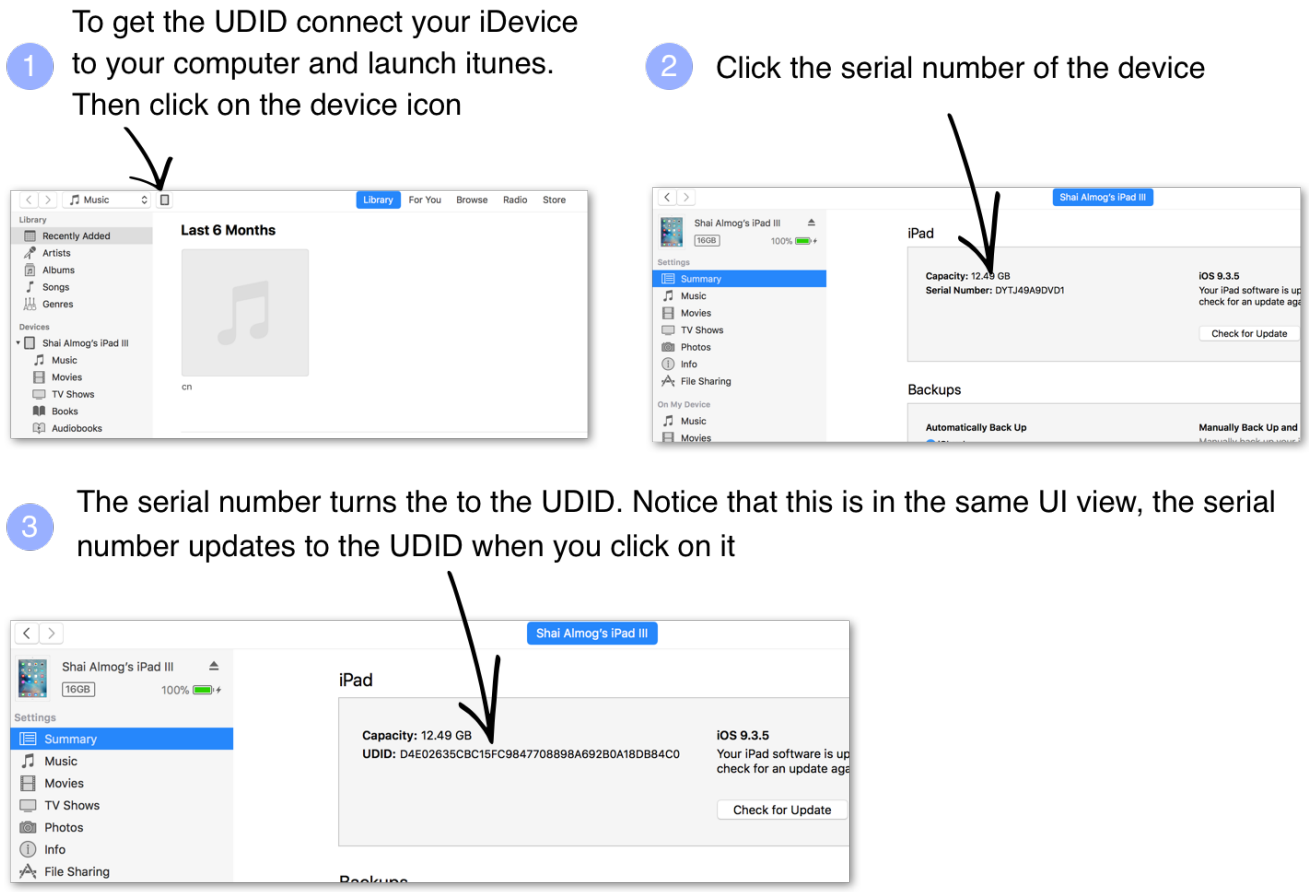


Figure 11. Get the UDID of a Device



You can right-click the UDID and select `copy` to copy it

The simplest and most reliable process for getting a UDID is through iTunes. Other approaches have worked in the past but this approach is guaranteed.



Ad hoc provisioning allows 1000 beta testers for your application but it's a more complex process that you won't discuss here although it's supported by Codename One

Build and install

Before you continue with the build you should sign up at https://www.codenameone.com/build-server.html where you can soon follow the progress of your builds. You need a Codename One account to build for the device.

Now that you have certificates, the process of device builds is a right click away for both OSes. You can right-click the project and select `Codename One` → `Send iOS Debug Build` or `Codename One` → `Send Android Build`.

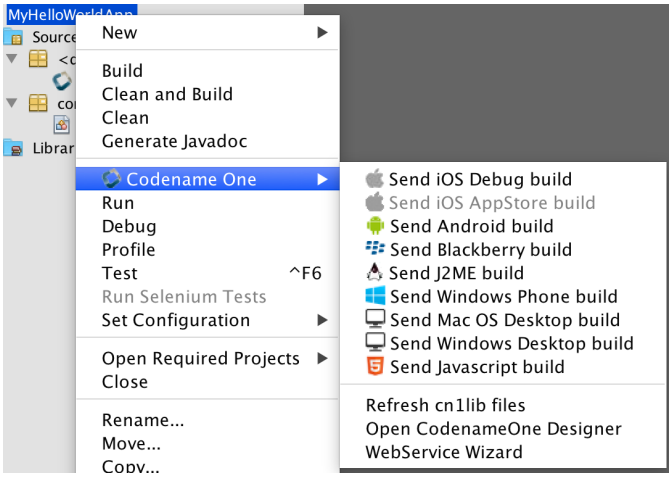


Figure 12. Right click menu options for sending device builds



The first time you send a build you will be prompted for the email and password you provided when signing up for Codename One

Once you send a build you should see the results in the build server page:

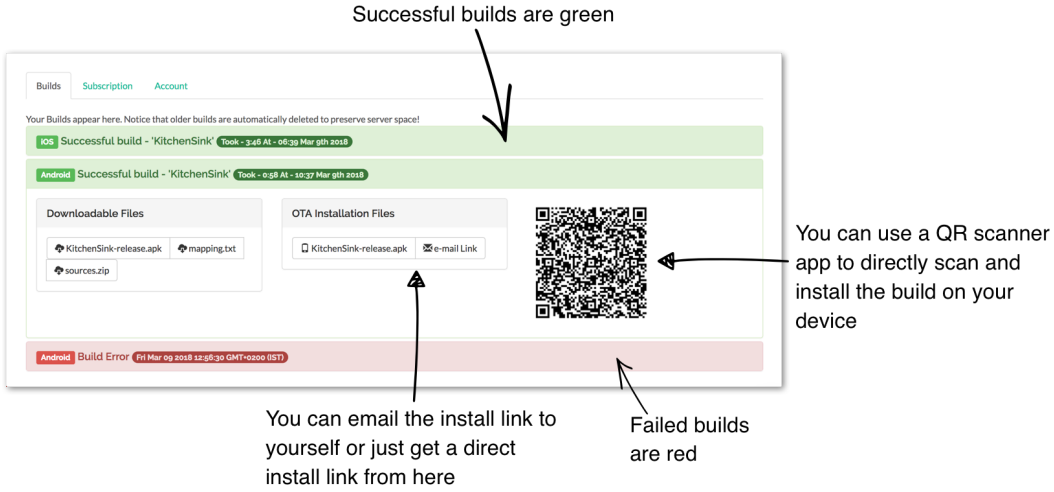


Figure 13. Build Results



On iOS make sure you use Safari when installing, as 3rd party browsers might have issues

Once you go through those steps you should have the `HelloWorld` app running on your device. This process is non-trivial when starting so if you run into difficulties don't despair and seek help at the discussion forum (https://www.codenameone.com/discussion-forum.html) or stack overflow (https://stackoverflow/tags/codenameone/). Once you go through signing and installation, it becomes easier.



You can also install the application either by emailing the installation link to your account (using the `e-mail Link` button)

You can also download the binaries to upload them to the appstores.

2.5. Kotlin

Codename One started before Kotlin became public. Kotlin has since shown itself as an interesting option for developers within the Android community. With that in mind, Codename One added support for Kotlin.

To use Kotlin with Codename One you can create a `kotlin` directory next to the `java` directory under the `common/src/main` directory. Kotlin code that resides there can work as usual and interact with the Java code.

Please notice the following:

- Don't use the project conversion tools or accept the warning that the project isn't a Kotlin project. You do your own build process
- Warnings and errors aren't listed and builds that claim to have errors might pass

2.5.1. Hello Kotlin

Due to the way Kotlin works you can create a regular Java project and convert sources to Kotlin. You can mix Java and Kotlin code without a problem and Codename One would "work."

The hello world Java source file looks like this (removed some comments and whitespace):

When you select that file and select the menu option Code → Convert Java file to Kotlin File you should get a familiar structure. The problem is that there are two bugs in the automatic conversion… That's the code for Kotlin behaves differently from standard Java.

The first problem is that Kotlin classes are final unless declared otherwise so you need to add the open keyword before the class declaration.

This is essential as the build server will fail with weird errors related to instanceof.



This applies to the main class of the project, other classes in Codename One can remain `final`

The second problem is that arguments are non-null by default. The `init` method might have a null argument. This fails with an exception. The solution is to add a question mark to the end of the call: `fun init(context: Any?)`.

Once all that's in place Kotlin should work. This should be possible for more JVM languages in the future.

Chapter 3. Basics: Themes, styles, components, and layouts

Start with a brief overview of the core ideas in Codename One. This chapter revisits them in more detail as it goes on.

3.1. Components

Every button, label, or other element on the screen in a Codename One application is a Component [https://www.codenameone.com/javadoc/com/codename1/ui/Component.html]. This is a simplified version of the class hierarchy:

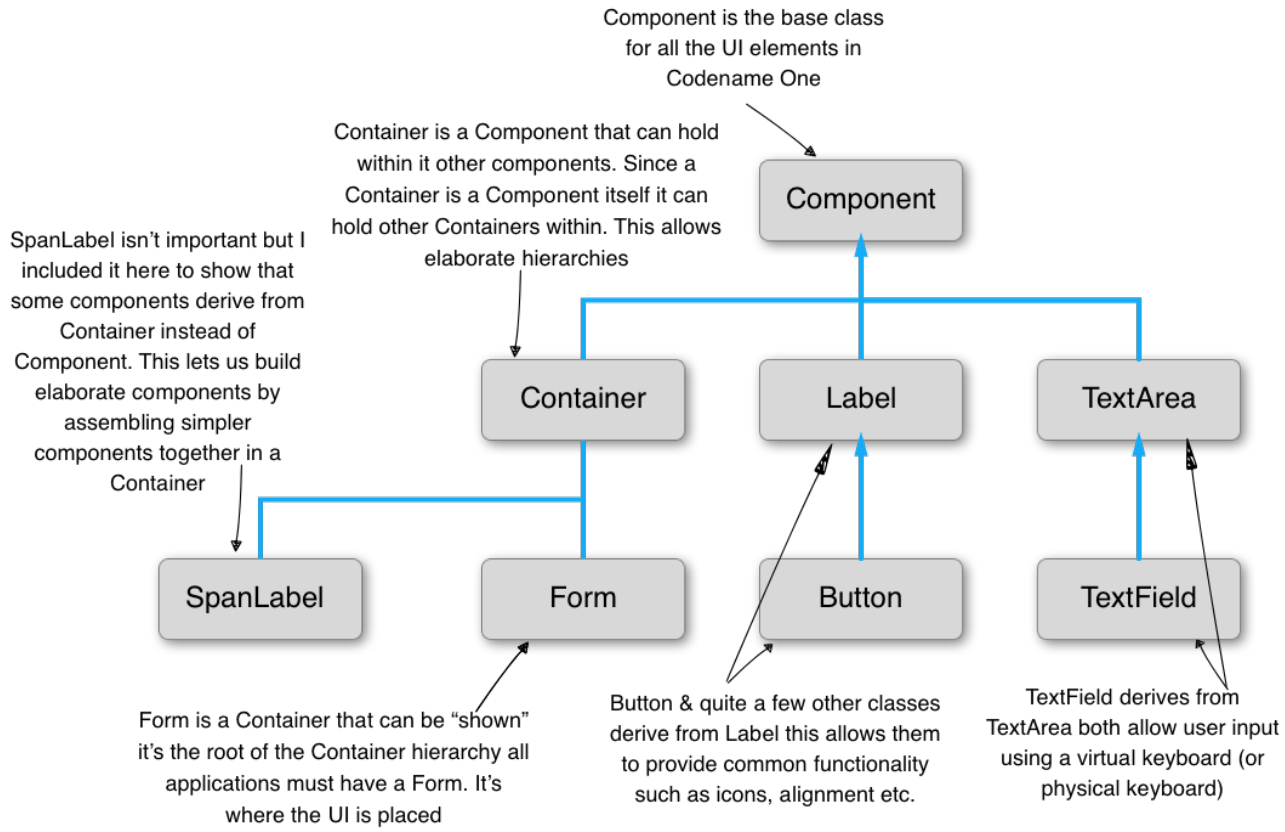


Figure 14. The Core Component Class Hierarchy

The Form [https://www.codenameone.com/javadoc/com/codename1/ui/Form.html] is a special `Component`. It's the root component shown to the user. `Container` is a `Component` type that can hold other components. That lets you create elaborate hierarchies by nesting `Container` instances.

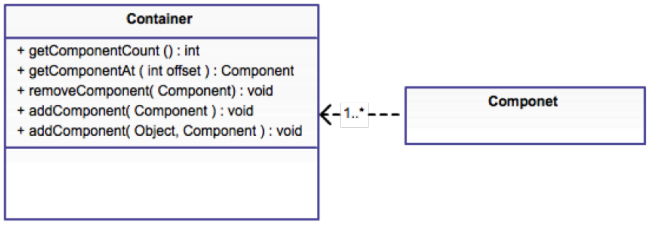


Figure 15. Component UML

A Codename One application is effectively a series of forms, but one `Form` can be visible at a time. The form contains everything on the screen. Under the hood, the `Form` has a few separate parts:

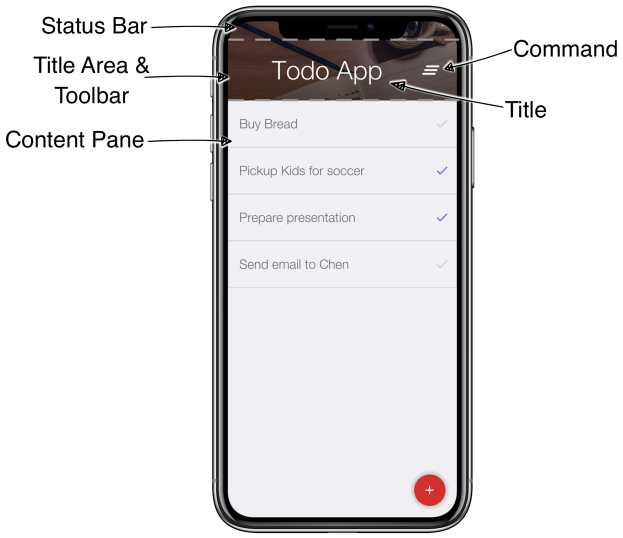


Figure 16. Structure of a Form

- Content Pane - this is the body of the `Form`. When you add a `Component` to the `Form`, it goes into the content pane. The content pane scrolls vertically by default.
- Title Area - you can't add directly to this area. The title area is managed by the `Toolbar` class. `Toolbar` sits at the top of the form and handles the title design. The title area has two parts:
  - Title of the `Form` and its commands (the buttons on the right and left of the title)
  - Status Bar - on iOS, the top area leaves room for the notch, battery, clock, and similar indicators. Without that space, those elements would overlap the title.

Now that the structure is clear, open the Java file `TodoApp.java` in the project you created. You should see the lines that set up the UI in the `start()` method:

3.1.1. Layout managers

A layout manager decides the size and location of components within a `Container`. Every `Container` has a layout manager. The default layout manager is `FlowLayout`.

To understand layouts, first understand a basic concept about `Component`: each component has a "preferred size." That's the size a component wants to occupy. For example, a `Label` uses the exact size needed to fit the label text, icon, and padding.

Understanding Preferred Size

The Component [https://www.codenameone.com/javadoc/com/codename1/ui/Component.html] class contains many useful methods. One of the most important is `calcPreferredSize()`, which recalculates the size a component wants when something changes.



By default Codename One invokes the `getPreferredSize()` method rather than `calcPreferredSize()` directly.
`getPreferredSize()` invokes `calcPreferredSize()` and caches the value.

The preferred size depends on internal constraints such as font size, border sizes, and padding.

When a layout manager positions and sizes the component, it **MIGHT** take the preferred size into account. It can also ignore it entirely.

For example, FlowLayout [https://www.codenameone.com/javadoc/com/codename1/ui/layouts/FlowLayout.html] always gives components their exact preferred size, while BorderLayout [https://www.codenameone.com/javadoc/com/codename1/ui/layouts/BorderLayout.html] resizes the center component by default and the other components along one axis.

You can define a group of components to have the same preferred width or height by using the `setSameWidth` and `setSameHeight` methods, for example:

Codename One has a `setPreferredSize` method that lets developers explicitly request a component size. But, that caused many problems. For example, preferred size should change with device orientation or similar operations. The API also encouraged accidental hardcoding of UI values such as forcing pixel sizes. As a result, the method was deprecated.

Use `setSameWidth` and `setSameHeight` when you need sizing alignment. Use `setHidden` when you need to hide a component. As a last resort, override `calcPreferredSize`.

A layout manager places a component based on its own logic and the preferred size, sometimes called the “natural size.” A `FlowLayout` walks the components in the order they were added and sizes them one after another. When it reaches the end of the row, it moves to the next row.



Use `FlowLayout` *for simple things*

`FlowLayout` is great for simple cases, but it has issues when components change size dynamically, such as a text field. In those cases, it can choose bad line breaks and take up too much space.

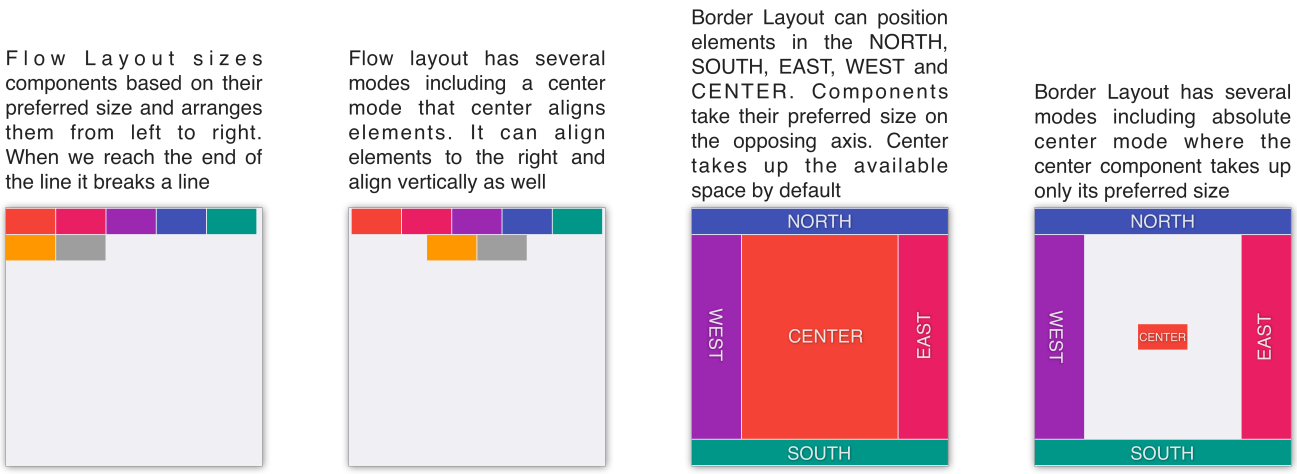


Figure 17. Layout Manager Primer Part I

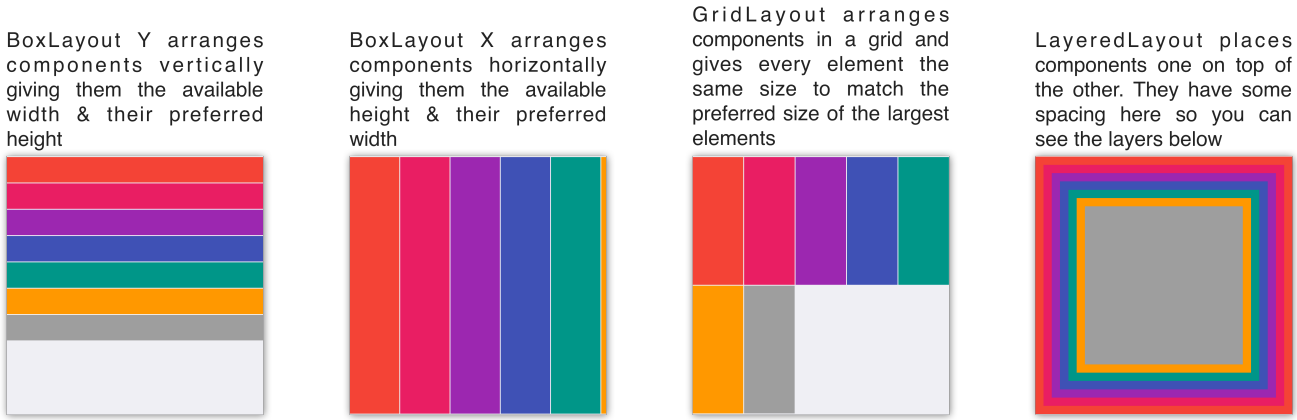


Figure 18. Layout Manager Primer Part II

Scrolling doesn't work well for all layout types because the positioning algorithm within a layout can break. Scrolling on the Y axis works well for `BoxLayout Y`, which is why it was picked for the `TodoForm`:

Table 1. Scrolling in Layout Managers

Layout	**Scrollable**
Flow Layout	Possible on Y axis only
Border Layout	Scrolling is blocked
Box Layout Y	Scrollable only on the Y axis
Box Layout X	Scrollable only on the X axis
Grid Layout	Scrollable
LayeredLayout	Not scrollable (usually)

Nesting Scrollable Containers

Only one element can be scrollable within the hierarchy. Otherwise, if you drag your finger over the `Form`, Codename One can't know which element you want to scroll. By default, the form's content pane scrolls on the Y axis unless you disable it explicitly. Setting the layout to `BorderLayout` disables scrolling implicitly.

It's fine to use non-scrollable layouts such as `BorderLayout` inside a scrollable container. In the TodoApp, for example, `TodoItem` uses `BorderLayout` inside a scrollable `BoxLayout Form`.

Layouts can be divided into two distinct groups:

- Constraint Based - `BorderLayout` (and a few others such as `GridBagLayout`, `MigLayout`, and `TableLayout`)
- Regular - All the other layout managers

When you add a `Component` to a `Container` with a regular layout you do so with a simple add method:

Listing 2. Adding to a Regular Container

```
Container cnt = new Container(BoxLayout.y());
cnt.add(new Label("Just Added"));
```

This works great for regular layouts but might not for constraint based layouts. A constraint based layout accepts another argument. For example: `BorderLayout` needs a location for the `Component`:

This line assumes you have an `import static com.codename1.ui.CN.*;` in the top of the file. In `BorderLayout` (which is a constraint based layout) placing an item in the `NORTH` places it in the top of the `Container`.



The `CN` class is a class that contains many static helper methods and functions. It's specifically designed for static import in this way to help keep your code terse

Static Global Context

The `CN` class is a thin wrapper around features in `Display`, `NetworkManager`, `Storage`, `FileSystemStorage` etc. It also adds common methods and constants from several other classes so Codename One code feels more terse for example, you can do:



That's optional, if you don't like static imports you can just write `CN.` for every element

From that point on you can write code that looks like this:

Instead of:

The same applies for most network manager calls e.g.:

Instead of:

Some things were changed so you won't have too many conflicts for example, `Log.p` or `Log.e` would have been problematic so you now have:

Instead of `Display.getInstance().getCurrent()` you now have `getCurrentForm()` since `getCurrent()` is too generic. For most methods you should just be able to remove the `NetworkManager` or `Display` access and it should "just work."

The motivation for this is threefold:

- Terse code
- Small performance gain
- Cleaner API without some baggage in `Display` or `NetworkManager`

Some of your samples in this guide might rely on that static import being in place. This helps

you keep the code terse and readable in the code listings.

Terse syntax

Almost every layout allows you to `add` a component using many variants of the add method:

Listing 3. Versions of add

```
Container cnt = new Container(BoxLayout.y());
cnt.add(new Label("Just Added")); ①
cnt.addAll(new Label("Adding Multiple"), ②
    new Label("Second One"));

cnt.add(new Label("Chaining")). ③
    add(new Label("Value"));
```

① Regular add

② `addAll` accepts many components and adds them in a batch

③ `add` returns the parent `Container` instance so you can chain calls like that

In the race to make code “tighter” you can make this even shorter. Most layout managers have their own custom terse syntax style for example:

To sum this up, you can use layout managers and nesting to create elaborate UI’s that implicitly adapt to different screen sizes and device orientation.

Flow Layout

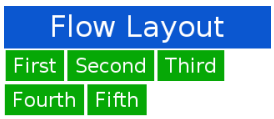


Figure 19. Flow Layout

Flow layout [https://www.codenameone.com/javadoc/com/codename1/ui/layouts/FlowLayout.html] lets the components "flow" horizontally and break a line when reaching the edge of the container. It’s the default layout manager for containers. Because it’s so flexible it’s also problematic as it can result in wrong preferred size values for the parent Container [https://www.codenameone.com/javadoc/com/codename1/ui/Container.html]. This can create a reflow issue, as a result recommend using flow layout for trivial cases. Avoid it for things such as text input etc. As the size of the text input can vary in runtime:

```
Form hi = new Form("Flow Layout", new FlowLayout());
hi.add(new Label("First")).
    add(new Label("Second")).
    add(new Label("Third")).
    add(new Label("Fourth")).
    add(new Label("Fifth"));
hi.show();
```

Flow layout also supports terse syntax shorthand such as:

```
Container flowLayout = FlowLayout.encloseIn(
        new Label("First"),
        new Label("Second"),
        new Label("Third"),
        new Label("Fourth"),
        new Label("Fifth"));
```

Flow layout can be aligned to the left (the default), to the center, or to the right. It can also be vertically aligned to the top (the default), middle (center), or bottom.

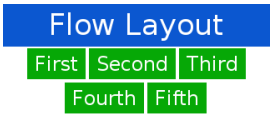


Figure 20. Flow layout aligned to the center

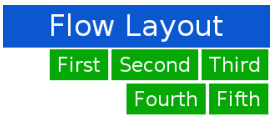


Figure 21. Flow layout aligned to the right

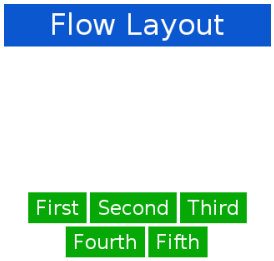


Figure 22. Flow layout aligned to the center horizontally & the middle vertically

Components within the flow layout get their natural preferred size by default and aren't stretched in any axis.



The natural sizing behavior is often used to prevent other layout managers from stretching components. For example: if you have a border layout element in the south and you want it to keep its natural size instead of adding the element to the south directly you can wrap it using `parent.add(BorderLayout.SOUTH, FlowLayout.encloseCenter(dontGrowThisComponent))`.

Box Layout

BoxLayout [https://www.codenameone.com/javadoc/com/codename1/ui/layouts/BoxLayout.html] places elements in a row (`X_AXIS`) or column (`Y_AXIS`) according to box orientation. Box is a simple and predictable layout that serves as the "workhorse" of component lists in Codename One.

You can create a box layout Y using something like this:

```
Form hi = new Form("Box Y Layout", new BoxLayout(BoxLayout.Y_AXIS));
hi.add(new Label("First")).
    add(new Label("Second")).
    add(new Label("Third")).
    add(new Label("Fourth")).
    add(new Label("Fifth"));
```

Which results in this

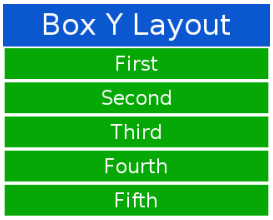


Figure 23. BoxLayout Y

Box layout also supports a shorter terse notation which you use here to show the X axis box:

```
Container box = BoxLayout.encloseX(new Label("First"),
        new Label("Second"),
        new Label("Third"),
        new Label("Fourth"),
        new Label("Fifth"));
```

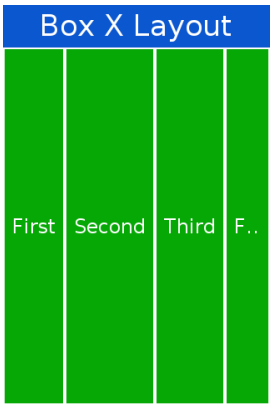


Figure 24. BoxLayout X

The box layout keeps the preferred size of its destination orientation and scales elements on the other axis. Specifically `X_AXIS` will keep the preferred width of the component while growing all the components vertically to match in size. Its `Y_AXIS` counterpart keeps the preferred height while growing the components horizontally.

This behavior is useful since it allows elements to align as they would all have the same size.

Sometimes the growing behavior in the X axis is undesired, for these cases you can use the `X_AXIS_NO_GROW` variant.



Figure 25. BoxLayout X_AXIS_NO_GROW



`FlowLayout` *vs.* `BoxLayout.X_AXIS`

When applicable recommend `BoxLayout` over `FlowLayout` as it acts more consistently in all situations. Another advantage of `BoxLayout` is the fact that it grows and thus aligns the components in a consistent column or row

Border Layout

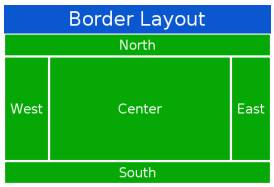


Figure 26. Border Layout

Border layout [https://www.codenameone.com/javadoc/com/codename1/ui/layouts/BorderLayout.html] is unique. `BorderLayout` is a constraint-based layout that can place up to five components in one of the five positions: `NORTH`, `SOUTH`, `EAST`, `WEST` or `CENTER`:

```
Form hi = new Form("Border Layout", new BorderLayout());
hi.add(BorderLayout.CENTER, new Label("Center")).
    add(BorderLayout.SOUTH, new Label("South")).
    add(BorderLayout.NORTH, new Label("North")).
    add(BorderLayout.EAST, new Label("East")).
    add(BorderLayout.WEST, new Label("West"));
hi.show();
```



The Constraints are Included in the `CN` *class*

You can use the static import of the `CN` class and then the syntax can be `add(SOUTH, new Label("South"))`

The layout always stretches the `NORTH`/`SOUTH` components on the X-axis to fill the container and the `EAST`/`WEST` components on the Y-axis. The center component is stretched to fill the remaining area by default. For example, the `setCenterBehavior` allows you to manipulate the behavior of the center component so it's placed in the center without stretching.

For example:

```
Form hi = new Form("Border Layout", new BorderLayout());
((BorderLayout)hi.getLayout()).setCenterBehavior(BorderLayout.CENTER_BEHAVIOR_CENTER);
hi.add(BorderLayout.CENTER, new Label("Center")).
    add(BorderLayout.SOUTH, new Label("South")).
    add(BorderLayout.NORTH, new Label("North")).
    add(BorderLayout.EAST, new Label("East")).
    add(BorderLayout.WEST, new Label("West"));
hi.show();
```

Results in:

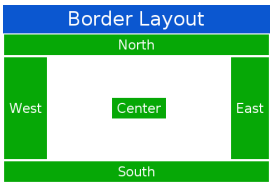


Figure 27. Border Layout with CENTER_BEHAVIOR_CENTER



Scrolling is Disabled in Border Layout

Because of its scaling behavior scrolling a border layout makes no sense. `Container` implicitly blocks scrolling on a border layout, but it can scroll its parents/children

For RTL the EAST and WEST values are implicitly reversed as shown in this image:

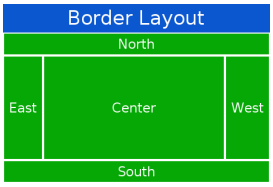


Figure 28. Border Layout in RTL mode

RTL and Bidi

RTL (Right To Left) or Bidi (bi-directional) are common terms used for languages such as Hebrew, Arabic etc. These languages are written from the right to left direction hence all the UI needs to be "reversed." Bidi denotes the fact that while the language is written from right to left, the numbers are still written in the other direction hence two directions...



Preferred Size Still Matters

The preferred size of the center component doesn't matter in border layout but the preferred size of the sides is. For example: If you place a large component in the `SOUTH` it will take up the entire screen and won't leave room for anything

Grid Layout

GridLayout [https://www.codenameone.com/javadoc/com/codename1/ui/layouts/GridLayout.html] accepts a predefined grid (rows/columns) and grants all components within it equal size based on the

dimensions of the largest components.



The main use case for this layout is a grid of icons for example: like one would see in the iPhone home screen

If the number of `rows * columns` is smaller than the number of components added a new row is implicitly added to the grid. For example, if the number of components is smaller than available cells (won't fill the last row) blank spaces will be left in place.

In this example you can see that a 2×2 grid is used to add 5 elements, this results in an additional row that's implicitly added turning the grid to a 3×2 grid implicitly and leaving one blank cell:

```
Form hi = new Form("Grid Layout 2x2", new GridLayout(2, 2));
hi.add(new Label("First")).
    add(new Label("Second")).
    add(new Label("Third")).
    add(new Label("Fourth")).
    add(new Label("Fifth"));
```

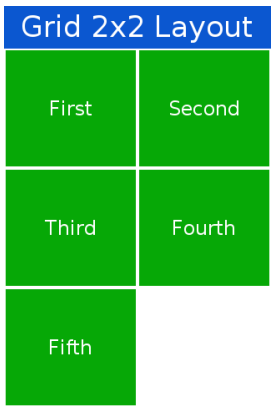


Figure 29. Grid Layout 2×2

When you use a 2×4 size ratio you would see elements getting cropped as you do here. The grid layout uses the grid size first and doesn't pay too much attention to the preferred size of the components it holds.

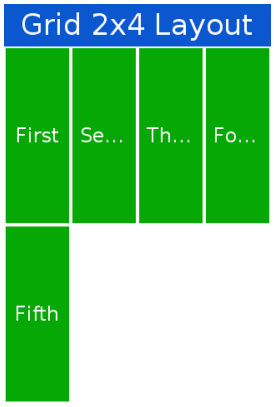


Figure 30. Grid Layout 2×4

Grid also has an autoFit attribute that can be used to automatically calculate the column count based on available space and preferred width. This is useful for working with UI's where the device

orientation might change.

A terse syntax for working with a grid also exists in two versions, one that uses the "auto-fit" option and another that accepts the number of columns. Here's a sample of the terse syntax coupled with auto-fit followed by screenshots of the same code in two orientations:

```
GridLayout.encloseIn(new Label("First"),
    new Label("Second"),
    new Label("Third"),
    new Label("Fourth"),
    new Label("Fifth"));
```

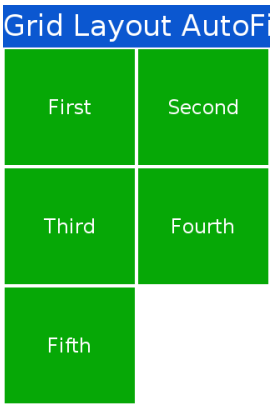


Figure 31. Grid Layout autofit portrait

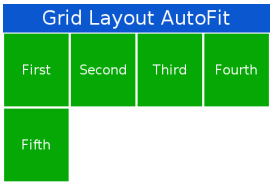


Figure 32. Grid Layout autofit landscape

3.1.2. Table Layout

The TableLayout [https://www.codenameone.com/javadoc/com/codename1/ui/table/TableLayout.html] is an elaborate **constraint based** layout manager that can arrange elements in rows/columns while defining constraints to control complex behavior such as spanning, alignment/weight etc.



Note the Different Package for `TableLayout`

The `TableLayout` is in the `com.codename1.ui.table` package and not in the layouts package.
This is because `TableLayout` was originally designed for the Table [https://www.codenameone.com/javadoc/com/codename1/ui/table/Table.html] class.

Despite being constraint based the `TableLayout` isn't strict about constraints and will implicitly add a constraint when one is missing. This is unlike the `BorderLayout` which will throw an exception in this case.



Unlike `GridLayout` `TableLayout` won't implicitly add a row if the row/column count

> is wrong

```
Form hi = new Form("TableLayout", new TableLayout(2, 2));
hi.add(new Label("First"));
hi.add(new Label("Second"));
hi.add(new Label("Third"));
hi.add(new Label("Fourth"));
hi.add(new Label("Fifth"));
hi.show();
```

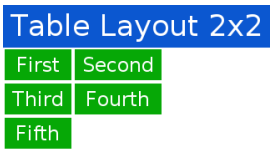


Figure 33. 2×2 TableLayout with 5 elements, notice that the last element is missing

`TableLayout` supports the ability to grow the last column which can be enabled using the `setGrowHorizontally` method. You can also use a shortened terse syntax to construct a `TableLayout` but since the `TableLayout` is a constraint based layout you won't be able to use its full power with this syntax.

The default usage of the `encloseIn` method below uses the `setGrowHorizontally` flag:

```
Container table = TableLayout.encloseIn(2,
        new Label("First"),
        new Label("Second"),
        new Label("Third"),
        new Label("Fourth"),
        new Label("Fifth"));
Form hi = new Form("TableLayout Enclose 2", new BorderLayout());
hi.add(BorderLayout.CENTER, table);
hi.show();
```

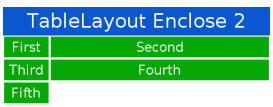


Figure 34. `TableLayout.encloseIn()` *with default behavior of growing the last column*

The full potential

`TableLayout` is a beast, to truly appreciate it you need to use the constraint syntax which allows you to span, align and set width/height for the rows and columns.

`TableLayout` works with a Constraint [https://www.codenameone.com/javadoc/com/codename1/ui/table/TableLayout.Constraint.html] instance that can communicate your intentions into the layout manager. Such constraints can include more than one attribute for example: span and height.



`TableLayout` constraints can't be reused for more than one component

The constraint class supports the following attributes

Table 2. Constraint Properties

`column`	The column for the table cell. This defaults to -1 which will place the component in the next available cell
`row`	Like column, defaults to -1 as well
`width`	The column width in percentages, -1 will use the preferred size. -2 for width will take up the rest of the available space
`height`	Like width but doesn't support the -2 value
`spanHorizontal`	The cells that should be occupied horizontally defaults to 1 and can't exceed the column count - current offset.
`spanVertical`	Like spanHorizontal with the same limitations
`horizontalAlign`	The horizontal alignment of the content within the cell, defaults to the special case -1 value to take up all the cell space can be either -1, `Component.LEFT`, `Component.RIGHT` or `Component.CENTER`
`verticalAlign`	Like horizontalAlign can be one of -1, `Component.TOP`, `Component.BOTTOM` or `Component.CENTER`



You need to set `width`/`height` to one cell in a column/row

The table layout constraint sample tries to show some unique things you can do with constraints.

The constraint sample below spans the title across all three columns and spans the notes row across two columns:

```
TableLayout layout = new TableLayout(4, 3);
layout.setGrowHorizontally(true);
Form hi = new Form("Table Layout", layout);
```

```
TableLayout.Constraint title = layout.createConstraint();
title.setHorizontalSpan(3);
title.setHorizontalAlign(Component.CENTER);
hi.add(title, new Label("Invoice"));

hi.add(new Label("Item"));
hi.add(new Label("Qty"));
hi.add(new Label("Total"));
hi.add(new Label("Design"));
hi.add(new Label("2"));
hi.add(new Label("$120"));

TableLayout.Constraint notes = layout.createConstraint();
notes.setHorizontalSpan(2);
notes.setHeightPercentage(40);
hi.add(notes, new SpanLabel("Notes span two columns"));
hi.add(new Button("Pay"));

hi.show();
```

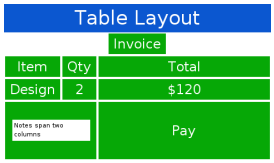


Figure 35. TableLayout constraints can be used to create elaborate UI's

TextMode Layout

`TextModeLayout` is a unique layout manager. It acts like `TableLayout` on Android and like `BoxLayout.Y_AXIS` in other platforms. Internally it delegates to one of these two layout managers so in a sense it doesn't have as much functionality of its own.

For example: this is a sample usage of `TextModeLayout`:

```
TextModeLayout tl = new TextModeLayout(3, 2);
Form f = new Form("Pixel Perfect", tl);
TextComponent title = new TextComponent().label("Title");
TextComponent price = new TextComponent().label("Price");
TextComponent location = new TextComponent().label("Location");
TextComponent description = new TextComponent().label("Description").multiline(true);

f.add(tl.createConstraint().horizontalSpan(2), title);
f.add(tl.createConstraint().widthPercentage(30), price);
f.add(tl.createConstraint().widthPercentage(70), location);
f.add(tl.createConstraint().horizontalSpan(2), description);
f.setEditOnShow(title.getField());
f.show();
```

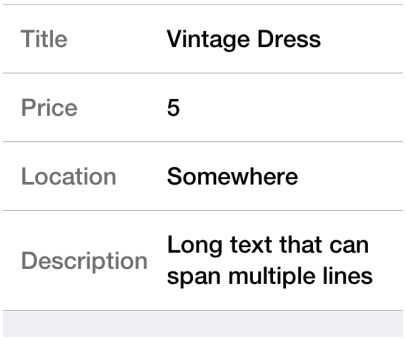


Figure 36. TextModeLayout on iOS

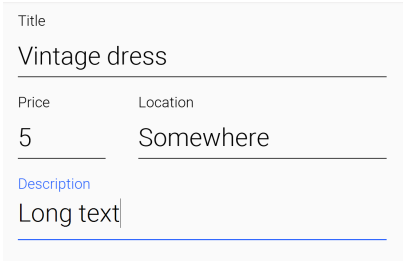


Figure 37. TextModeLayout on Android with the same code

As you can see from the code and samples above there is a lot going on under the hood. On Android you want a layout that's like `TableLayout` so you can "pack" the entries. On iOS you want a box layout Y kind of layout but you also want the labels/text to align...

The `TextModeLayout` isn't a layout as much as it's a delegate. When running in the Android mode (which you refer to as the "on top" mode) the layout is almost an exact synonym of `TableLayout` and in fact delegates to an underlying `TableLayout`. In fact there is a `public final` table instance within the layout that you can refer to directly...

One small difference exists between the `TextModeLayout` and the underlying `TableLayout`: your choice to default to align entries to `TOP` with this mode.



Aligning to TOP is important for error handling for `TextComponent` in Android otherwise the entries "jump"

When working in the non-android environment you use a `BoxLayout` on the Y axis as the delegate. One thing you do here differs from a default box layout: grouping. Grouping allows the labels to align by setting them to the same width, internally it invokes `Component.setSameWidth()`. Since text components hide the labels there is a special `group` method there that can be used. For example, this is implicit with the `TextModeLayout` which is pretty cool.

`TextModeLayout` was created specifically for the `TextComponent` and `InputComponent` so check out the section about them in the components chapter.

3.1.3. Layered Layout

When used without constraints, the LayeredLayout [https://www.codenameone.com/javadoc/com/codename1/ui/layouts/LayeredLayout.html] places the components in order one on top of the other and sizes them all to the size of the largest component. This is useful when trying to create an overlay on top of an existing component. For example: an "x" button to allow removing the component.



Figure 38. The X on this button was placed there using the layered layout code below

The code to generate this UI is slightly complex and contains few relevant pieces. The truly relevant piece is this block:

```
hi.add(LayeredLayout.encloseIn(settingsLabel,
        FlowLayout.encloseRight(close)));
```

You're doing three distinct things here:

1. You're adding a layered layout to the form
2. You're creating a layered layout and placing two components within. This would be the equivalent of creating a `LayeredLayout` `Container` and invoking `add` twice
3. You use `FlowLayout` to position the `X` close button in the right position



When used without constraints, the layered layout sizes all components to the exact same size one on top of the other. It requires that you use another container within; to position the components correctly

This is the full source of the example for completeness:

```
Form hi = new Form("Layered Layout");
int w = Math.min(Display.getInstance().getDisplayWidth(), Display.getInstance().getDisplayHeight());
Button settingsLabel = new Button("");
Style settingsStyle = settingsLabel.getAllStyles();
settingsStyle.setFgColor(0xff);
settingsStyle.setBorder(null);
settingsStyle.setBgColor(0xff00);
settingsStyle.setBgTransparency(255);
settingsStyle.setFont(settingsLabel.getUnselectedStyle().getFont().derive(w / 3, Font.STYLE_PLAIN));
FontImage.setMaterialIcon(settingsLabel, FontImage.MATERIAL_SETTINGS);
Button close = new Button("");
close.setUIID("Container");
close.getAllStyles().setFgColor(0xff0000);
FontImage.setMaterialIcon(close, FontImage.MATERIAL_CLOSE);
hi.add(LayeredLayout.encloseIn(settingsLabel,
        FlowLayout.encloseRight(close)));
```

Forms have a built in layered layout that you can access through `getLayeredPane()`, this allows you

to overlay elements on top of the content pane.

The layered pane is used internally by components such as InteractionDialog [https://www.codenameone.com/javadoc/com/codename1/components/InteractionDialog.html], AutoComplete [https://www.codenameone.com/javadoc/com/codename1/ui/AutoCompleteTextField.html] etc.



Codename One also includes a GlassPane that resides on top of the layered pane. Its useful if you want to "draw" on top of elements but is harder to use than layered pane

Insets and reference components

LayeredLayout [https://www.codenameone.com/javadoc/com/codename1/ui/layouts/LayeredLayout.html] supports insets [https://www.codenameone.com/javadoc/com/codename1/ui/layouts/LayeredLayout.LayeredLayoutConstraint.Inset.html] for its children. This effectively allows you to position child components precisely where you want them, relative to their container or siblings. This functionality forms the under-pinnings of the GUI Builder's Auto layout mode.

As an example, suppose you wanted to position a button in the lower right corner of its container. This can be achieved with LayeredLayout [https://www.codenameone.com/javadoc/com/codename1/ui/layouts/LayeredLayout.html] as follows:

```
Container cnt = new Container(new LayeredLayout());
Button btn = new Button("Submit");
LayeredLayout ll = (LayeredLayout)cnt.getLayout();
cnt.add(btn);
ll.setInsets(btn, "auto 0 0 auto");
```

The result is:





The thing new here is this line:

```
ll.setInsets(btn, "auto 0 0 auto");
```

This is called after `btn` has already been added to the container. It says that you want its insets to be "auto" on the top and left, and `0` on the right and bottom. This **insets** string follows the CSS notation of `top right bottom left` (that's: start on top and go clockwise), and the values of each inset may be

provided in pixels (px), millimetres (mm), percent (%), or the special "auto" value. Like CSS, you can also specify the insets using a 1, 2, or 3 values. For example:

1. `"1mm"` - Sets 1mm insets on all sides.
2. `"1mm 2mm"` - Sets 1mm insets on top and bottom; 2mm on left and right.
3. `"1mm 10% 2mm"` - Sets 1mm on top, 10% on left and right, and 2mm on bottom.
4. `"1mm 2mm 1px 50%"` - Sets 1mm on top, 2mm on right, 1px on bottom, and 50% on left.

`auto` insets

The special "auto" inset indicates that it's a flexible inset. If all insets are set to "auto," then the component will be centered both horizontally and vertically inside its "bounding box."



The "inset bounding box" is the containing box from which a component's insets are measured. If the component's insets aren't linked to any other components, then its inset bounding box will be the inner bounds (that's: taking padding into account) of the component's parent container.

If one inset is fixed (that's: defined in px, mm, or %), and the opposite inset is "auto," then the "auto" inset will allow the component to be its preferred size. If you want to position a component to be centered vertically, and 5mm from the left edge, you could do:

```
ll.setInsets(btn, "auto auto auto 5mm");
```

Resulting in:

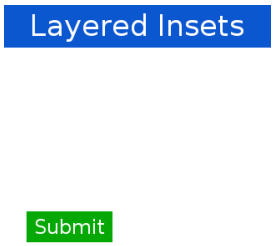


Figure 39. Button vertically centered 5mm from left edge

Move it to the right edge with:

```
ll.setInsets(btn, "auto 5mm auto auto");
```

% insets

Percent (%) insets are calculated about the inset bounding box. A 50% inset is measured as 50% of the length of the bounding box on the inset's axis. For example: A 50% inset on top would be 50% of

the height of the inset bounding box. A 50% inset on the right would be 50% of the width of the inset bounding box.

Insets, margin, and padding

A component's position in a layered layout is determined as follows: (Assume that `cmp` is the component that you're positioning, and `cnt` is the container (In pseudo-code):

```
x = cnt.paddingLeft + cmp.calculatedInsetLeft + cmp.marginLeft
y = cnt.paddingTop + cmp.calculatedInsetTop + cmp.marginTop
w = cnt.width - cnt.verticalScroll.width - cnt.paddingRight - cmp.calculatedInsetRight - cmp.marginRight - x
h = cnt.height - cnt.horizontalScroll.height - cnt.paddingBottom - cmp.calculatedInsetBottom - cmp.marginBottom - y
```



The `calculatedInsetXXX` values here will be the same as the corresponding provided inset if the inset has no reference component. If it does have a reference component, then the calculated inset will depend on the position of the reference component.

If no inset is specified, then it's assumed to be 0. This ensures compatibility with designs that were created before layered layout supported insets.

Component references: Linking components together

If all you need to do is position a component relative to its parent container's bounds, then mere insets provide you with enough vocabulary to achieve this. But most UIs are more complex than this and require another concept: reference components. Often you will want to position a component relative to another child of the same container. This is also supported.

For example, suppose you want to place a text field in the center of the form (both horizontally and vertically), and have a button placed beside it to the right. Positioning the text field is trivial (`setInset(textField, "auto")`), but there is no inset that you can provide that would position the button to the right of the text field. To do your goal, you need to set the text field as a reference component of the button's left inset, which "links" the button's left inset to the text field. Here is the syntax:

```
Container cnt = new Container(new LayeredLayout());
LayeredLayout ll = (LayeredLayout)cnt.getLayout();
Button btn = new Button("Submit");
TextField tf = new TextField();
cnt.add(tf).add(btn);
ll.setInsets(tf, "auto")
  .setInsets(btn, "auto auto auto 0")
  .setReferenceComponentLeft(btn, tf, 1f);
```

This would result in:

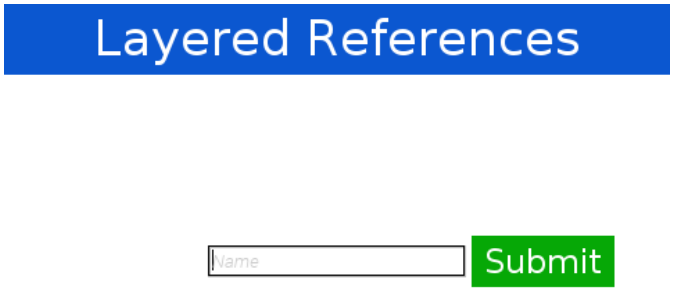


Figure 40. Button's left inset linked to text field

The two active lines here are the last two:

```
ll.setInsets(tf, "auto")
  .setInsets(btn, "auto auto auto 0")
  .setReferenceComponentLeft(btn, tf, 1f);
```



The reference position is defined as the distance, expressed as a fraction of the reference component's length on the inset's axis, between the reference component's leading (outer) edge and the point from which the inset is measured. A reference position of 0 means that the inset is measured from the leading edge of the reference component. A value of 1.0 means that the inset is measured from the trailing edge of the reference component. A value of 0.5 means that the inset is measured from the center of the reference component. Etc. Any floating point value can be used, though the most common values are 0 and 1.

The definition above may make reference components and reference position seem more complex than they are. Some examples:

1. **For a top inset**:
   a. referencePosition == 0 ⇒ the inset is measured from the top edge of the reference component.
   b. referencePosition == 1 ⇒ the inset is measured from the bottom edge of the reference component.
2. **For a bottom inset**:
   a. referencePosition == 0 ⇒ the inset is measured from the **bottom** edge of the reference component.
   b. referencePosition == 1 ⇒ the inset is measured from the **top** edge of the reference component.
3. **For a left inset**:
   a. referencePosition == 0 ⇒ the inset is measured from the **left** edge of the reference component.

b. referencePosition == 1 ⇒ the inset is measured from the **right** edge of the reference component.

4. **For a right inset**:
   a. referencePosition == 0 ⇒ the inset is measured from the **right** edge of the reference component.
   b. referencePosition == 1 ⇒ the inset is measured from the **left** edge of the reference component.

Layers In Codename One

Codename One allows placing components one on top of the other and you commonly use layered layout to do that. The form class has a built-in `Container` that resides in a layer on top of the content pane of the form.

When you add an element to a form it implicitly goes into the content pane. But, you can use `getLayeredPane()` and add any `Component` there. Such a `Component` will appear above the content pane. Notice that this layer resides below the title area (on the Y axis) and won't draw on top of that.

When Codename One introduced the layered pane it was instantly useful. But, its popularity caused conflicts. Two separate pieces of code using the layered pane could collide with one another. Codename One solved it with `getLayeredPane(Class c, boolean top)`. This method allocates a layer for a specific class within the layered pane. This way if two different classes use this method instead of the `getLayeredPane()` method they won't collide. Each will get its own container in a layered layout within the layered pane seamlessly. The `top` flag indicates whether you want the layer to be the top most or bottom most layer within the layered pane (assuming it wasn't created already). This allows you to place a layer that can appear above or below the already installed layers.

You only make use of the layered pane in this book but there are two more layers on top of it. The form layered pane is identical to the layered pane but spans the entire height of the `Form` (including the title area). As a result the form layered pane is slower as it needs to handle some special cases to support this functionality.

The glass pane is the top most layer, unlike the layered pane it's purely a graphical layer. You can only draw on the glass pane with a `Painter` instance and a `Graphics` object. You can't add components into that layer.

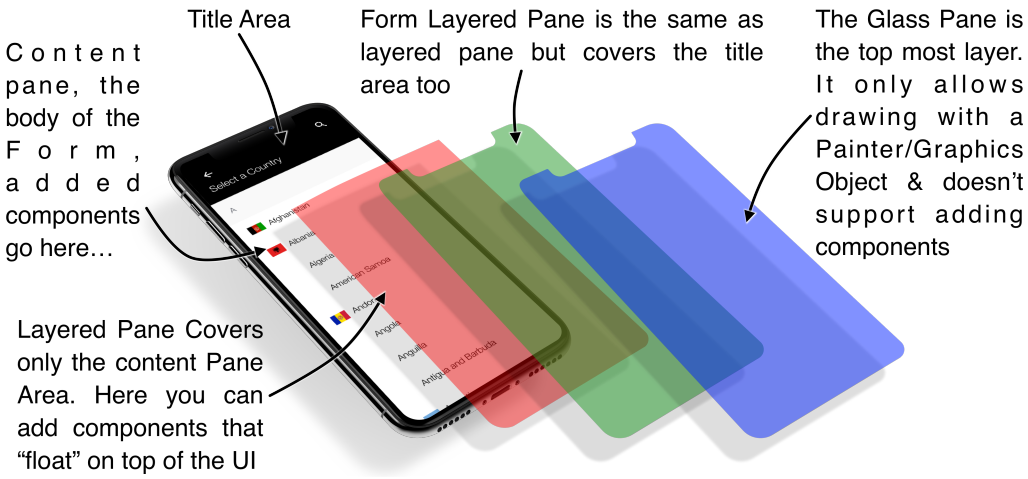


Figure 41. The Layered Pane

3.1.4. GridBag Layout

GridBagLayout [https://www.codenameone.com/javadoc/com/codename1/ui/layouts/GridBagLayout.html] was introduced to simplify the process of porting existing Swing/AWT code with a more familiar API. The API for this layout is problematic as it was designed for AWT/Swing where styles were unavailable. As a result it has its own insets API instead of using elements such as padding/margin.

Your recommendation is to use Table which is as powerful but has better Codename One integration.

To show `GridBagLayout` you ported http://docs.oracle.com/JavaSE/tutorial/uiswing/layout/gridbag.html[the sample from the Java tutorial] to Codename One:

```
private static Button gridButton(String text) {
    Button button = new Button(text);
    button.setCapsText(false);
    Style style = button.getAllStyles();
    style.setBgColor(0xf4f8ff);
    style.setBgTransparency(255);
    style.setFgColor(0x0d47a1);
    style.setBorder(Border.createLineBorder(1, 0x2b5c9e));
    style.setPaddingUnit(Style.UNIT_TYPE_DIPS);
    style.setPadding(2, 2, 2, 2);
    return button;
}

public static Form createForm() {
    Form hi = new Form("GridBagLayout", new BorderLayout());
    Container grid = new Container(new GridBagLayout());
    Style gridStyle = grid.getAllStyles();
    gridStyle.setPaddingUnit(Style.UNIT_TYPE_DIPS);
    gridStyle.setPadding(4, 4, 4, 4);
    Button button;
```

```
    GridBagConstraints c = new GridBagConstraints();
    //natural height, maximum width
    c.fill = GridBagConstraints.HORIZONTAL;
    button = gridButton("One");
    c.weightx = 0.5;
    c.fill = GridBagConstraints.HORIZONTAL;
    c.gridx = 0;
    c.gridy = 0;
    grid.addComponent(c, button);

    button = gridButton("Two");
    c.fill = GridBagConstraints.HORIZONTAL;
    c.weightx = 0.5;
    c.gridx = 1;
    c.gridy = 0;
    grid.addComponent(c, button);

    button = gridButton("Three");
    c.fill = GridBagConstraints.HORIZONTAL;
    c.weightx = 0.5;
    c.gridx = 2;
    c.gridy = 0;
    grid.addComponent(c, button);

    button = gridButton("Long-Named Button 4");
    c.fill = GridBagConstraints.HORIZONTAL;
    c.ipady = 40;      //make this component tall
    c.weightx = 0.0;
    c.gridwidth = 3;
    c.gridx = 0;
    c.gridy = 1;
    grid.addComponent(c, button);

    button = gridButton("5");
    c.fill = GridBagConstraints.HORIZONTAL;
    c.ipady = 0;       //reset to default
    c.weighty = 1.0;   //request any extra vertical space
    c.anchor = GridBagConstraints.PAGE_END; //bottom of space
    c.insets = new Insets(10,0,0,0);  //top padding
    c.gridx = 1;       //aligned with button 2
    c.gridwidth = 2;   //2 columns wide
    c.gridy = 2;       //third row
    grid.addComponent(c, button);
    hi.add(BorderLayout.NORTH, grid);
    return hi;
}
```

Notice that because of the way gridbag works you didn't provide any terse syntax API for it although it should be possible.

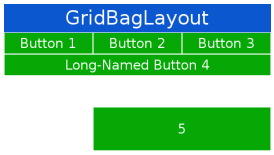


Figure 42. GridbagLayout sample from the Java tutorial running on Codename One

3.1.5. Group Layout

GroupLayout [https://www.codenameone.com/javadoc/com/codename1/ui/layouts/GroupLayout.html] is a layout that would be familiar to the users of the NetBeans GUI builder (Matisse) [https://netbeans.org/features/java/swing.html]. Its a layout manager that's hard to use for manual coding but is powerful for some elaborate use cases. Although `MiGLayout` and `LayeredLayout` might be superior options.

It was originally added during the LWUIT days as part of an internal try to port Matisse to LWUIT. It's still useful to this day as developers copy and paste Matisse code into Codename One and produce elaborate layouts with drag.

Since the layout is based on an older version of `GroupLayout` some things need to be adapted in the code or you should use the special "compatibility" library for Matisse to get better interaction. You also recommend tweaking Matisse to use import statements instead of full package names, that way if you use `Label` changing the awt import to a Codename One import will make it use work for Codename One's `Label`.

Unlike any other layout manager `GroupLayout` adds the components into the container instead of the standard API. This works for GUI builder code but as you can see from this sample it doesn't make the code readable:

```
public static Form createForm() {
    Form hi = new Form("GroupLayout");

    Label label1 = new Label();
    Label label2 = new Label();
    Label label3 = new Label();
    Label label4 = new Label();
    Label label5 = new Label();
    Label label6 = new Label();
    Label label7 = new Label();

    label1.setText("label1");

    label2.setText("label2");

    label3.setText("label3");

    label4.setText("label4");

    label5.setText("label5");

    label6.setText("label6");

    label7.setText("label7");

    GroupLayout layout = new GroupLayout(hi.getContentPane());
    hi.setLayout(layout);
    layout.setHorizontalGroup(
        layout.createParallelGroup(GroupLayout.LEADING)
```

```
        .add(layout.createSequentialGroup()
            .addContainerGap()
            .add(layout.createParallelGroup(GroupLayout.LEADING)
                .add(layout.createSequentialGroup()
                    .add(label1, GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE, GroupLayout.PREFERRED_SIZE)
                    .addPreferredGap(LayoutStyle.RELATED)
                    .add(layout.createParallelGroup(GroupLayout.LEADING)
                        .add(label4, GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE, GroupLayout.PREFERRED_SIZE)
                        .add(label3, GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE, GroupLayout.PREFERRED_SIZE)
                        .add(label2, GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE, GroupLayout.PREFERRED_SIZE)))
                .add(label5, GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE, GroupLayout.PREFERRED_SIZE)
                .add(layout.createSequentialGroup()
                    .add(label6, GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE, GroupLayout.PREFERRED_SIZE)
                    .addPreferredGap(LayoutStyle.RELATED)
                    .add(label7, GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE, GroupLayout.PREFERRED_SIZE)))
            .addContainerGap(296, Short.MAX_VALUE))
    );
    layout.setVerticalGroup(
        layout.createParallelGroup(GroupLayout.LEADING)
        .add(layout.createSequentialGroup()
            .addContainerGap()
            .add(layout.createParallelGroup(GroupLayout.TRAILING)
                .add(label2, GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE, GroupLayout.PREFERRED_SIZE)
                .add(label1, GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE, GroupLayout.PREFERRED_SIZE))
            .addPreferredGap(LayoutStyle.RELATED)
            .add(label3, GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE, GroupLayout.PREFERRED_SIZE)
            .addPreferredGap(LayoutStyle.RELATED)
            .add(label4, GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE, GroupLayout.PREFERRED_SIZE)
            .addPreferredGap(LayoutStyle.RELATED)
            .add(label5, GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE, GroupLayout.PREFERRED_SIZE)
            .addPreferredGap(LayoutStyle.RELATED)
            .add(layout.createParallelGroup(GroupLayout.LEADING)
                .add(label6, GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE, GroupLayout.PREFERRED_SIZE)
                .add(label7, GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE, GroupLayout.PREFERRED_SIZE))
            .addContainerGap(150, Short.MAX_VALUE))
    );
    return hi;
}
```

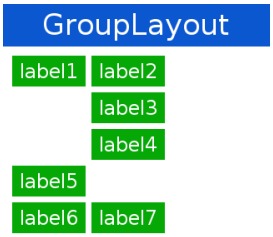


Figure 43. GroupLayout Matisse generated UI running in Codename One

If you're porting newer Matisse code there are simple changes you can do:

- Change `addComponent` to `add`
- Change `addGroup` to `add`
- Remove references to `ComponentPlacement` and reference `LayoutStyle` directly

3.1.6. Mig Layout

MigLayout [https://www.codenameone.com/javadoc/com/codename1/ui/layouts/mig/MigLayout.html] is a popular cross-platform layout manager that was ported to Codename One from Swing.



MiG is still considered experimental so proceed with caution!
The API was deprecated to serve as a warning of its experimental status.

The best reference for MiG would probably be its quick start guide (PDF link) [http://www.miglayout.com/QuickStart.pdf]. As a reference you ported one of the samples from that PDF to Codename One:

```
public static Form createForm() {
    Form hi = new Form("MigLayout",
            new MigLayout("wrap 2, insets 4mm", "[right]3mm[32mm]", "[]12[]12[]12[]"));
    hi.add(new Label("First name"));
    hi.add("growx, w 32mm", new TextField("", "First name"));
    hi.add(new Label("Last name"));
    hi.add("growx, w 32mm", new TextField("", "Last name"));
    hi.add(new Label("Phone"));
    hi.add("growx, w 32mm", new TextField("", "Phone"));
    Button ok = new Button("OK");
    ok.setCapsText(false);
    Style okStyle = ok.getAllStyles();
    okStyle.setBgColor(0xf4f8ff);
    okStyle.setBgTransparency(255);
    okStyle.setFgColor(0x0d47a1);
    okStyle.setBorder(Border.createLineBorder(1, 0x2b5c9e));
    okStyle.setPaddingUnit(Style.UNIT_TYPE_DIPS);
    okStyle.setPadding(2, 2, 2, 2);
    hi.add("span 2, growx", FlowLayout.encloseCenter(ok));
    return hi;
}
```

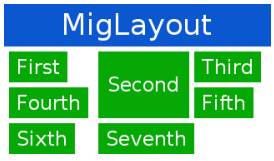


Figure 44. MiG layout sample ported to Codename One

It should be reasonably easy to port MiG code but you should notice the following:

- MiG handles a lot of the spacing/padding/margin issues that are missing in Swing/AWT. With

Codename One styles you have the padding and margin which are probably a better way to do a lot of the things that MiG does

- The `add` method in Codename One can be changed as shown in the sample above.
- The constraint argument for Codename One `add` calls appears before the `Component` instance.

3.2. Themes and styles

Next you need to introduce you to 3 important terms in Codename One: Theme, Style, and UIID.

Themes are similar conceptually to CSS, in fact they can be created with CSS syntax as this guide covers soon. The various Codename One ports ship with a native theme representing the appearance of the native OS UI elements. Every Codename One application has its own theme that derives the native theme and overrides behavior within it.

If the native theme has a button defined, you can override properties of that button in your theme. This allows you to customize the look while retaining some native appearances. This works by merging the themes to one big theme where your application theme overrides the definitions of the native theme. This is pretty like the cascading aspect of CSS if you're familiar with that.

Themes consist of a set of UIID definitions. Every component in Codename One has a UIID associated with it. UIID stands for User Interface Identifier. This UIID connects the theme to a specific component. A UIID maps to CSS classes if you're familiar with that concept. For example, Codename One doesn't support the complex CSS selector syntax options as those can impact runtime performance.

For example: see this code where:

This is a text field component (user input field) but it will look like a `Label`.

Effectively you told the text field that it should use the UIID of `Label` when it's drawing itself. It's common to do tricks like that in Codename One. For example: `button.setUIID("Label")` would make a button appear like a label and allow you to track clicks on a `Label`.

The UIID's translate the theme elements into a set of `Style` objects. These `Style` objects get their initial values from the theme but can be further manipulated after the fact. To make the text field's foreground color red you could use this code:

The color is in hexadecimal `RRGGBB` format so `0xff00` would be green and `0xff0000` would be red.

`getAllStyles()` returns a `Style` object but why do you need "all" styles?

Each component can have one of 4 states and each state has a `Style` object. This means you can have 4 style objects per Component:

- **Unselected:** used when a component isn't touched and doesn't have focus. You can get that object with `getUnselectedStyle()`.
- **Selected:** used when a component is touched or if focus is drawn for non-touch devices. You can get that object with `getSelectedStyle()`.

- **Pressed:** used when a component is pressed. Notice it's applicable to buttons and button subclasses. You can get that object with `getPressedStyle()`.
- **Disabled:** used when a component is disabled. You can get that object with `getDisabledStyle()`.

The `getAllStyles()` method returns a special case `Style` object that lets you set the values of all 4 styles from one class so the code before would be equivalent to invoking all 4 `setFgColor` methods. For example, `getAllStyles()` works for setting properties not for getting them!



don't use `getStyle()` *for manipulation*

`getStyle()` returns the current `Style` object which means it will behave inconsistently. The paint method uses `getStyle()` as it draws the current state of the `Component` but other code should avoid that method. Use the specific methods instead: `getUnselectedStyle()`, `getSelectedStyle()`, `getPressedStyle()`, `getDisabledStyle()` and `getAllStyles()`

As you can see, it's a bit of a hassle to change styles from code which is why the theme is so appealing.

3.2.1. Theme

A theme allows you to define the styles externally through a set of UIID's (User Interface ID's). Themes can be authored directly in CSS and then compiled into the Codename One resource file, which keeps styling concerns separate from application logic.

The theme is stored in the `theme.res` file in the project.

You load the theme file using this line of code in the `init(Object)` method in the main class of the application:

Listing 4. Theme Loading Code

```
theme = UIManager.initFirstTheme("/theme");
```

In a CSS project this file is generated automatically from the stylesheet. Legacy applications that still edit the resource file by hand can continue to do so, but new projects should prefer the CSS workflow described below.

This code is shorthand for resource file loading and for the installation of theme. You could technically have more than one theme in a resource file at which point you could use `initNamedTheme()` instead. The resource file is a special file format that includes inside it many features:

- Themes
- Images
- Localization Bundles
- Data files

3.2.2. Working with CSS themes

Modern Codename One projects ship with a `src/main/css/theme.css` file (or an equivalent stylesheet). Editing this file allows you to define UIIDs using standard CSS syntax together with Codename One–specific extensions such as `cn1-derive` for inheritance and the `#Constants` block for theme constants. Each time you build or run the project, the build tool compiles the CSS into the `theme.res` resource file automatically. Saving the CSS while the simulator is running will also trigger a refresh so you can iterate on styling.

Because the CSS compiler produces the final resource file, you should treat the generated `theme.res` as an output artifact and keep your changes in the CSS source. Images referenced from CSS rules (for example: background images or multi-images) should be placed alongside the stylesheet so that they're picked up by the compiler. More details about the supported selectors and properties are covered in the dedicated CSS chapter later in this guide.

3.3. GUI builder

The GUI builder allows you to arrange components visually within a UI using drag, property sheets etc. With the GUI builder you can create elaborate, rich UI's without writing the layout code. Forms designed in the builder use auto layout mode by default, which lets you position and resize components on a canvas while using `LayeredLayout` behind the scenes.

3.3.1. Hello world

The GUI builder is invoked through the Codename One Maven plugin and works the same way regardless of which IDE you use to edit the surrounding Java code. Two goals to know about:

- `cn1:create-gui-form` generates a new GUI form (the `.gui` XML file plus the matching Java source).
- `cn1:guibuilder` opens the GUI builder so you can edit an existing form.

Both goals accept a `className` parameter that points at the fully qualified class name of the form. From the root of a multi-module Codename One project, create a new form like this:

```
mvn cn1:create-gui-form -DclassName=com.example.MyForm
```

This generates two files inside the `common` module:

1. `common/src/main/guibuilder/com/example/MyForm.gui`
2. `common/src/main/java/com/example/MyForm.java`



If you run the goal from inside the `common` submodule the files are written directly under that module's `src/main/⋯` directory. By default the goal creates a `Form`; pass `-DguiType=Dialog` or `-DguiType=Container` to generate one of the other types, and pass `-DautoLayout=false` to opt out of auto layout mode.

To open the form in the GUI builder:

```
mvn cn1:guibuilder -DclassName=com.example.MyForm
```

The full goal reference, including every parameter, lives in Section B.7, “Create GUI Form (`create-gui-form`)” and Section B.15, “GUIBuilder goal (`guibuilder`)” in the Maven goals appendix.

Basic usage

Notice that the UI of the GUIBuilder might change in various ways but the basic concepts should remain the same.

The GUI builder is controlled through its main toolbar, notice that your changes will be applied when you click the Save button on the right:

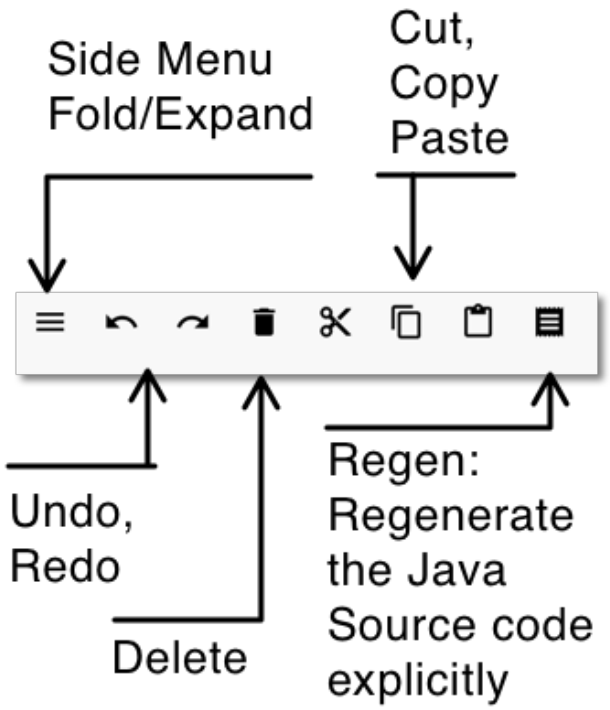


Figure 45. The features of the left toolbar

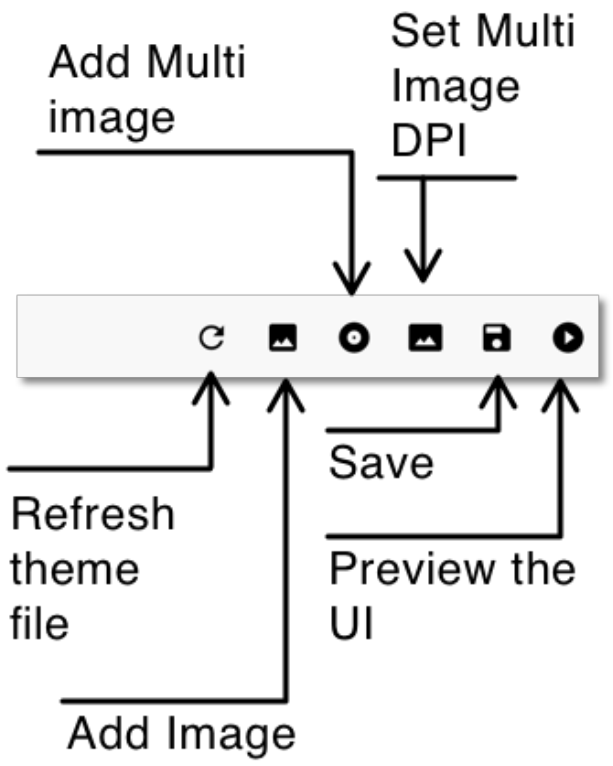


Figure 46. The features of the right toolbar

The main UI includes four important parts:

- Main Form — This is where you place the components of the UI you're building
- Component Tree — This is a logical representation of the component hierarchy within the Main Form. It's often easier to pick a component from the tree rather than the form itself
- Property Inspector — When you select an element in the tree or form you can see its details here. You can then edit the various details of the component in this area
- Palette — Components can be dragged from the palette to the Main Form and placed in the UI

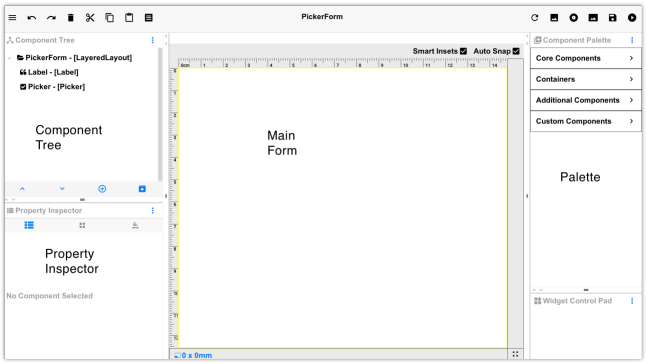


Figure 47. The four parts of the GUI builder

You will start by selecting the Component Palette and dragging a button into the UI:

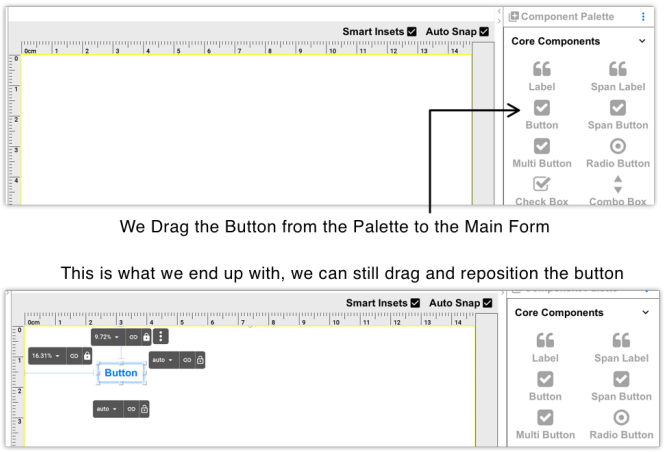


Figure 48. You can drag any component you want from the palette to the main UI

By default the auto layout mode of the GUI builder uses layered layout to position components. Sides can be bound to a component or to the `Form`. You then use distance units to determine the binding behavior. The GUI builder tries to be "smart" and guesses your intention as you drag the components along.

When you select the component you placed you can edit the properties of that component:

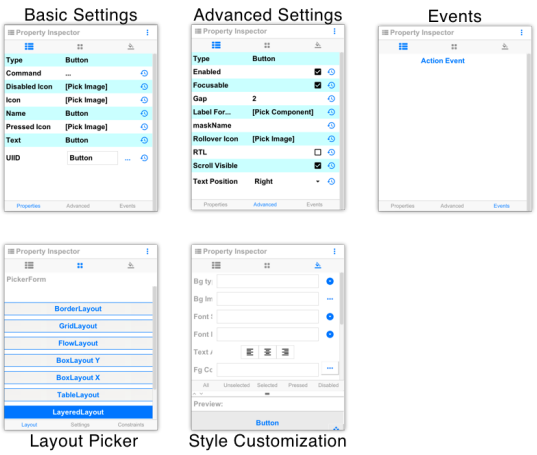


Figure 49. Properties allow you to customize everything about a component

There are five property sheets per component:

- Basic Settings — These include the basic configuration for a component for example: name, icon, text etc.
- Advanced Settings — These include features that aren't as common such as icon gap, mask etc.
- Events — By clicking a button in this tab a method will be added to the source file with a callback matching your component name. This binds an event to a button, text field etc.
- Layout — You can determine the layout of the parent `Container` here. For auto layout this should stay as layered layout, but you can nest other layout types in here
- Style Customization — This isn't a theme, if you want to customize the style of a specific component you can do that through this UI. The theme works on a more global/reusable level and this is designed for a specific component

For things like setting the text on the component you can use a convenient "long click" on the

component to edit the text in place as such:



Figure 50. Use the long click to edit the text "in place"

Events



As of now, the events tab was disabled. See issue 3593 for more info. [https://github.com/codenameone/CodenameOne/issues/3593#issuecomment-1133647486]

When a component supports broadcasting events you can bind such events by selecting it, then selecting the events tab and clicking the button matching the event type



Figure 51. The events tab is listed below supported event types can be bound above

Once an event is bound the IDE will open to the event code for example:

```
public void onButton_1ActionEvent(com.codename1.ui.events.ActionEvent ev) {
}
```



Some IDEs generate the project source code after you explicitly build the project so if your code needs to access variables etc. try building first

Within the code you can access all the GUI components you defined with the `gui_` prefix for example: `Button_1` from the UI is represented as:

```
private com.codename1.ui.Button gui_Button_1 = new com.codename1.ui.Button();
```

Underlying XML

Saving the project generates an XML file representing the UI into the res directory in the project, the GUI file is created in a matching hierarchy in the project under the `res/guibuilder` directory:

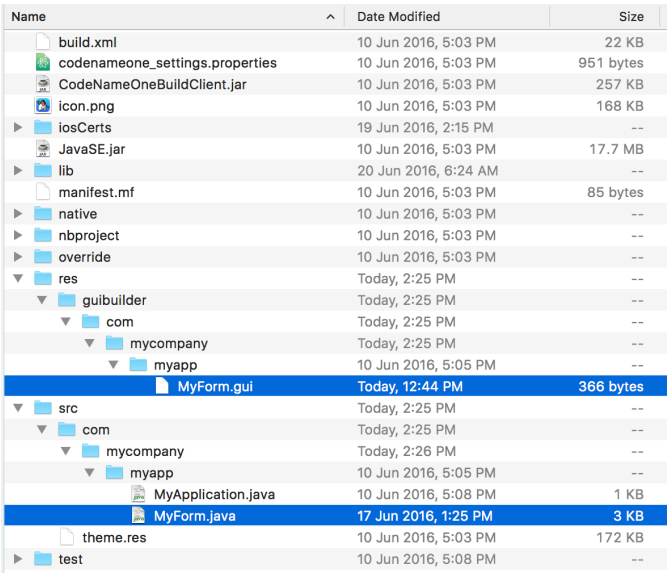


Figure 52. The java and GUI files in the hierarchy



If you refactor (rename or move) the Java file it's connection with the GUI file will break. You need to move/rename both

You can edit the GUI file directly but changes won't map into the GUI builder unless you reopen it. These files should be under version control as they're the main files that change. The GUI builder file for the button and label code looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>

<component type="Form" layout="FlowLayout" flowLayoutFillRows="false" flowLayoutAlign="1"
flowLayoutValign="0"  title="My new title" name="MyForm">
  <component type="Button" text="Button" name="Button_1" actionEvent="true">
  </component>
  <component type="Label" text="Hi World" name="Label_1">
  </component>
</component>
```

This format is simple. The file triggers the following Java source file:

```
/**
 * GUI builder created Form
 *
```

```
 * @author shai
 */
public class MyForm extends com.codename1.ui.Form {

    public MyForm() {
        this(com.codename1.ui.util.Resources.getGlobalResources());
    }

    public MyForm(com.codename1.ui.util.Resources resourceObjectInstance) {
        initGuiBuilderComponents(resourceObjectInstance);
    }

//-- DON'T EDIT BELOW THIS LINE!!!
    private com.codename1.ui.Label gui_Label_1 = new com.codename1.ui.Label();
    private com.codename1.ui.Button gui_Button_1 = new com.codename1.ui.Button();


// <editor-fold defaultstate="collapsed" desc="Generated Code">
    private void guiBuilderBindComponentListeners() {
        EventCallbackClass callback = new EventCallbackClass();
        gui_Button_1.addActionListener(callback);
    }

    class EventCallbackClass implements com.codename1.ui.events.ActionListener, com.codename1.ui.events
.DataChangedListener {
        private com.codename1.ui.Component cmp;
        public EventCallbackClass(com.codename1.ui.Component cmp) {
            this.cmp = cmp;
        }

        public EventCallbackClass() {
        }

        public void actionPerformed(com.codename1.ui.events.ActionEvent ev) {
            com.codename1.ui.Component sourceComponent = ev.getComponent();
            if(sourceComponent.getParent().getLeadParent() != null) {
                sourceComponent = sourceComponent.getParent().getLeadParent();
            }

            if(sourceComponent == gui_Button_1) {
                onButton_1ActionEvent(ev);
            }
        }

        public void dataChanged(int type, int index) {
        }
    }
    private void initGuiBuilderComponents(com.codename1.ui.util.Resources resourceObjectInstance) {
        guiBuilderBindComponentListeners();
        setLayout(new com.codename1.ui.layouts.FlowLayout());
        setTitle("My new title");
        setName("MyForm");
        addComponent(gui_Label_1);
        addComponent(gui_Button_1);
        gui_Label_1.setText("Hi World");
        gui_Label_1.setName("Label_1");
        gui_Button_1.setText("Click Me");
        gui_Button_1.setName("Button_1");
    }// </editor-fold>

//-- DON'T EDIT ABOVE THIS LINE!!!
    public void onButton_1ActionEvent(com.codename1.ui.events.ActionEvent ev) {
    }

}
```



Don’t touch the code within the don’t EDIT comments…

The GUI builder uses the "magic comments" approach where code is generated into those areas to match the XML defined in the GUI builder. Various IDEs generate that code at different times. Some will generate it when you run the app while others will generate it as you save the GUI in the builder.

You can write code within the class both by using the event mechanism, by writing code in the constructors or through overriding functionality in the base class.

3.3.2. Auto layout mode

New forms created with the GUI Builder use auto layout mode. In this mode you can move and resize your components as you see fit. You aren’t constrained to the positions dictated by the form’s layout manager.



All forms designed in auto layout mode use `LayeredLayout`

Auto layout mode is built upon the inset support in LayeredLayout. Component positioning uses insets and reference components, not absolute positioning

As an example, drag a button onto a blank form and see what happens. The button will be "selected" initially after adding it, so you’ll see its outline and resize handles for adjusting its size and position. You’ll also see four floating labels (above, below, to the left, and to the right) that show the corresponding side’s inset values and allow you to adjust them.

When a component is selected you can drag it to reposition it or use the resize handles to change its size. The floating inset labels update as you drag so you can fine tune spacing without leaving the canvas.

Press the mouse inside the bounds of the button and drag it around to reposition it. You will notice that the inset labels change to reflect the new inset values. If you drag the button close to the edge of the form, the corresponding inset value will change to millimetres. If you move farther away from the edge, it will change to percentage values.

The inset control

Take a closer look at the inset control (the inset controls are the black buttons that appear to the top, bottom, left, and right of the selected component).



Figure 53. The inset control allows you to change the inset size and units, toggle it between fixed and flexible, and link it to another component.

Each control has three sections:

1. **The inset value drop-down menu**. This shows the current value of the inset (for example: 0mm, 25%, auto, etc.). If you click on this, it will open a menu that will allow you to change the units. If the inset is in millimetres, it will have options for pixels, and percent. If the inset is in percent, it will have options for pixels and millimetres. Etc. It also includes a text field to enter

an inset value explicitly.



2. **The "Link" Button**  - If the inset is linked to a reference component, then this button will be highlighted "blue," and hovering over it will highlight the reference component in the UI so that you can see which component it's linked to. Clicking on this button will open a dialog that will allow you to "break" this link. You can drag this button over any component in the form to "link."
3. **The "Lock" Button"**  - This button allows you to toggle the inset between "flexible" (that's: auto) and "fixed" (that's: millimetres or percent).

Auto snap

Notice the "autosnap" checkbox that appears in the top-right corner of the GUI builder window.

Figure 54. Autosnap checkbox

Autosnap does what it sounds like: It automatically snaps two components together when you drag them near each other. This is handy for linking components together without having to explicitly link them (using the "link" button). This feature is turned on by default. If autosnap is turned off, you can still start a "snap" by holding down the ALT/Option key on your keyboard during the drag.

Smart insets

Beside the "autosnap" checkbox is another checkbox named `Smart Insets`.

Figure 55. Smart insets checkbox

Smart Inset uses some heuristics during a drag to try to determine how the insets should be linked. The heuristics are basic (it tries to link to the nearest neighbor component usually), but you will be working on improving this for future releases. This feature is turned off by default while it's still being refined. The goal is to improve this to the point where it **always** makes the correct link choices - at which time you will be able to use the GUI builder without having any knowledge of insets or reference components.

The widget control pad

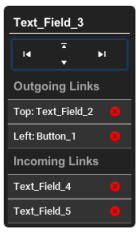


Figure 56. Widget control pad

When a component is selected, you should see a black floating panel appear in the lower right of the screen.

This is the widget control pad, and it provides an alternative view of the component's links. It also provides a useful list of incoming links (that's: components that "depend on" this component's positioning). Sometimes, you may want to disconnect incoming links so that you can drag the component without affecting the position of dependent components.

This control pad also includes game-pad-like controls (up, down, left, right), that allow you to "tab" the component to the next guide in that direction. Tab positions exist at component edges in the form. This is useful for aligning components with each other.

Keyboard Short-Cuts

1. **Arrow Keys** - Use the up/down/left/right arrow keys to nudge the selected component a little bit at a time. This is a convenient way to move the component to a position that's more precise than can be achieved with a mouse drag.
2. **Arrow Keys** + **SHIFT** - Hold down the SHIFT key while pressing an arrow key and it will "tab" the component to the next tab marker. The form has implicit tab markers at the edge of each component on the form.
3. **ALT/Option Key** + **Click or Drag** - Holding down the option/alt key while clicking or dragging a component will resulting in "snapping" behaviour even if autosnap is turned off.

Sub-Containers

Sometimes, you may need to add sub-containers to your form to aid in grouping your components together. You can drag a container onto your form using the "Container" palette item (under "Core Components"). The default layout the subcontainer will be LayeredLayout so that you're able to position components within the sub-container with precision, like on the root container.

You can also change the layout of subcontainers to another classical layout manager (for example: grid layout, box layout, etc.) and drag components directly into it. This is useful if parts of your form lend themselves to a different layout. As an example, drag a container onto the canvas that uses BoxLayout Y. (You can find this under the "Containers" section of the component palette).

Drag the button (that was earlier on the form) over that container, and you should see a drop-zone become highlighted.



Figure 57. Dropping container on child container with box layout y

You can drop the button directly there. You can As you drag more components into the sub-container, you'll see them automatically laid out vertically.

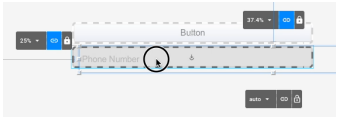


Figure 58. Box Layout Y dropping 2nd child

The canvas resize tool

When designing a UI with the GUI builder it's ** important that you periodically test the form's "resizing" behavior so that you know how it will behave on different devices. Components may appear to be positioned when the canvas is one size, but become out of whack when the container is resized. After every manipulation you perform, it's good practice to drag the canvas resize tool (the button in the lower right corner of the GUI builder) smaller and bigger so you can see how the positions are changed. If things grow out of whack, you may need to toggle an inset between fixed and auto, or add a link between some components so that the resizing behavior matches your expectations.

Chapter 4. Advanced theming

Before getting into CSS, this section introduces a few advanced theme concepts. Those ideas also apply to CSS because CSS uses features such as theme constants.

4.1. Working with UIIDs

A UIID (User Interface Identifier) uniquely identifies a group of theme definitions for UI components. For example, a `Button` UIID can be linked to a component, and the theme can then define that component's appearance.

A major advantage of UIIDs is the ability to change a component's UIID. For example, to create a multiline label, use code like this:

```
TextArea t = new TextArea(); // ...
t.setUIID("Label");
t.setEditable(false);
```



Components such as SpanLabel [https://www.codenameone.com/javadoc/com/codename1/components/SpanLabel.html] use this approach internally.

UIIDs can be customized in the GUI builder and support per-component styling.



The component class name often matches the UIID, but the two remain separate entities.

4.2. Theme layering

Theme layering helps in two common cases:

- A platform needs a slightly different theme.
- A specific use case needs theme customization. For example, a user might select larger fonts.

Theme layering is straightforward and doesn't require rebuilding the entire theme. A theme can be applied on top of another theme, like CSS cascading. To create a layer, add a new theme with the Add Theme button.



Remove the `includeNativeBool` constant from the new theme.

In the new theme, define the changes you need. For example, to use a larger default font, define that property for the relevant UIIDs and leave the other properties unchanged.

In a non-GUI-builder app, the theme loading code looks like this by default:

```
Resources theme = UIManager.initFirstTheme("/theme");
```

Update it to look like this:

```
Resources theme = UIManager.initNamedTheme("/theme", "Theme");
```



This example assumes that the main theme is named `Theme`, not the layer theme that was added.

The original code relies on the theme being at position 0 in the theme name array. That might not always be true.

To add the theme layer, use:

```
UIManager.getInstance().addThemeProps(theme.getTheme("NameOfLayerTheme"));
```

The `addThemeProps` call places the secondary theme on top of the primary "Theme" and keeps the original UIIDs in "Theme" intact.

To apply theme changes to a running application, call `Form`s `refreshTheme()`. This updates the UI and provides visual feedback for the theme changes.

4.3. Override resources in platform

When you adapt an application to different OS conventions, one common need is to use different icons. Sometimes you also need to change behavior based on device type, for example to use a different UI structure for a tablet.

Codename One lets you override a resource for a specific platform. You can redefine a resource for that platform and add platform-specific resources.

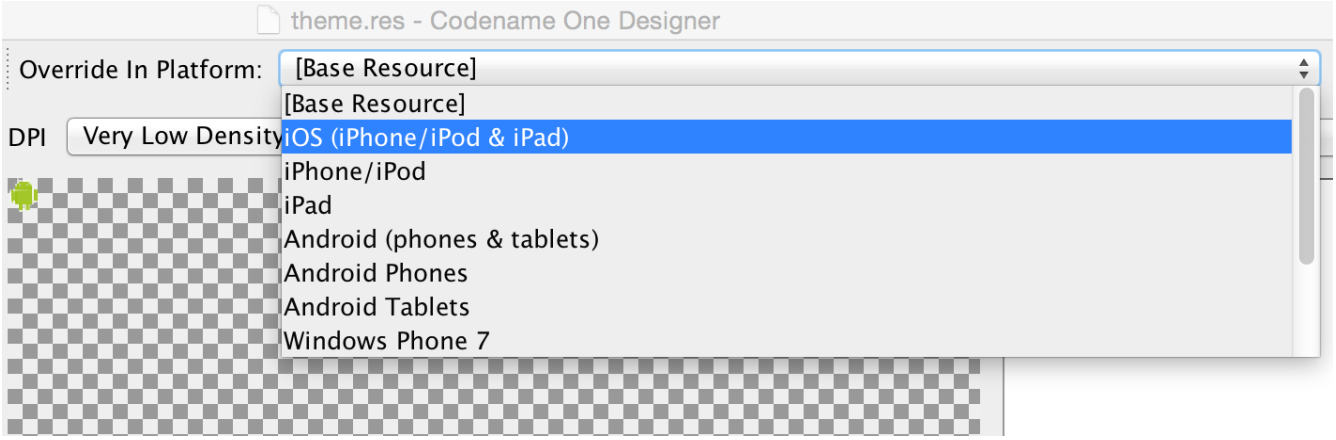


Figure 59. Override resources for specific platform

`Overridden` resources take precedence over embedded resources, which lets you change the look or even behavior for a specific platform or OS.



Overriding the theme is dangerous because a theme has external dependencies, such as image borders. The solution is to use theme layering and override the

layer.

To override a resource, select the platform where the override applies.

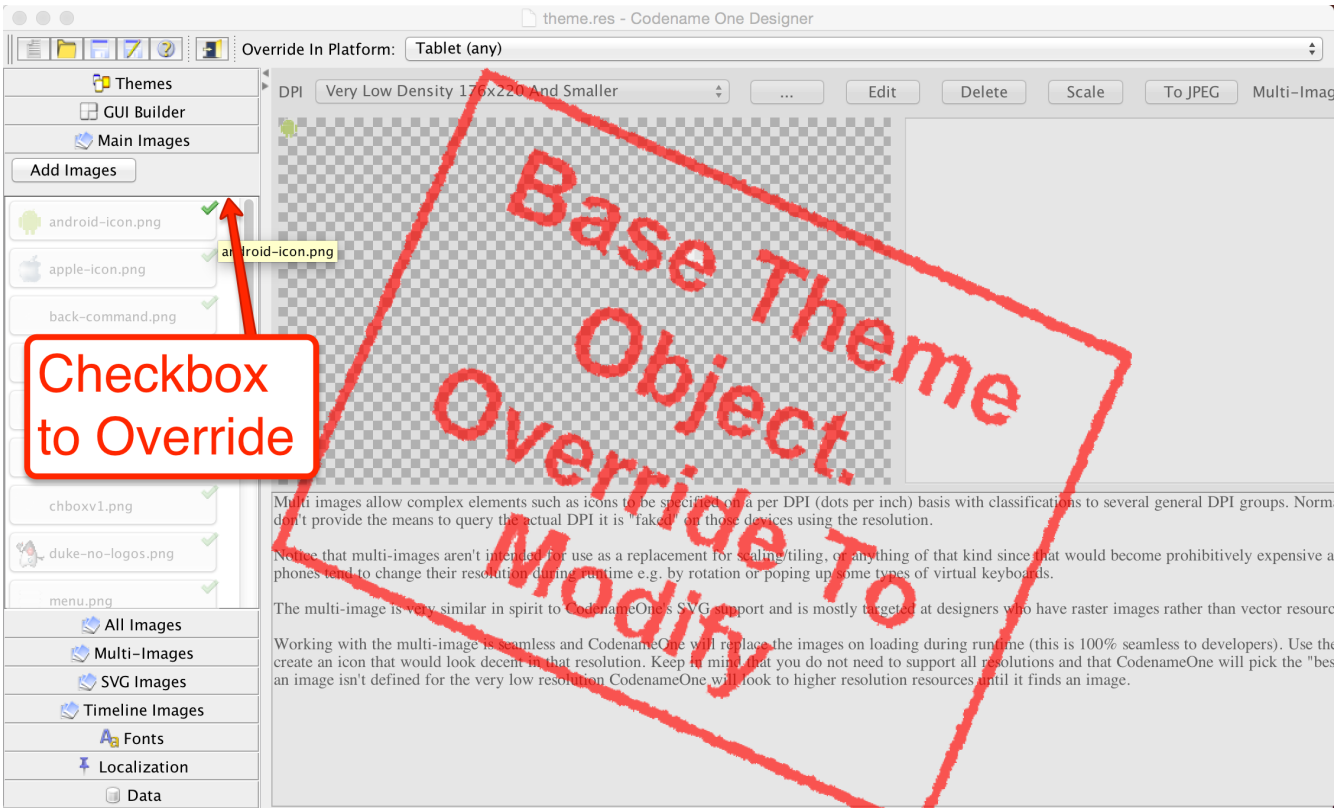


Figure 60. Override for platform, allows you to override the checked resources and replace them with another resource

You can then click the green checkbox to mark that resource as specific to this platform. All resources added while that platform is selected apply to that platform. If you change your mind, delete the override in override mode and it will stop applying.

4.4. Theme constants

The Codename One Designer has a tab for creating constants. These constants can add global values and behavior hints to Codename One and its components. Constants are always strings. Some conventions let the UI adapt to input types. For example, a constant that ends with `Bool` is treated as a boolean value and displays as a checkbox. An `Int` suffix displays a numeric picker, and an `Image` suffix shows a combo box for picking an image.



The combo box in the designer for adding a theme constant is editable, so you can type any value you want.

To use a constant, call the appropriate UIManager [https://www.codenameone.com/javadoc/com/codename1/ui/plaf/UIManager.html] methods:

- `getThemeConstant`
- `isThemeConstant`
- `getThemeImageConstant`

Internally, Codename One has many built-in constants, and the list keeps growing. As features are added, the list is updated, but theme constants are inherently ad hoc and some may not appear here.

Table 3. Theme Constants

Constant	Description/Argument
alwaysTensileBool	Enables tensile drag even when there is no scrolling in the container (only for scrollable containers)
backGestureThresholdInt	The threshold for the back gesture in the SwipeBackSupport [https://www.codenameone.com/javadoc/com/codename1/ui/util/SwipeBackSupport.html] class, defaults to 5
backUsesTitleBool	Indicates to the GUI builder that the back command should use the title of the previous form and not just the word "Back"
buttonRippleBool	`true` to activate the material design ripple effect on the buttons. This effect draws a growing circle from the point of touch onwards. This is `false` by default except for Android where it defaults to `true`. This is equivalent to `setButtonRippleEffectDefault(bool)` in the `Button` class
capsButtonTextBool	`true` to activate the caps text mode in the buttons. When activated `setText` on `Button` and all the constructors will invoke `upcase()` on all the strings effectively making the application buttons use uppercase exclusively. This is `false` by default except for Android where it defaults to `true`. It's equivalent to `Button.setCapsTextDefault(boolean)` and can be tuned to an individual `Component` via `Component.setRippleEffect(boolean)`
capsButtonUiids	A list of the UIID's that should be capitalized by default (in supported platforms) other than the `Button` and `RaisedButton` which are already capitalized. This list can be separated by spaces or commas for example, `capsButtonUiids=UpcaseButton,OtherCustomButton`
defaultCommandImage	Image to give a command with no icon
dialogButtonCommandsBool	Place commands in the dialogs as buttons
dialogBlurRadiusInt	Sets the default Gaussian blur radius for the background of the dialogs. The default value is -1 indicating no blur

Constant	Description/Argument
dialogPosition	Place the dialog in an arbitrary border layout position (for example, North, South, Center, etc.)
centeredPopupBool	Popup of the combo box will appear in the center of the screen
changeTabOnFocusBool	Useful for feature phones, allows changing the tab when the focus changes, without pressing a key
checkBoxCheckDisImage	CheckBox image to use instead of Codename One drawing it on its own
checkBoxCheckedImage	CheckBox image to use instead of Codename One drawing it on its own
checkBoxOppositeSideBool	Indicates the check box should be drawn on the opposite side to the text and not next to the text
checkBoxUncheckDisImage	CheckBox image to use instead of Codename One drawing it on its own
checkBoxUncheckedImage	CheckBox image to use instead of Codename One drawing it on its own
comboImage	Combo image to use instead of Codename One drawing it on its own
commandBehavior	Deprecated: don't use this constant as it conflicts with the `Toolbar`. Indicates how commands should act, as a touch menu, native menu etc. Possible values: SoftKey, Touch, Bar, Title, Right, Native
ComponentGroupBool	Enables component group, which allows components to be logically grouped together, so the UIID's of components would be modified based on their group placement. This allows for some unique styling effects where the first/last elements have different styles from the rest of the elements. It's disabled by default, thus leaving its usage up to the designer
dialogTransitionIn	Default transition for dialog
dialogTransitionInImage	Default transition Image [https://www.codenameone.com/javadoc/com/codename1/ui/Image.html] for dialog, causes a Timeline [https://www.codenameone.com/javadoc/com/codename1/ui/animations/Timeline.html] transition effect
dialogTransitionOut	Default transition for dialog

Constant	Description/Argument
defaultCommandImage	An image to place on a command if none is defined, only applies to touch commands
defaultEmblemImage	The emblem painted on the side of the multibutton, by default this is an arrow on some platforms
emblemUiid	Overrides the UIID applied to emblem labels rendered by composite buttons such as `MultiButton` and `SpanMultiButton`.
iconUiid	Allows theming the icon component used by `SpanLabel`, `SpanButton`, `MultiButton`, and `SpanMultiButton` when they fetch icon styling via theme constants.
iosScrollMotionBool	Enables iOS-style scroll physics (nonlinear rubber-band during drag and critically damped tensile snap-back). When unset it defaults to `true` on the iOS platform and `false` elsewhere. Set explicitly to override the platform default. See Section 4.4.1, “Scroll motion tuning”
imageviewerNavigationArrowsBool	Enables side navigation arrows by default for `ImageViewer` instances using the `ImageViewer` UIID (or a lowercased custom UIID prefix).
imageviewerThumbnailBarHeightMM	Sets the default thumbnail strip height (in millimeters) for `ImageViewer` navigation thumbnails.
imageviewerThumbnailsBool	Enables the bottom thumbnail strip by default for `ImageViewer` instances using the `ImageViewer` UIID (or a lowercased custom UIID prefix).
textUiid	Overrides the text component UIID for `SpanLabel`, `SpanButton`, `MultiButton`, and `SpanMultiButton` instances when provided as a theme constant.
useLargerTextScaleBool	When `true`, UIManager applies the larger text accessibility scale from `Display.getLargerTextScale()` when `Display.isLargerTextEnabled()` is enabled.
dialogTransitionOutImage	Default transition Image [https://www.codenameone.com/javadoc/com/codename1/ui/Image.html] for dialog, causes a Timeline [https://www.codenameone.com/javadoc/com/codename1/ui/animations/Timeline.html] transition effect

Constant	Description/Argument
DecayMotionScaleFactorInt	Velocity-to-distance multiplier for the exponential-decay scroll motion (see Section 4.4.1, “Scroll motion tuning”). Default is `950`. Larger values make flicks travel farther for the same release velocity
disabledColor	Color to use when disabling entries by default
dlgButtonCommandUIID	The UIID used for dialog button commands
dlgCommandButtonSizeInt	Minimum size to give to command buttons in the dialog
dlgCommandGridBool	Places the dialog commands in a grid for uniform sizes
dlgInvisibleButtons	Includes an RRGGBB color for the line separating dialog buttons, as is the case with Android 4 and iOS 7 buttons in dialogs
dlgSlideDirection	Slide hints
dlgSlideInDirBool	Slide hints
dlgSlideOutDirBool	Slide hints
drawMapPointerBool	Indicates whether a pointer should appear in the center of the map component
fadeScrollBarBool	Boolean indicating if the scrollbar should fade when there is inactivity
setFadeScrollBarSpeed(int) *(API)*	Sets the per-frame opacity decrement for scrollbar fading (not milliseconds). Approximate fade duration is `durationMs ~= (255 / speed) * (1000 / frameRate)`. At 60 FPS: speed `5` ~= `850ms`, `3` ~= `1400ms`, `2` ~= `2100ms`, `1` ~= `4250ms`
fadeScrollEdgeBool	Places a fade effect at the edges of the screen to show that it’s possible to scroll until you reach the edge (common on Android)
fadeScrollEdgeInt	Amount of pixels to fade out at the edge
firstCharRTLBool	Indicates to the GenericListCellRenderer [https://www.codenameone.com/javadoc/com/codename1/ui/list/GenericListCellRenderer.html] that it should determine RTL status based on the first character in the sentence
noTextModeBool	Indicates that the on/off switch in iOS shouldn’t draw text on top of the switch, which is the case for iOS 7+ but not for prior versions

Constant	Description/Argument
fixedSelectionInt	Number corresponding to the fixed selection constants in List [https://www.codenameone.com/javadoc/com/codename1/ui/List.html]
formTransitionIn	Default transition for form
formTransitionInImage	Default transition Image [https://www.codenameone.com/javadoc/com/codename1/ui/Image.html] for form, causes a Timeline [https://www.codenameone.com/javadoc/com/codename1/ui/animations/Timeline.html] transition effect
formTransitionOut	Default transition for form
formTransitionOutImage	Default transition Image [https://www.codenameone.com/javadoc/com/codename1/ui/Image.html] for form, causes a Timeline [https://www.codenameone.com/javadoc/com/codename1/ui/animations/Timeline.html] transition effect
globalToobarBool	Indicates that the Toolbar API should be on/off by default for all forms
hasRaisedButtonBool	Is true in platforms where the theme has the `RaisedButton` UIID defined. This is only true in the native Android theme to allow some material design guidelines
hideBackCommandBool	Hides the back command from the side menu when possible
hideEmptyTitleBool	Indicates that a title with no content should be hidden even if the border for the title occupies space
hideLeftSideMenuBool	Hides the side menu icon that appears on the left side of the UI
hideRightSideMenuBool	Hides the toolbar menu icon that appears on the right side of the title area
ignorListFocusBool	Hide the focus component of the list when the list doesn't have focus
infiniteImage	The image used by the infinite progress component, the component will rotate it as needed
infiniteMaterialDesignSize	Size (in millimeters) of the generated material design spinner when `InfiniteProgress` runs in material mode. Defaults to 6.667mm

Constant	Description/Argument
infiniteMaterialImageSize	Size (in millimeters) of the generated spinner image when no custom `infiniteImage` is provided. Defaults to 7mm
infiniteDefaultColor	Hex RGB color used for the autogenerated material spinner when `infiniteImage` isn't supplied. Defaults to `777777`
interactionDialogSpeedInt	Controls the duration in milliseconds that `InteractionDialog` uses when animating into and out of the layered pane (including the directional dispose helpers). Defaults to `400` if the constant isn't defined.
includeNativeBool	True to derive from the platform native theme, false to create a blank theme that only uses the basic defaults
labelGap	Positive floating point value representing the default gap value between the label text and the icon in millimeters
listItemGapInt	Built-in item gap in the list, this defaults to 2, which predated padding/margin in Codename One
listLongPressBool	Indicates whether a list should handle long press events, defaults to true
mapTileLoadingImage	An image to preview while loading the MapComponent [https://www.codenameone.com/javadoc/com/codename1/maps/MapComponent.html] tile
mapTileLoadingText	The text of the tiles in the MapComponent [https://www.codenameone.com/javadoc/com/codename1/maps/MapComponent.html] during loading, defaults to "Loading..."
mapZoomButtonsBool	Indicates whether buttons should be drawn on the map component
mediaBackImage	Media icon used by the media player class
mediaFwdImage	Media icon used by the media player class
mediaPauseImage	Media icon used by the media player class
mediaPlayImage	Media icon used by the media player class
menuButtonBottomBool	When set to true this flag aligns the menu button to the bottom portion of the title. Defaults to false

Constant	Description/Argument
menuButtonTopBool	When set to true this flag aligns the menu button to the top portion of the title. Defaults to false
landscapeTitleUiidBool	When true the toolbar swaps to landscape-specific UIIDs (such as `ToolbarLandscape` /`BackCommandLandscape`) for layout and icon styling
menuHeightPercent	Allows positioning and sizing the menu
menuImage	The three dot menu image used in Android and the Toolbar [https://www.codenameone.com/javadoc/com/codename1/ui/Toolbar.html] to show more command entries
menuImageSize	The size in millimeters (floating point value) of the generated side menu image, this is only used if you don't supply a custom image. The default value is 4.5.
menuPrefSizeBool	Allows positioning and sizing the menu
menuSlideDirection	Defines menu entrance effect
menuSlideInDirBool	Defines menu entrance effect
menuSlideOutDirBool	Defines menu entrance effect
menuTransitionIn	Defines menu entrance effect
menuTransitionInImage	Defines menu entrance effect
menuTransitionOut	Defines menu exit effect
menuTransitionOutImage	Defines menu entrance effect
menuWidthPercent	Allows positioning and sizing the menu
iosStyleBackArrowBool	Renders the back command using the iOS-style chevron arrow (with a small icon gap) when the toolbar shows back text
minimizeOnBackBool	Indicates whether the form should minimize the entire application when the physical back button is pressed (if available) and no command is defined as the back command. Defaults to true
onOffIOSModeBool	Indicates whether the on/off switch should use the iOS or Android mode
otherPopupRendererBool	Indicates that a separate renderer UIID/instance should be used to the list within the combo box popup

Constant	Description/Argument
PackTouchMenuBool	Enables preferred sized packing of the touch menu (true by default), when set to false this allows manually determining the touch menu size using percentages
paintsTitleBarBool	Indicates that the StatusBar UIID should be added to the top of the form to space down the title area, as is the case on iOS 7+ where the status bar is painted on top of the UI
rightSideMenuImage	Overrides the default icon used for the right-side toolbar menu button
rightSideMenuPressImage	Optional pressed-state icon for the right-side toolbar menu button
popupCancelBodyBool	Indicates that a cancel button should appear within the combo box popup
PopupDialogArrowBool	Indicates whether the popup dialog has an arrow, notice that this constant will change if you change UIID of the popup dialog
PopupDialogArrowBottomImage	Image of the popup dialog arrow, notice that this constant will change if you change UIID of the popup dialog
PopupDialogArrowTopImage	Image of the popup dialog arrow, notice that this constant will change if you change UIID of the popup dialog
PopupDialogArrowLeftImage	Image of the popup dialog arrow, notice that this constant will change if you change UIID of the popup dialog
PopupDialogArrowRightImage	Image of the popup dialog arrow, notice that this constant will change if you change UIID of the popup dialog
popupNoTitleAddPaddingInt	Adds padding to a popup when no title is present
popupTitleBool	Indicates that a title should appear within the combo box popup
pullToRefreshImage	The arrow image used to draw the `pullToRefresh` animation
pureTouchBool	Indicates the pure touch mode
radioOppositeSideBool	Indicates the radio button should be drawn on the opposite side to the text and not next to the text
radioSelectedDisImage	Radio button image
radioSelectedImage	Radio button image

Constant	Description/Argument
radioUnselectedDisImage	Radio button image
radioUnselectedImage	Radio button image
radioSelectedDisFocusImage	Radio button image
radioSelectedFocusImage	Radio button image
radioUnselectedDisFocusImage	Radio button image
radioUnselectedFocusImage	Radio button image
releaseRadiusInt	Indicates the distance from the button with dragging, in which the button should be released, defaults to 0
rendererShowsNumbersBool	Indicates whether renderers should render the entry number
reverseSoftButtonsBool	Swaps the softbutton positions
rightSideMenuImage	Same as sideMenuImage only for the right side, optional and defaults to sideMenuImage
rightSideMenuPressImage	Same as sideMenuPressImage only for the right side, optional and defaults to sideMenuPressImage
rubberBandCoefficientInt	Resistance coefficient for the iOS-style rubber-band compression used while the finger drags past a scroll edge. Value is interpreted as hundredths, so `55` means `0.55` (the iOS reference value). Larger values let the content be pulled farther for the same finger travel; smaller values make the content feel stiffer. Only active when `iosScrollMotionBool` is `true`. See Section 4.4.1, “Scroll motion tuning”
ScrollMotion	Selects the post-release flick model. `DECAY` (default) uses exponential velocity decay controlled by `ScrollMotionTimeConstantInt` and `DecayMotionScaleFactorInt`. Any other value (for example, `FRICTION`) falls back to the linear-friction model. See Section 4.4.1, “Scroll motion tuning”
ScrollMotionTimeConstantInt	Time constant in milliseconds for the exponential-decay flick model (active when `ScrollMotion=DECAY`). Default is `500`. Smaller values stop the flick sooner
scrollVisibleBool	`true`/`false` default is platform dependent. Toggles whether the scroll bar is visible

Constant	Description/Argument
showBackCommandOnTitleBool	Used by the Toolbar [https://www.codenameone.com/javadoc/com/codename1/ui/Toolbar.html] API to show whether the back button should appear on the title
shrinkPopupTitleBool	Indicates the title of the popup should be set to 0 if it's missing
sideMenuAnimSpeedInt	The speed at which a sidemenu moves defaults to 300 milliseconds
sideMenuFoldedSwipeBool	Indicates the side menu could be opened via swiping
sideMenuImage	The image representing the side menu, three lines (Hamburger menu)
sideMenuPressImage	Optional pressed version of the sideMenuImage
sideMenuScrollVisibleBool	Indicates whether the scroll bar on the side menu should be visible or not, defaults to hidden
sideMenuShadowBool	Indicates whether the shadow for the side menu should be drawn
sideMenuShadowImage	The image used when drawing the shadow (a default is used if this isn't supplied)
sideMenuSizeTabPortraitInt	The size of the side menu when expanded in a tablet in portrait mode
sideMenuSizePortraitInt	The size of the side menu when expanded in a phone in portrait mode
sideMenuSizeTabLandscapeInt	The size of the side menu when expanded in a tablet in landscape mode
sideMenuSizeLandscapeInt	The size of the side menu when expanded in a phone in landscape mode
sideMenuTensileDragBool	Enables/disables the tensile drag behavior within the opened side menu
sideSwipeActivationInt	Indicates the threshold in the side menu bar at which a swipe should trigger activation, defaults to 15 (percent)
sideSwipeSensitiveInt	Indicates the region of the screen that's sensitive to side swipe in the side menu bar, defaults to 10 (percent)
sigButtonOKUIID	Defines the UIID applied to the confirmation button in the signature capture dialog (defaults to `Button`).

Constant	Description/Argument
sigButtonResetUIID	Defines the UIID applied to the reset/clear button in the signature capture dialog (defaults to `Button`).
sigButtonCancelUIID	Defines the UIID applied to the cancel button in the signature capture dialog (defaults to `Button`).
slideDirection	Default slide transition settings
slideInDirBool	Default slide transition settings
slideOutDirBool	Default slide transition settings
sliderThumbImage	The thumb image that can appear on the sliders
snapGridBool	Snap to grid toggle
statusBarScrollsUpBool	Indicates that a tap on the status bar should scroll up the UI, only relevant in OSes where paintsTitleBarBool is true
switchButtonPadInt	Indicates the padding in the on/off switch, defaults to 16
switchMaskImage	Indicates the mask image used in iOS mode to draw on top of the switch
switchOnImage	Indicates the on image used in iOS mode to draw the on/off switch
switchOffImage	Indicates the off image used in iOS mode to draw the on/off switch
TabEnableAutoImageBool	Indicates images should be filled by default for tabs
TabSelectedImage	Default selected image for tabs (if TabEnableAutoImageBool=true)
TabUnselectedImage	Default unselected image for tabs (if TabEnableAutoImageBool=true)
tabPlacementInt	The placement of the tabs in the Tabs [https://www.codenameone.com/javadoc/com/codename1/ui/Tabs.html] component: TOP = 0, LEFT = 1, BOTTOM = 2, RIGHT = 3
tabsSlideSpeedInt	The time of the animation that occurs (in milliseconds) between releasing a swiped tab and reaching the next tab. Defaults to 200
tabsFillRowsBool	Indicates if the tabs should fill the row using flow layout
tabsGridBool	Indicates whether tabs should use a grid layout thus forcing all tabs to have identical sizes

Constant	Description/Argument
tabsOnTopBool	Indicates the tabs should be drawn on top of their content in a layered UI, this allows a tab to intrude into the content of the tabs
textCmpVAlignInt	The vertical alignment of the text component: TOP = 0, CENTER = 4, BOTTOM = 2
textComponentErrorColor	A hex RGB color which defaults to null in which case this has no effect. When defined this will change the color of the border and label to the given color to match the material design styling. This implements the red border underline in cases of error and the label text color change
textComponentOnTopBool	Toggles the on top mode which makes things look like they do on Android. This defaults to true on Android and false on other OSes. This can also be manipulated via the `onTopMode(boolean)` method in `InputComponent`, but the layout will only use the theme constant
textComponentAnimBool	toggles the animation mode which again can be manipulated by a method in `InputComponent`. If you want to keep the UI static without the floating hint effect set this to false. Notice this defaults to true only on Android
textComponentFieldUIID	sets the UIID of the text field to something other than `TextField` this is useful for platforms such as iOS where the look of the text field is different within the text component. This allows you to make the background of the text field transparent when it's within the `TextComponent` and make it different from the regular text field
textFieldCursorColorInt	The color of the cursor as an integer (not hex)
tickerSpeedInt	The speed of label/button etc. (in milliseconds)
tintColor	The aarrggbb hex color to tint the screen when a dialog is shown
topMenuSizeTabPortraitInt	The size of the side menu when expanded and attached to the top in a tablet in portrait mode
topMenuSizePortraitInt	The size of the side menu when expanded and attached to the top in a phone in portrait mode
topMenuSizeTabLandscapeInt	The size of the side menu when expanded and attached to the top in a tablet in landscape mode

Constant	Description/Argument
topMenuSizeLandscapeInt	The size of the side menu when expanded and attached to the top in a phone in landscape mode
touchCommandFillBool	Indicates how the touch menu should lay out the commands within
touchCommandFlowBool	Indicates how the touch menu should lay out the commands within
transitionSpeedInt	Indicates the default speed for transitions
treeFolderImage	Picture of a folder for the Tree [https://www.codenameone.com/javadoc/com/codename1/ui/tree/Tree.html] class
treeFolderOpenImage	Picture of a folder expanded for the `Tree` class
treeNodeImage	Picture of a file node for the `Tree` class
tensileDragBool	Indicates that tensile drag should be enabled/disabled. This is set by platform themes
tensileSnapMinDurationInt	Minimum duration in milliseconds for the tensile snap-back animation when the finger is released past a scroll edge. Defaults to `400` when `iosScrollMotionBool` is `true` and `300` otherwise. The actual duration scales with the distance being snapped; this is only a floor
tensileSnapMotion	Motion curve used for the tensile snap-back. `SPRING` (the default when `iosScrollMotionBool` is `true`) applies a critically damped spring envelope that settles softly. `DECELERATION` (the default elsewhere) applies the legacy quadratic ease-out. See Section 4.4.1, “Scroll motion tuning”

4.4.1. Scroll motion tuning

Scroll feel is controlled by three stages, each tunable via theme constants:

1. **While the finger is dragging past an edge**--when `iosScrollMotionBool` is `true`, the over-edge distance is compressed by the iOS UIScrollView rubber-band function `c * d * dim / (c * d + dim)`, where `d` is the raw finger displacement past the edge, `dim` is the viewport dimension, and `c` is `rubberBandCoefficientInt / 100` (default `0.55`). The result is that the content feels heavier the farther you pull it, asymptotically approaching `dim`. When `iosScrollMotionBool` is `false`, the legacy behavior is used: a linear 1:1 follow clamped to `tensileLength`.
2. **After the finger is lifted, while there is still velocity**--the `ScrollMotion` constant selects the model. `DECAY` (default) uses exponential velocity decay with a time constant of `ScrollMotionTimeConstantInt` milliseconds and a distance scale of `DecayMotionScaleFactorInt`.

Any other value (for example, `FRICTION`) falls back to linear friction.

3. **When the scroll comes to rest past an edge**--a tensile snap-back animates the content back to the edge. `tensileSnapMotion` picks the curve: `SPRING` is a critically damped envelope (iOS-like soft settle) and `DECELERATION` is the legacy quadratic ease-out. The duration scales with distance but never falls below `tensileSnapMinDurationInt` milliseconds.

   `iosScrollMotionBool` is the single switch that flips stage 1 and the default of stage 3 to the iOS-matching behavior. It defaults to `true` on the iOS platform only. Every individual constant can still be overridden regardless of this switch, so you can, for example, enable the nonlinear rubber-band globally (`iosScrollMotionBool=true`) while keeping the legacy quadratic snap-back (`tensileSnapMotion=DECELERATION`).

Tuning suggestions

- To make scrolling feel stiffer on iOS (less pull), reduce `rubberBandCoefficientInt` from `55` toward `35`.
- To make flicks stop sooner, reduce `ScrollMotionTimeConstantInt` (default `500`) or `DecayMotionScaleFactorInt` (default `950`).
- To enable iOS-style physics on Android or desktop for consistency, set `iosScrollMotionBool=true` explicitly.
- To disable iOS physics on the simulator while debugging, set `iosScrollMotionBool=false` in your development theme.

Dynamic Theme Swapping & Theme Constants

Once a theme constant is set by a theme, it isn't removed on a refresh when replacing the theme.

For example, if one would set the `comboImage` constant to a specific value in theme A and then switch to theme B, that doesn't define the `comboImage`, the original theme A `comboImage` might remain!

The reason for this is simple: when extracting the constant values, components keep the values in cache locally and just don't track the change in value. Furthermore, since the components allow manually setting values, it's impractical for them to track whether a value was set by a constant or explicitly by the user.

The solution for this is to either manually reset undesired values before replacing a theme (for example, for the case, above by calling the default look and feel method for setting the combo image with a null value), or defining a constant value to replace the existing value.

4.5. Native theming

Codename One uses a theme constant called `includeNativeBool`, when that constant is set to `true` Codename One starts by loading the native theme first and then applying all the theme settings. This means your theme "derives" the style of the native theme first, like the cascading effect of CSS. Internally this is what the theme layering section covered.

By avoiding this flag you can create themes that look *EXACTLY* the same on all platforms.



If you avoid the native theming you might be on your own. A few small device oddities such as the iOS status bar are abstracted by native theming. Without it you will need to do everything from scratch

You can simulate different OS platforms by using the native theme menu option

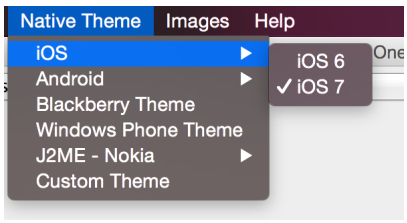


Figure 61. The native theme menu option

Developers can pick the platform of their liking and see how the theme will appear in that particular platform by selecting it and having the preview update on the fly.

4.6. Under the hood of the theme engine

To truly understand a theme you need to understand what it is. Internally a theme is a `Hashtable` key/value pair between UIID based keys and their respective values. For example: the key:

```
return "Button.fgColor=ffffff";
```

Will set the foreground color of the Button [https://www.codenameone.com/javadoc/com/codename1/ui/Button.html] UIID to white.

When a Codename One Component [https://www.codenameone.com/javadoc/com/codename1/ui/Component.html] is instantiated it requests a Style [https://www.codenameone.com/javadoc/com/codename1/ui/plaf/Style.html] object from the UIManager [https://www.codenameone.com/javadoc/com/codename1/ui/plaf/UIManager.html] class. The `Style` object is based on the settings within the theme and can be modified through code or by using the theme.

You can replace the theme dynamically in runtime and refresh the styles assigned to the various components using the refreshTheme() [https://www.codenameone.com/javadoc/com/codename1/ui/Component.html#refreshTheme--] method.



It's a common mistake to invoke `refreshTheme()` without actually changing the theme. You see developers doing it when all they need is a `repaint()` or `revalidate()`. Since `refreshTheme()` is ** expensive recommend that you don't use it unless you need to...

A theme `Hashtable` key comprises:

```
[UIID.][type#]attribute
```

The *UIID*, corresponds to the component's UIID for example: `Button`, CheckBox

[https://www.codenameone.com/javadoc/com/codename1/ui/CheckBox.html] etc. It's optional and may be omitted to address the global default style.

The type is omitted for the default unselected type, and may be one of *sel* (selected type), *dis* (disabled type) or *press* (pressed type). The attribute should be one of:

- `derive` - the value for this attribute should be a string representing the base component.
- `bgColor` - represents the background color for the component, if applicable, in a web hex string format RRGGBB for example: ff0000 for red.
- `fgColor` - represents the foreground color, if applicable.
- `border` - an instance of the border class, used to display the border for the component.
- `bgImage` - an Image [https://www.codenameone.com/javadoc/com/codename1/ui/Image.html] object used in the background of a component.
- `transparency` - a `String` containing a number between 0-255 representing the alpha value for the background. This applies to the bgColor.
- `margin` - the margin of the component as a `String` containing 4 comma separated numbers for top, bottom, left, right.
- `padding` - the padding of the component, it has an identical format to the margin attribute.
- `font` - A Font [https://www.codenameone.com/javadoc/com/codename1/ui/Font.html] object instance.
- `alignment` - an `Integer` object containing the LEFT/RIGHT/CENTER constant values defined in Component.
- `textDecoration` - an `Integer` value containing one of the TEXT_DECORATION_* constant values defined in Style.
- `backgroundType` - a `Byte` object containing one of the constants for the background type defined in Style [https://www.codenameone.com/javadoc/com/codename1/ui/plaf/Style.html] under BACKGROUND_*.
- `backgroundGradient` - contains an `Object` array containing 2 integers for the colors of the gradient. If the gradient is radial it contains 3 floating points defining the x, y & size of the gradient.

To set the foreground color of a selected button to red, a theme will define a property like:

```
Button.sel#fgColor=ff0000
```

This information is useful for understanding how things work within Codename One, but it can also be useful in runtime.

For example: to increase the size of all fonts in the application, you can do something like:

```
Hashtable h = new Hashtable();
h.put("font", largeFont);
UIManager.getInstance().addThemeProps(h);
Display.getInstance().getCurrent().refreshTheme();
```

4.7. Understanding images and Multi-Images



This section provides a high-level overview of images. You dive deeper into the various types of images in the graphics section [https://www.codenameone.com/manual/graphics.html#deep-into-images-section].

When working with a theme, you often use images for borders or backgrounds. You also use images within the GUI for various purposes and most such images will be extracted from the resource file.

Adding a standard JPEG/PNG image to the resource file is straight forward, and the resulting image can be viewed within the images section. For example, due to the wide difference between device types, an image that would be appropriate in size for an iPhone 3gs would not be appropriate in size for a Nexus device or an iPhone 4 (but perhaps, it will be right for iPad 1 and iPad 2).

The density of the devices varies and Codename One tries to simplify the process by unifying everything into one set of values to show density. For simplicity's sake, density is sometimes expressed in pixels, but it's mapped internally to actual screen measurements where possible.

A multi-image is an image that has many varieties for different densities, and thus looks sharp in all the densities. Since scaling on the device can't interpolate the data (due to performance considerations), significant scaling on the device becomes impractical. For example, a multi-image will provide the "right" resolution image for the given device type.

From the programming perspective this is seamless, a developer accesses one image and has no ability to access the images in the different resolutions. Within the designer, but, you can explicitly define images for many resolutions and perform high quality scaling so the "right" image is available.

You can use two basic methods to add a multi-image: quick add and standard add.

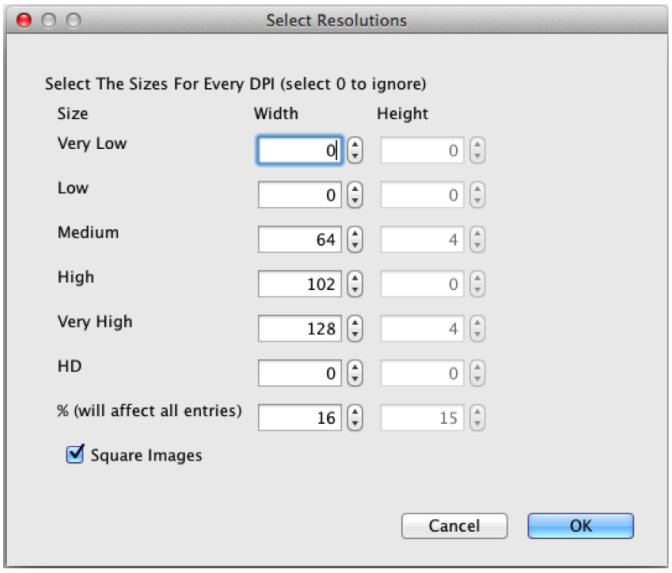
Both methods rely on understanding the source resolution of the image, for example: if you have an icon that you expect to be 128×128 pixels on iPhone 4, 102×102 on nexus one and 64×64 on iPhone 3gs. You can provide the source image as the 128 pixel image and perform a quick add option while picking the # High# density option.

This will show to the algorithm that your source image is designed for the " high" density and it will scale for the rest of the densities accordingly.



This relies on the common use case of asking your designer to design for one high end device (for example: iPhone X) then you can take the resources and add them as "HD" resources. They will automatically adapt to the lower resolutions

Or, you can use the standard add multi-image dialog and set it like this:



Notice that you selected the square image option, essentially eliminating the height option. Setting values to 0 prevents the system from generating a multi-image entry for that resolution, which will mean a device in that category will fall on the closest alternative.

The percentage value will change the entire column, and it means the percentage of the screen. For example: You know the icon is 128 for the high resolution, you can move the percentage until you reach something close to 128 in the " High" row and the other rows will represent a size that should be pretty close in physical size to the 128 figure.

At runtime, you can always find the host device's approximate pixel density using the `Display.getDeviceDensity()` method. This will return one of:

Table 4. Densities

Constant	Density	Example Device
`Display.DENSITY_VERY_LOW`	~ 88 ppi	
`Display.DENSITY_LOW`	~ 120 ppi	Android ldpi devices
`Display.DENSITY_MEDIUM`	~ 160 ppi	iPhone 3GS, iPad, Android mdpi devices
`Display.DENSITY_HIGH`	~ 240 ppi	Android hdpi devices
`Display.DENSITY_VERY_HIGH`	~ 320 ppi	iPhone 4, iPad Air 2, Android xhdpi devices
`Display.DENSITY_HD`	~ 540 ppi	iPhone 6+, Android xxhdpi devices
`Display.DENSITY_560`	~ 750 ppi	Android xxxhdpi devices
`Density.DENSITY_2HD`	~ 1000 ppi	
`Density.DENSITY_4K`	~ 1250ppi	

4.8. Use millimeters for Padding/Margin and font sizes

When configuring your styles, you should rarely use "Pixels" as the unit for padding, margins, font

size, and border thickness because the results will be inconsistent on different densities. Instead, you should use millimeters for all non-zero units of measurement.

As you now understand the complexities of DPI it should be clear why this is important.

4.8.1. Fractions of millimeters

Sometimes millimeters don't give you enough precision for what you want to do. The designer allows you to specify integer values for most units. For example, you can achieve more precise results when working directly in Java. The `Display.convertToPixels()` method will allow you to convert millimeters (or DIPS) to pixels. It also takes an integer input, but you can use it to get a multiplier that you can then use to convert any millimeter value you want into pixels.

For example:

```
double pixelsPerMM = ((double) Display.getInstance().convertToPixels(10, true)) / 10.0;
```

And now you can set the padding on an element to 1.5mm. For example:

```
myButton.getAllStyles().setPaddingUnit(Style.UNIT_TYPE_PIXELS);
int pixels = (int) (1.5 * pixelsPerMM);
myButton.getAllStyles().setPadding(pixels, pixels, pixels, pixels);
```

4.9. Creating a great looking side menu

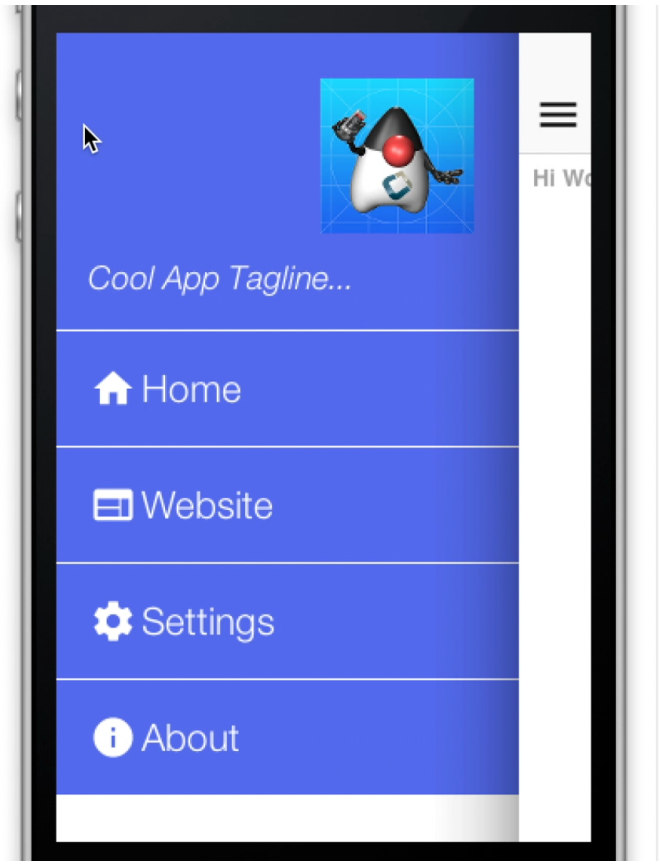


Figure 62. Side Menu final result

A side menu is a crucial piece of an elegant application. You will explain how one creates a simple side menu that's elegant, portable and easy to build. This is a good "starting point" side menu from which you can build more elaborate designs.

To get this result you will start from a native theme and a bare-bones application to keep things simple.

The code for the side menu is this:

```
Form hi = new Form("Hi World");

Toolbar tb = hi.getToolbar();
Image icon = theme.getImage("icon.png"); ①
Container topBar = BorderLayout.east(new Label(icon));
topBar.add(BorderLayout.SOUTH, new Label("Cool App Tagline...", "SidemenuTagline")); ②
topBar.setUIID("SideCommand");
tb.addComponentToSideMenu(topBar);

tb.addMaterialCommandToSideMenu("Home", FontImage.MATERIAL_HOME, e -> {}); ③
tb.addMaterialCommandToSideMenu("Website", FontImage.MATERIAL_WEB, e -> {});
tb.addMaterialCommandToSideMenu("Settings", FontImage.MATERIAL_SETTINGS, e -> {});
tb.addMaterialCommandToSideMenu("About", FontImage.MATERIAL_INFO, e -> {});

hi.addComponent(new Label("Hi World"));
hi.show();
```

① This is the icon which was used instead of a logo it appears in the top right of the side menu

② This is the top bar containing the tagline and the icon it's styled as if it's a command but you can put anything here for example: an image etc.

③ The commands are added as usual to the side menu with no styling or functionality, the entire look is determined by the theme

Next you will open the designer tool to style the UI

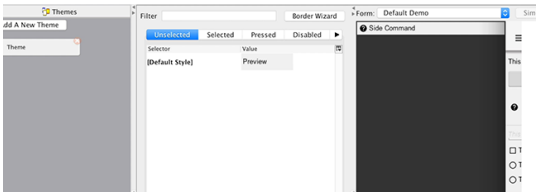


Figure 63. Open the side menu so you will get the right values in the combo box on add

Now when you press `Add` the side menu entries will appear in the combo box (you can type them but this is more convenient). You will start with the `SideNavigationPanel` style:

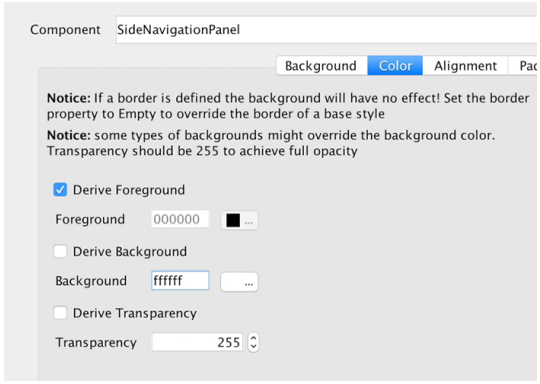


Figure 64. The SideNavigationPanel has an opaque white background

The `SideCommand` style is a bit more elaborate, you start with a white foreground and an opaque bluish/purple color:

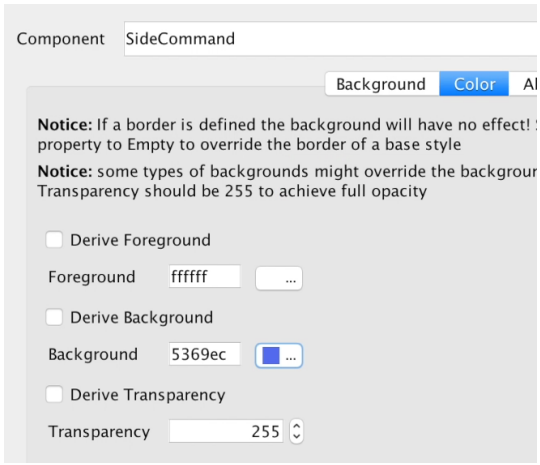


Figure 65. The SideCommand has a white foreground and opaque bluish background

You will set padding to 3 millimeters which provides everything a good feel and spacing. This is important for finger touch sensitivity.

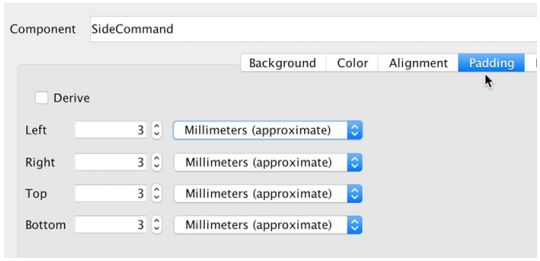


Figure 66. Padding is 3mm so it will feel spacious and touch friendly

You will set margin to 0 except for the bottom one pixel which will leave a nice white line by showing off the background. This means the commands will have a space between them and the white style you gave to the `SideNavigationPanel` will appear through that space.

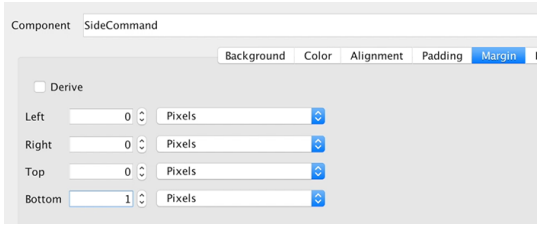


Figure 67. Margin is 0 except for a thin line below each command

Setting the border to empty is crucial!

The iOS version of the side command inherits a border style so you must "remove" it by defining a different border in this case an empty border. Since borders take precedence over color this would have prevented the color changes you made from appearing.

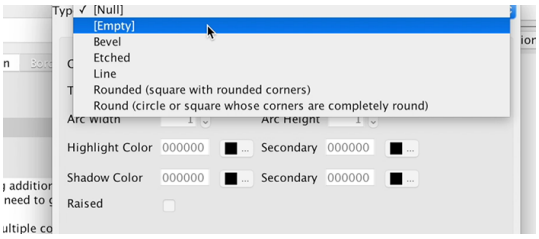


Figure 68. Border must be defined as Empty

Next you need to pick a good looking font and make sure it's large enough. You use millimeters size it for all OS's and override the derived text decoration which has a value in the iOS native theme so it can impact the final look.

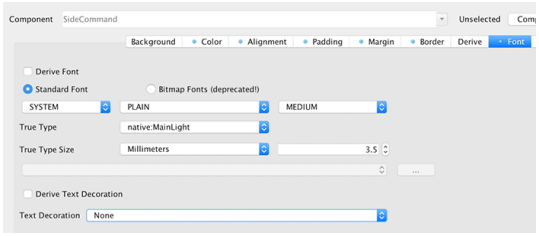


Figure 69. Pick a good looking font for the side command

Next you need to move to the selected tab and add a new side command entry that derives from the unselected version. You will pick a new color that's slightly deeper and will make the selected style appear selected. You will also copy and paste this selected style to the pressed style.

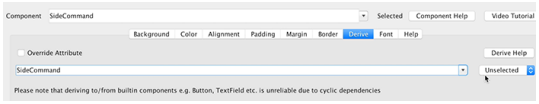


Figure 70. Selected & Pressed SideCommand

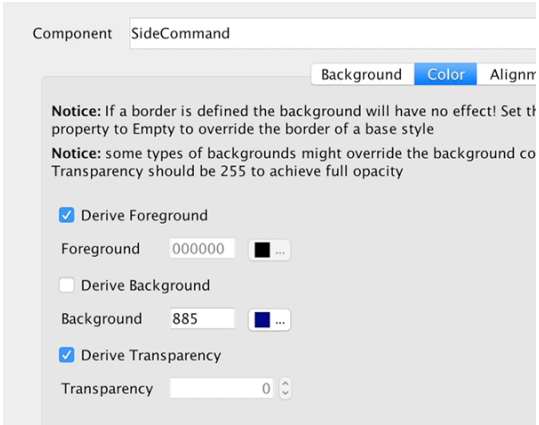


Figure 71. Color for the Selected/Pressed SideCommand

The `SidemenuTagline` is a `SideCommand` style that was slightly adapted. You will remove the padding and margin because the whole section is wrapped in a side command and you don't want double padding. You will leave 1mm padding at the top for a bit of spacing from the logo.

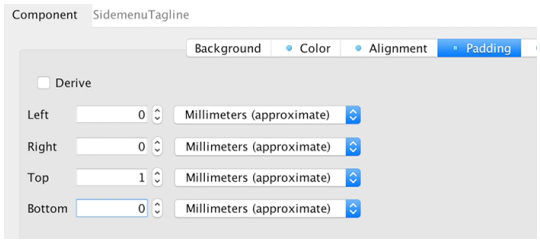


Figure 72. Padding of the SidemenuTagline

You will also update the font to a smaller size and italic styling so it will feel like a tagline.

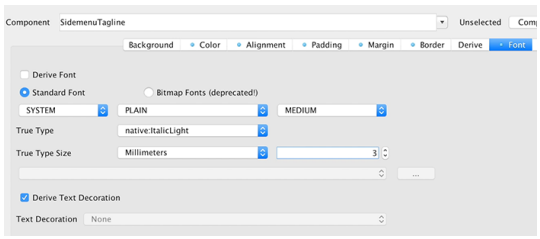


Figure 73. Font for the SideMenuTagline is slightly smaller and italic

The last change for the theme is for the `StatusBarSideMenu` UIID which is a spacing on the top of the sidemenu. This spacing is there for iOS devices which render the clock/battery/reception symbols on top of the app. You will set the padding to 0.

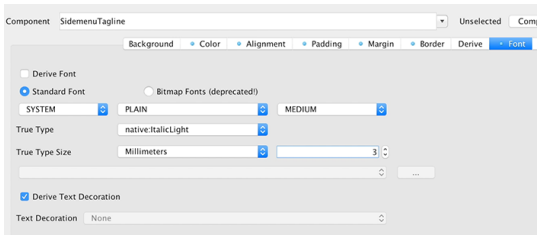


Figure 74. StatusBarSideMenu padding for the top of the side menu

Finally, you will add the icon image (or a logo if you have it) into the theme as a multi image so you can use it within the side menu as a good looking logo. A large icon image works as a 2HD multi-image but you can use many strategies to get a fitting image for this spot.



Rounded images work well here, you can round images dynamically using masking

These steps produce the UI above as a side menu, they might seem like a long set of steps but each step is pretty simple as you walk through each one. This does show off the versatility and power of Codename One as a change to one step can create a radically different UI design.

4.10. Converting a PSD to a theme

Codename One provides extensive support for designing beautiful user interfaces, but it isn't necessarily obvious to new developers how to achieve their desired results. A common workflow for app design includes a PSD file with mock-ups of the UI, created by a professional designer.



PSD is the Adobe Photoshop file format, it's the most common format for UI designs in the industry

For this tutorial you adapt a slick looking sign-up form found online and convert it to a Codename One component that can be used inside an application.

The process you followed was:

1. Find the PSD design you want to use: this PSD file [http://freebiesbug.com/psd-freebies/iphone-6-ui-kit/] created by Adrian Chiran [https://dribbble.com/adrianchiran] (you mirrored it here [https://www.codenameone.com/files/iOS_UI-Kit.psd] in case it goes offline):

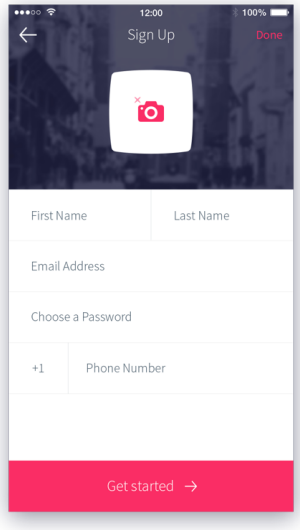


Figure 75. Sign Up form Design

2. Re-create the general structure and layout of the design in a Codename One `Form` using nested components and layout managers. Here is a break-down of how you structured the component hierarchy in the `Form`:

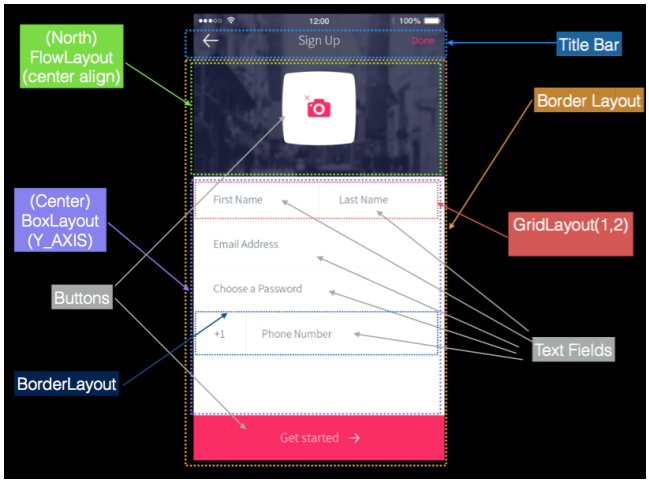


Figure 76. Component hierarchy and layouts

3. Extract the images you needed using Photoshop - this process is often referred to as "cutting"
4. Extract the fonts, colors, and styles you needed to reproduce the design in Codename One
5. Import images into the Codename one project, and define theme styles so that your components match the look of the original design

Here is a screenshot of the resulting component running inside the Codename One simulator:

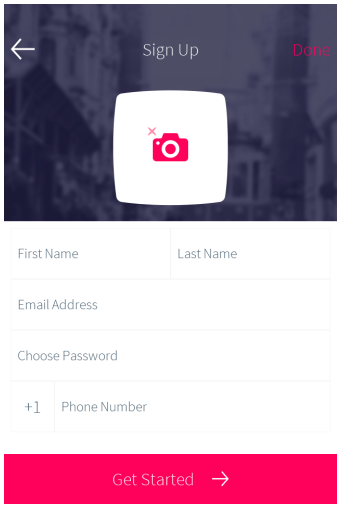


Figure 77. Resulting app in the Codename One simulator

4.10.1. Breaking down the PSD

Open the PSD you're interested in using Photoshop.



You might be missing fonts in your system so you can either install them or ignore that. Keep in mind that some fonts might not be redistributable with your application

In this PSD you want one of the screen designs so initially you want to remove everything that isn't related so you can get your bearings more effectively:

- Select the drag tool (the top left tool)
- In the toolbar for the tool (top bar area) check the `Auto Select` mode
- Select the `Layer Mode` for auto selection (sometimes group would actually be better so feel free to experiment)
- Click on the portion in the PSD that you're interested in

You should end up with something like this where a layer is selected in the layers window:

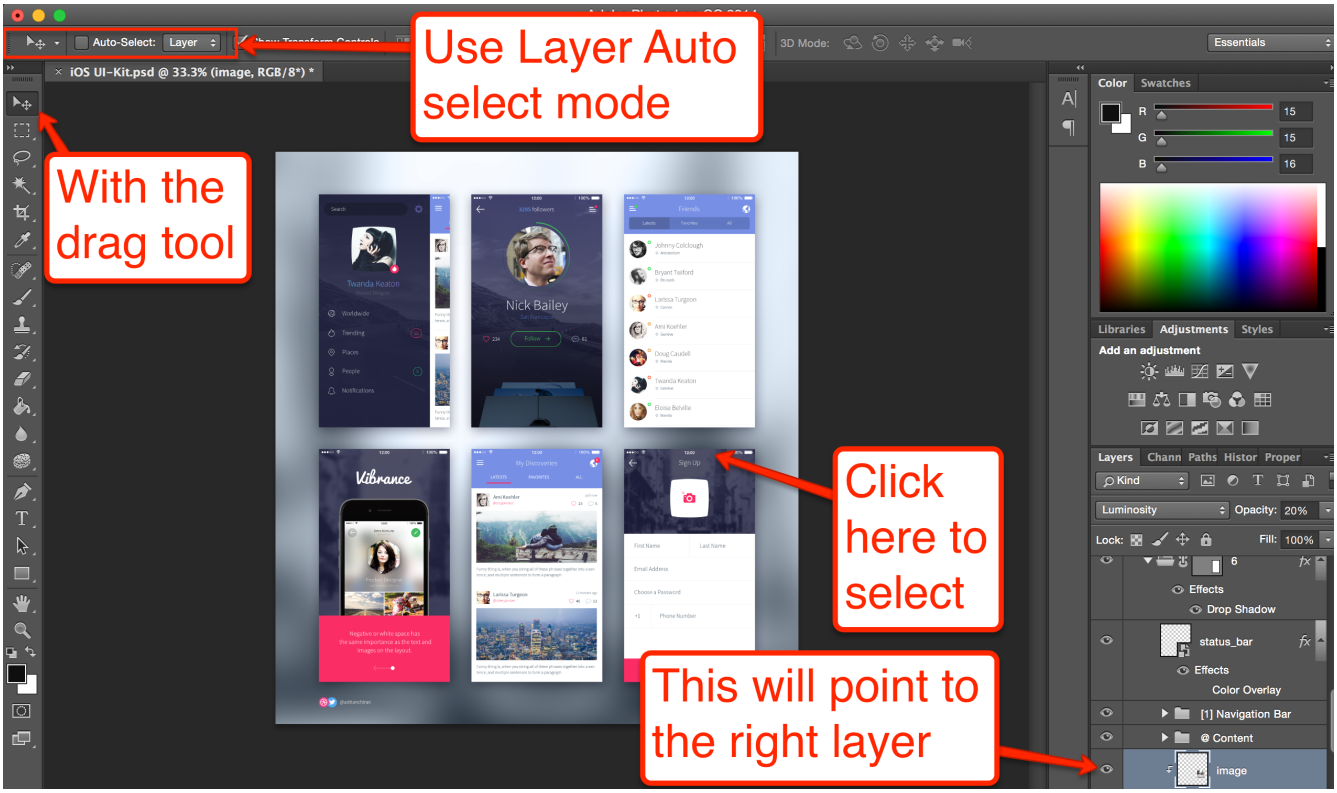


Figure 78. Selecting a layer from the region you're interested in

Scroll up the hierarchy a bit and uncheck/recheck the eye icon on the left until you locate the right element layer.

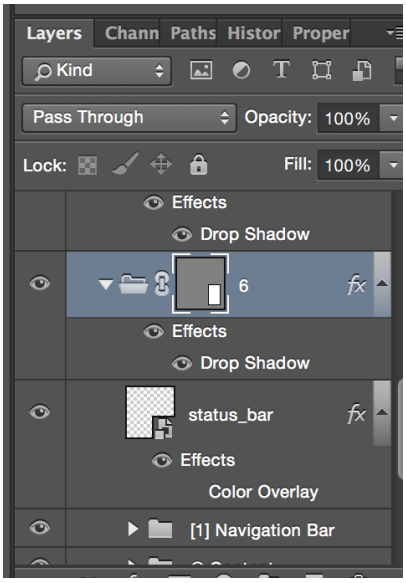


Figure 79. Selecting a layer from the region you're interested in

Right-click the layer and select `Convert To Smart Object`.



The right click menu will present different options when you click different areas of the layer, clicking on the left area of the layer works

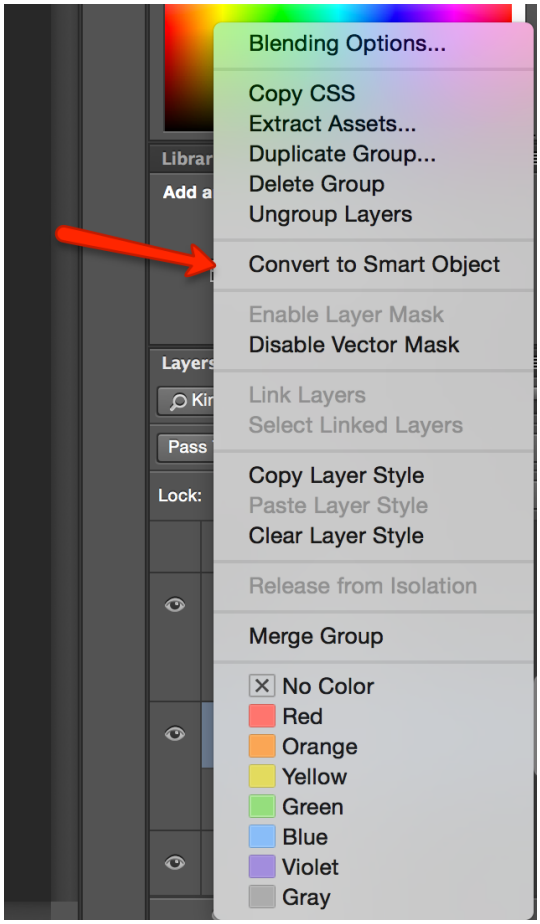


Figure 80. In the right click menu option select "Convert To Smart Object"

Once the layer hierarchy is a smart object you can double-click it which will open the sub hierarchy in a new tab and you now have the pieces of the image you care about.

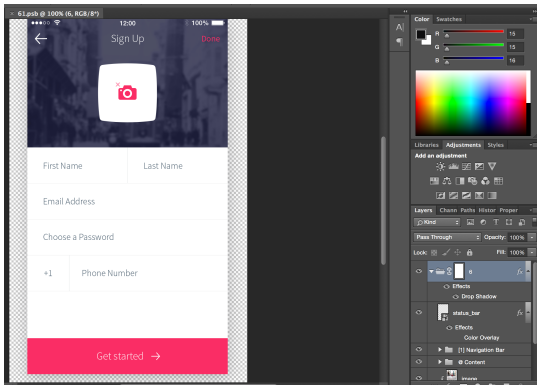


Figure 81. Double clicking the smart object allows you to edit the form you need

Removing the noise

The first thing you need to do is remove from the image all the things that you don't need. The status bar area on the top is redundant as if is a part of the phones UI. You can select it using the select tool and click the eye icon next to the layer to hide it.

You'd want to have the back arrow but thanks to the material design icons that are a part of Codename One you don't need that icon so you can hide that too.

You don't need the "Sign Up" or "Done" strings in the title either but before removing them you'd like to know the font that's used.

To discover that, click them to select the layer then switch to the text tool:



Figure 82. The text tool allows you to inspect the font used

Then double-click the text area layer to find out the font in the top of the UI like this:



Figure 83. The Done toolbar entry uses SourceSansPro Regular



Notice that you don't actually need to have the font installed; in this case the font isn't installed (hence the square brackets)

Also notice that the color of the font is accessible in that toolbar, by clicking the color element you get this dialog which shows the color value to be `f73267`, this is something you will use later

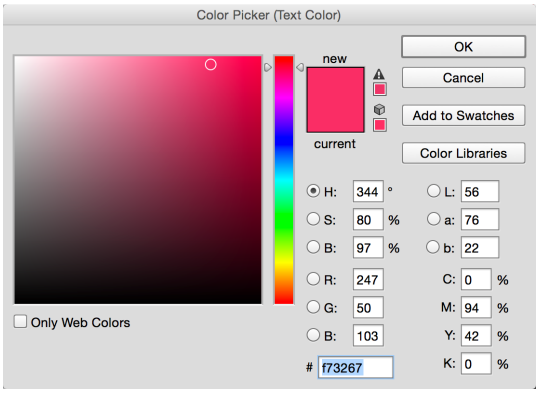


Figure 84. The color dialog lists the hex color at the bottom, you can paste that directly to the designer tool

You can now hide both text layers so they won't pose a problem later.

The camera Button

The camera button includes an icon and the button background itself. You can use that as a single image and be done with it, but for this tutorial you will take the harder route of separating this into a button background and a foreground image.

When you click on the camera icon you will notice that the camera icon comprises two separate layers: the camera and the "x" symbol above it. You can select both layers using `ctrl-click` (command click on the Mac) and convert both to a smart object together using the same method as before:



Figure 85. The camera smart object

Since the image is used as an icon you want it to be square which isn't the situation here!
This is important as a non-square image can trigger misalignment when dealing with icons and the background. You need to use the Image → Canvas Size menu and set the values to be the same (the higher value of the two).

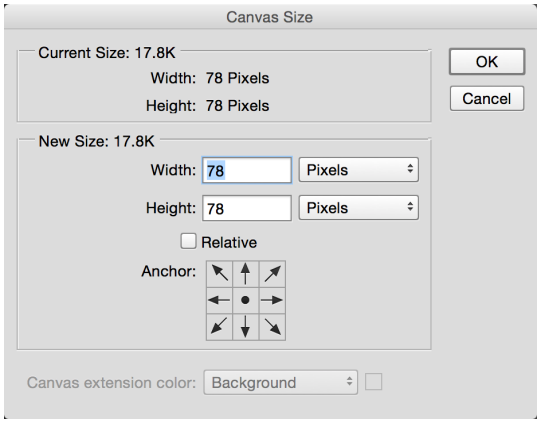


Figure 86. The canvas size dialog for the camera.png file

You can now use File → Export and save the first image resource you will need into a temporary directory. Make sure to save a PNG file to preserve quality and transparency!



Use File → Export and never use File → Save As. The latter can produce a huge size difference as it retains image meta-data

For convenience you will refer to the file as `camera.png` when you need it later.



Figure 87. The camera icon image

You can follow the exact same procedure with the parent button layer (the white portion) which you can convert to a smart object and export `camera-button.png`.

Figure 88. The camera button image set to a gray background so it will be visible

Now you can hide both of these elements and proceed to get the background image for the title.

Here the "smart object trick" won't work... An effects layer is in place and the smart object will provide you with the real underlying image instead of the look you actually want. For example, solving this is trivial now that you hid all the elements on top of the image!

You need to switch to the rectangular select tool:

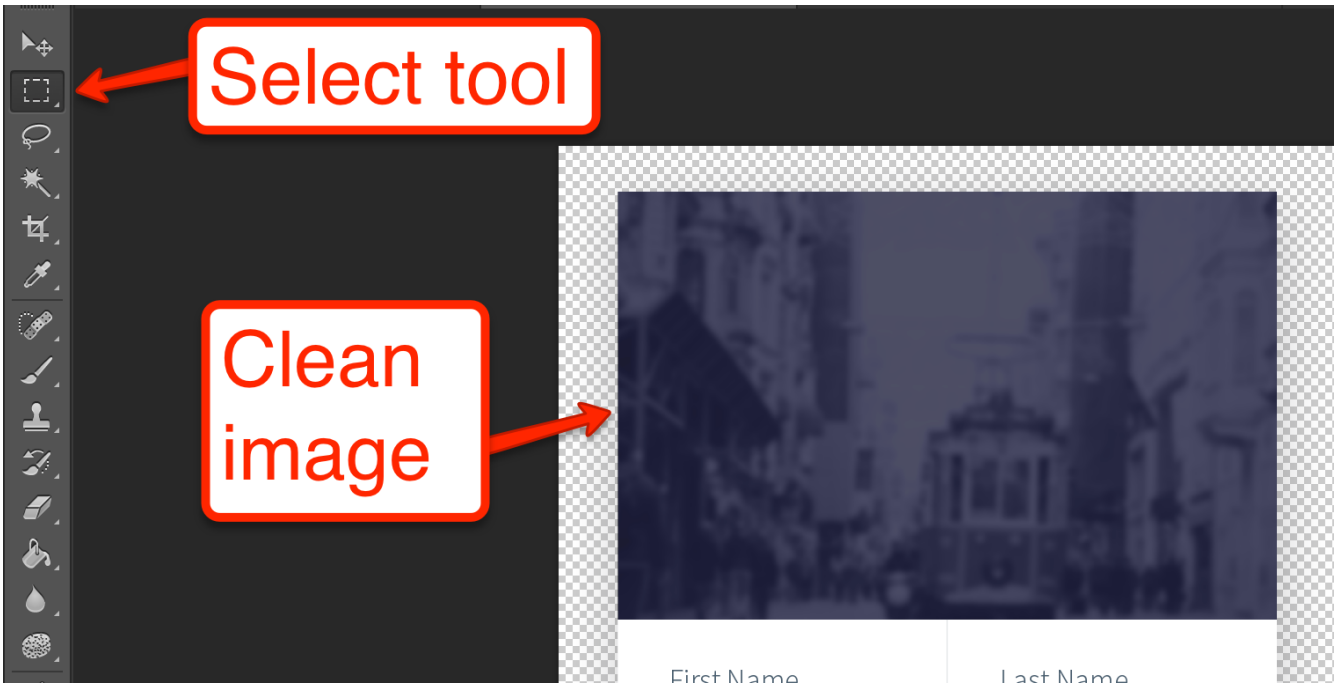


Figure 89. The select tool and the clean image you want to select

Now drag the select tool to select the image don't cross into the white pixels below the image. You can use the zoom value and set it to a high value to get the selection right.

When the selection is right click Edit → Copy Merged. Copy would copy a specific layer but in this case you want to copy what you see on the screen!

Now click File → New it should have the Presets set to Clipboard which means the newly created image is based on what you copied (great UX). Just accept that dialog and paste (Ctrl-V or Command-V).

You can now save the image, since it's a background using JPEG is totally acceptable in this case. You named it `background.jpg`.



Figure 90. The background image

The last thing you need is the colors used in the UI. You can use the "eye drop" tool in a high zoom level to discover the colors of various elements for example: the text color is `4d606f` and the separator color is `f5f5f5`:

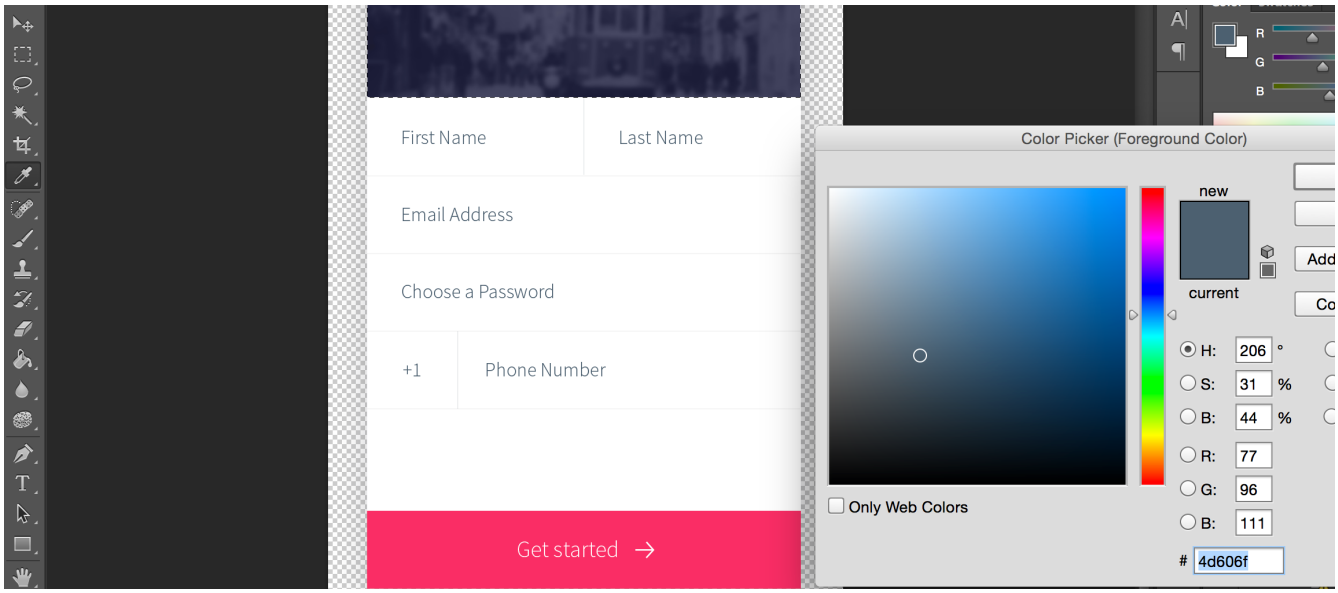


Figure 91. The eye drop tool can be pointed at an area of the image to get the color in that region

4.10.2. The Code

While that was verbose it was simple. You will create a simple barebones manual application with the native theme.



The reason for this is to avoid "noise," if you use a more elaborate theme it would have some existing settings. This can make the tutorial harder to follow

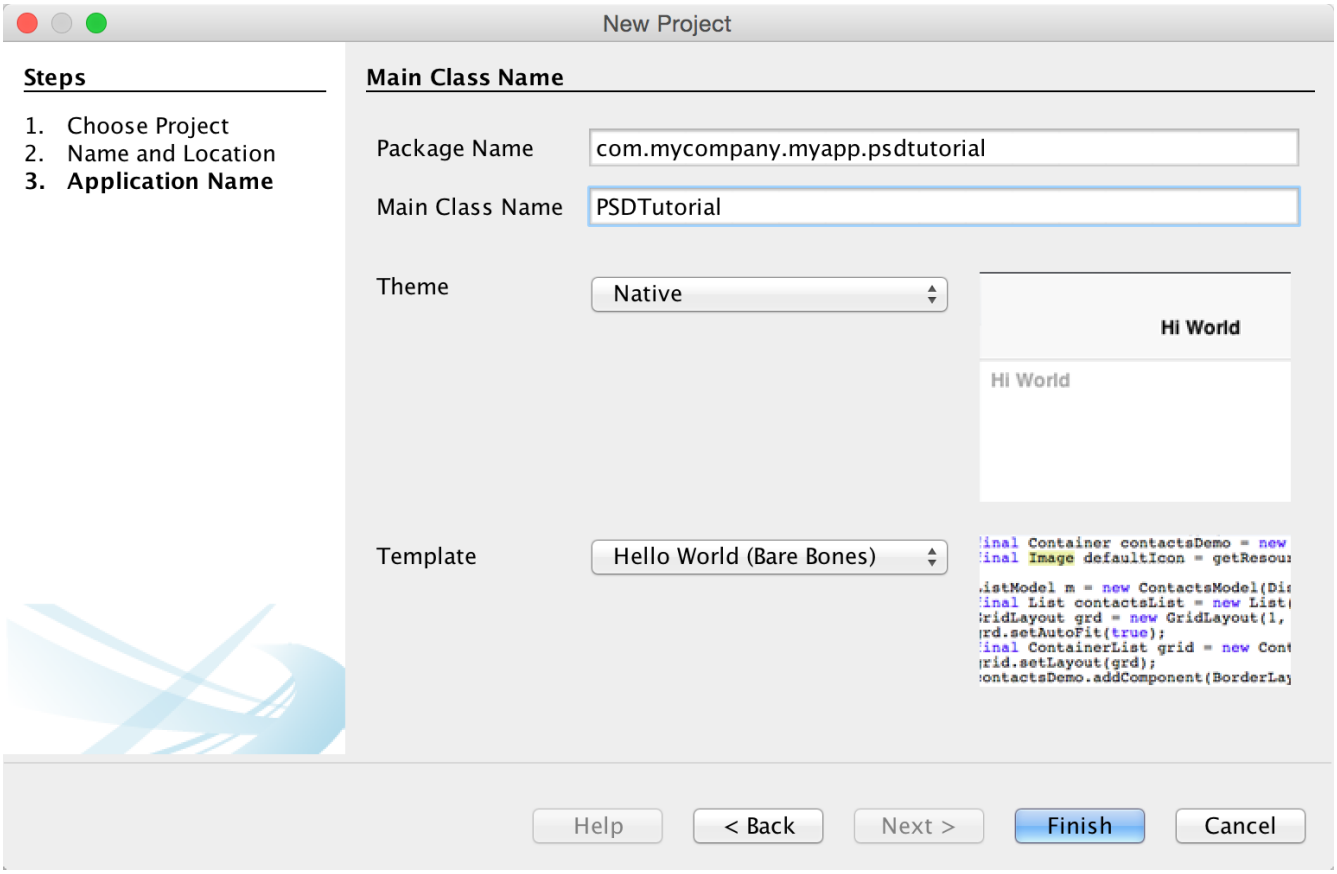


Figure 92. Simple bare-bones app settings

Once the project is created double-click the `theme.res` file and within the designer select Images → Quick Add Multi Images. Select the 3 images you created above: `background.jpg`, `camera.png` & `camera-button.png`. Leave the default setting on # High# and press OK.

Then save the resource file so you can use these images from code.

Here is the source code you used to work with the UI above there are comments within the code explaining some logic:

```
private Label createSeparator() {
    Label sep = new Label();
    sep.setUIID("Separator");
    // the separator line  is implemented in the theme using padding and background color, by default labels
    // are hidden when they have no content, this method disables that behavior
    sep.setShowEvenIfBlank(true);
    return sep;
}
```

```
if (current != null) {
    current.show();
    return;
}
// The toolbar uses the layered mode so it resides on top of the background image, the theme makes
// it transparent so we will see the image below it, we use border layout to place the background image on
// top and the "Get started" button in the south
Form psdTutorial = new Form("Signup", new BorderLayout());
Toolbar tb = new Toolbar(true);
psdTutorial.setToolbar(tb);
```

```
// we create 4mm material arrow images for the back button and the Get started button
Style iconStyle = psdTutorial.getUIManager().getComponentStyle("Title");
FontImage leftArrow = FontImage.createMaterial(FontImage.MATERIAL_ARROW_BACK, iconStyle, 4);
FontImage rightArrow = FontImage.createMaterial(FontImage.MATERIAL_ARROW_FORWARD, iconStyle, 4);

// we place the back and done commands in the toolbar, we need to change UIID of the "Done" command
// so we can color it in Red
tb.addCommandToLeftBar("", leftArrow, (e) -> Log.p("Back pressed"));
Command doneCommand = tb.addCommandToRightBar("Done", null, (e) -> Log.p("Done pressed"));
tb.findCommandComponent(doneCommand).setUIID("RedCommand");

// The camera button is comprised of 3 pieces. A label containing the image and the transparent button
// with the camera icon on top. This is all wrapped in the title container where the title background image
// is placed using the theme. We chose to use a Label rather than a background using the cameraLayer so
// the label will preserve the original size of the image without scaling it and take up the space it needs
Button cameraButton = new Button(theme.getImage("camera.png"));
Container cameraLayer = LayeredLayout.encloseIn(
        new Label(theme.getImage("camera-button.png")),
        cameraButton);
cameraButton.setUIID("CameraButton");
Container titleContainer = Container.encloseIn(
        new BorderLayout(BorderLayout.CENTER_BEHAVIOR_CENTER),
        cameraLayer, BorderLayout.CENTER);
titleContainer.setUIID("TitleContainer");

TextField firstName = new TextField("", "First Name");
TextField lastName = new TextField("", "Last Name");
TextField email = new TextField("", "Email Address", 20, TextField.EMAILADDR);
TextField password = new TextField("", "Choose a Password", 20, TextField.PASSWORD);
TextField phone = new TextField("", "Phone Number", 20, TextField.PHONENUMBER);
Label phonePrefix = new Label("+1");
phonePrefix.setUIID("TextField");

// The phone and full name have vertical separators, we use two table layouts to arrange them correctly
// so the vertical separator will be in the right place
TableLayout fullNameLayout = new TableLayout(1, 3);
Container fullName = new Container(fullNameLayout);
fullName.add(fullNameLayout.createConstraint().widthPercentage(49), firstName).
        add(fullNameLayout.createConstraint().widthPercentage(1), createSeparator()).
        add(fullNameLayout.createConstraint().widthPercentage(50), lastName);
Container fullPhone = TableLayout.encloseIn(3, phonePrefix, createSeparator(), phone);

// The button in the south portion needs the arrow icon to be on the right side so we place the text on the left
Button southButton = new Button("Get started", rightArrow);
southButton.setTextPosition(Component.LEFT);
southButton.setUIID("SouthButton");

// we add the components and the separators the center portion contains all of the elements in a box
// Y container which we allow to scroll. BorderLayout Containers implicitly disable scrolling
Container by = BoxLayout.encloseY(
        fullName,
        createSeparator(),
        email,
        createSeparator(),
        password,
        createSeparator(),
        fullPhone,
        createSeparator()
);
by.setScrollableY(true);
psdTutorial.add(BorderLayout.NORTH, titleContainer).
        add(BorderLayout.SOUTH, southButton).
```

```
        add(BorderLayout.CENTER, by);

psdTutorial.show();
```

4.10.3. Styling the UI

The code above is most of the work but you still need to put everything together using the theme. This is what you've so far:

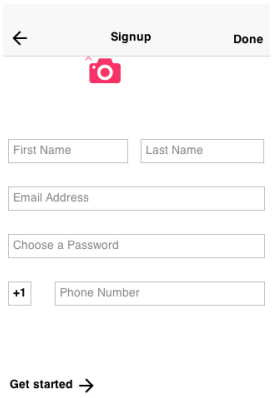


Figure 93. Before applying the changes to the theme this is what you've

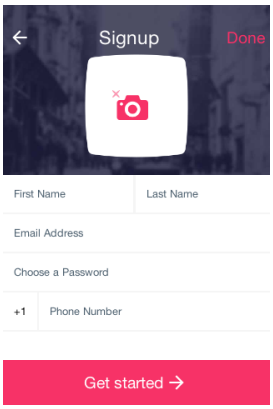


Figure 94. This is what you're aiming at with no more code changes

This looks like a major set of changes but it requires 10 UIID definitions to get to this look!

Open the designer and select the theme. Press the `Add` button and type in `TitleContainer`. Uncheck derive for the background and select `IMAGE_SCALED_FILL` for the `Type` and the `background.jpg` image.

Define the padding as:

- Left - 3 millimeters
- Right - 3 millimeters
- Top - 8 millimeters
- Bottom - 2 millimeters

This will allow enough space for the title. Define margin as 0 on all sides. Then press `OK`.

Add the "Title" UIID. In the `Color` tab define the foreground as `ffffff` define transparency as `0` (fully

transparent so you will see the TitleContainer). Define padding as 1 millimeter on all sides and margin as 0 on all sides.

In the Border tab press the ... button and select [Empty].

In the Font tab select the True Type as native:MainThin. Select the True Type Size as millimeters and set the value to 3.5.

Press OK to save the changes.

Copy the Title UIID and paste it, change the name to "TitleCommand" and press OK to save the changes.

Copy the Title UIID again and paste it, change the name to RedCommand. In the Color tab set the foreground color to f73267. In the Font tab set the True Type to native:MainLight and set the size to 3. Press OK to save the changes.

Add the "TitleArea" UIID. In the Color tab define transparency as 0 (fully transparent so you will see the TitleContainer). Define padding and margin as 0 on all sides.
In the Border tab press the ... button and select [Empty]. Press OK to save the changes.

Add the "TextField" UIID. In the Color tab define transparency as 255 (fully opaque) and the background as ffffff (white). Define padding as 2 millimeters on all sides and margin as 0 on all sides.
In the Border tab press the ... button and select [Empty]. In the Font tab set the True Type to native:MainLight and set the size to 2. Press OK to save the changes.

Copy the TextField UIID again and paste it, change the name to TextHint. In the Color tab set the foreground color to 4d606f. Press OK to save the changes.

Add the "SouthButton" UIID. In the Color tab define transparency as 255 (fully opaque) and the background as f73267 (red) and the foreground as ffffff (white). Define Alignment as Center.

Define padding as:

- Left - 1 millimeter
- right - 1 millimeter
- top - 2 millimeters
- bottom - 2 millimeters

Define margin as 0 on all sides. In the Font tab set the True Type to native:MainThin and set the size to 3. Press OK to save the changes.

Add the "CameraButton" UIID. In the Color tab define transparency as 0 (fully transparent). Define Alignment as Center.
Define padding as:

- Left - 1 millimeter
- right - 1 millimeter

- top - 3 millimeters
- bottom - 1 millimeter



This helps spacing away from the title

Define margin as 1 millimeter on all sides. Press `OK` to save the changes.

You can now save the theme and the app should look like the final result!

Not quite there yet

One last piece would become apparent if you actually try to run this code. When pressing the buttons/text fields you would see their look change due to the different styles for focus/press behavior.

You can derive the regular styles from the selected/pressed styles but one of the simplest ways is to copy & paste the styles to the pressed/selected tabs. You can copy `CameraButton`, `RedCommand`, `SouthButton` & `TextField` to the selected state. Then copy `CameraButton`, `RedCommand` & `SouthButton` to the pressed state to get the complete app running!

Chapter 5. Native modern themes

Codename One ships an iOS Modern (liquid-glass) theme and an Android Material 3 theme that you can opt your app into with a single build hint. Both are fully overridable - their color palettes, type ramps, and state-specific styles are all reachable from your own `theme.css` and from runtime API.

The legacy iOS 7 (iOS) and Holo Light (Android) themes remain the default so existing apps see no behavior change.

5.1. Selecting a theme

Three build hints control the platform native theme. None of them are required; with all three unset every app continues to load the legacy theme it always did.

Hint	Values	Description

`ios.themeMode`	`auto` (default)
	`modern` / `liquid`
	`ios7` / `flat`
	`legacy` / `iphone`
		`auto` and `ios7` both load the iOS 7 flat theme. `modern` / `liquid` opts in to the modern theme generated from `native-themes/ios-modern/theme.css`. `legacy` / `iphone` loads the pre-iOS 7 theme. The `auto` → `modern` flip is planned for a future release.

`and.themeMode`	`auto` / `modern` / `material`
	`hololight` (legacy default)
	`legacy`
		`auto` and `modern` / `material` opt in to the modern theme generated from `native-themes/android-material/theme.css`. `hololight` is Android Holo Light (the pre-modern default for existing apps). `legacy` is the pre-Holo theme. The legacy alias `cn1.androidTheme` is still honored for back-compat.

`nativeTheme`	`modern` / `legacy` / `custom`	Cross-platform shortcut that sets both `ios.themeMode` and `and.themeMode` together. `modern` = liquid glass + Material 3, `legacy` = iOS 7 + Holo Light, `custom` disables the framework theme entirely so your own `theme.css` owns the base layer. The legacy alias `cn1.nativeTheme` is still honored for back-compat.

The legacy `and.hololight=true` hint still works and maps to `and.themeMode=hololight`.

5.2. Light and dark mode

Both modern themes ship a dark variant via a `@media (prefers-color-scheme: dark)` block in their CSS. The CSS compiler rewrites each rule inside that block into a `$Dark<UIID>` style entry; UIManager picks them up whenever `CN.isDarkMode() == true`.

`Display.getInstance().setDarkMode(Boolean)` controls the active appearance:

- `null` (the default) - follow the device. iOS reads `UITraitCollection`, Android reads `Configuration.uiMode`. When the user toggles light / dark in system settings the app reflects the change.
- `true` / `false` - force the app into one mode regardless of system setting (useful for theme-preview screens or accessibility toggles).

5.3. Material 3 color palette (Android)

The Android Material 3 theme is built around the Material Design 3 "baseline" palette. Each color is referenced from one or more UIIDs; if you change a color the change ripples to every UIID that inherits from it.

Role	Light	Dark	Used by
primary	`#6750a4`	`#d0bcff`	Default `Button` fill, `BackCommand` / `TitleCommand` text, `SelectedTab` text color.
on-primary	`#ffffff`	`#381e72`	Text printed on top of the primary fill.
primary-container	`#eaddff`	`#4f378b`	`RaisedButton` (the elevated tone), `Button.pressed` and `MultiButton.pressed` highlight, `FloatingActionButton`.
surface	`#fef7ff`	`#141218`	`Form`, `ContentPane`, `Toolbar`, `TitleArea`, `List`, `Tabs`, `SideNavigationPanel`. The "page" color.
on-surface	`#1d1b20`	`#e6e0e9`	Body text on surfaces (`Label`, `Title`, `MultiLine1`).
surface-variant	`#e7e0ec`	`#49454f`	Track of the off-state `Switch`.
on-surface-variant	`#49454f`	`#cac4d0`	Secondary text (`SecondaryLabel`, `MultiLine2`, `UnselectedTab` text).
surface-container	`#f3edf7`	`#211f26`	`TextField` / `TextArea` filled background.
surface-container-highest	`#e6e0e9`	`#211f26`	`Dialog`, `PopupContent`. Sits one elevation step above `surface` so dialogs read as raised cards.
outline / outline-variant	`#79747e` / `#cac4d0`	`#938f99` / `#49454f`	`Separator`, `TertiaryLabel`, disabled-text muting.
state-pressed	`#d0bcff`	`#4f378b`	All `.pressed` overrides on Button-family UIIDs.
state-disabled / on-disabled	`#e0dce4` / `#a5a0ab`	`#2b2930` / `#5c5967`	All `.disabled` overrides.

To rebrand the app, override the color at the role level rather than touching every UIID. The example below layers a teal palette on top from your app's own `theme.css`; if you want the same

rebrand to apply at **runtime** (for in-app accent toggles, branded flavours, A/B tests) without recompiling the theme, see `Runtime accent palette override` further down - a single `addThemeProps({"@accent-color": …})` call retunes every accent-bearing UIID at once:

```
#Constants {
    includeNativeBool: true;
    darkModeBool: true;
}

/* Touch the four UIIDs that paint with primary / primary-container.
   The role-as-rule pattern keeps brand changes localised. */
Button { background-color: #00796b; }
Button.pressed { background-color: #4db6ac; color: #00251a; }
RaisedButton { background-color: #b2dfdb; color: #00251a; }
RaisedButton.pressed { background-color: #80cbc4; }
SelectedTab { color: #00796b; }
BackCommand { color: #00796b; }
TitleCommand { color: #00796b; }
```

5.4. iOS modern color palette

The iOS modern theme follows Apple's system palette.

Role	Light	Dark	Used by
accent	`#007aff`	`#0a84ff`	`Button` text, `RaisedButton` fill, `SelectedTab` text, `BackCommand` / `TitleCommand`, `FloatingActionButton`.
accent-pressed	`#0064d1`	`#64b1ff`	All `.pressed` accent overrides.
accent-disabled	`#b3d4ff`	`#004a99`	All `.disabled` accent overrides.
surface	`#ffffff`	`#000000`	Pure surface used by `Toolbar`, `TitleArea`, `Title`, `List`.
surface-grouped	`#f2f2f7`	`#1c1c1e`	Form / ContentPane (the iOS "grouped" form bg). `Dialog` / `Tabs` use this color at reduced opacity for the liquid-glass look.
surface-tertiary	`#e5e5ea`	`#2c2c2e`	`Button.pressed` highlight, elevated dark Dialog surface.
text-primary	`#000000`	`#ffffff`	`Label`, `MultiLine1`, `Title`, primary body text.
text-secondary	`#3c3c43`	`#ebebf5`	`SecondaryLabel`, `MultiLine2`, unselected tabs.
text-tertiary / text-disabled	`#8e8e93` / `#c7c7cc`	`#8e8e93` / `#48484a`	`TertiaryLabel`, `MultiLine3-4`, disabled-state text.

Role	Light	Dark	Used by
separator	#c6c6c8	#38383a	Separator.
success	#34c759	#30d158	Switch.selected, OnOffSwitch.selected.

To layer a brand color on top, override RaisedButton and accent-driven UIIDs the same way as Android above. The color names match Apple's UIColor.systemBlue etc. So you can mirror the SF Symbols semantics if you want.

5.5. Accent palette override

Both shipped native themes expose their accent palette as named constants you can retune from your app's own theme.css (or, for dynamic theming, at runtime). The CSS source uses var(--accent-color, fallback) references so the fallback ships as the baked-in default (the .res file loads fine with no override) and the compiler additionally emits a @cn1-bind:<UIID>.<key>=accent-color constant alongside each affected style key. When the resolved theme constants pick up a new @accent-color value (whether from your CSS or via runtime UIManager.addThemeProps), the framework fans the override out to every bound UIID at once - no per-UIID rule duplication, no theme recompile.

Table 5. iOS modern (native-themes/ios-modern/theme.css)

Constant	Light default	Dark default	Drives
@accent-color	#007aff	(see @accent-color-dark)	Button.fgColor, RaisedButton.bgColor, CheckBox.selected, RadioButton.selected, OnOffSwitch.fgColor, BackCommand, TitleCommand, FloatingActionButton.bgColor.
@accent-color-dark	n/a	#0a84ff	$Dark<UIID> counterparts of the above.
@accent-pressed-color / @accent-pressed-color-dark	#0064d1	#64b1ff	All .press#fg/bgColor accent overrides.
@accent-disabled-color / @accent-disabled-color-dark	#b3d4ff	#004a99	All .dis#fg/bgColor accent overrides.
@accent-on-color	#ffffff	(same)	Text color painted on top of the accent fill.

Table 6. Android Material 3 (native-themes/android-material/theme.css)

Constant	Light default	Dark default	Material 3 token
@accent-color / @accent-color-dark	#6750a4	#d0bcff	primary
@accent-on-color / @accent-on-color-dark	#ffffff	#381e72	on-primary
@accent-container-color / @accent-container-color-dark	#eaddff	#4f378b	primary-container

Constant	Light default	Dark default	Material 3 token
`@accent-on-container-color` / `@accent-on-container-color-dark`	`#21005d`	`#eaddff`	`on-primary-container`
`@accent-pressed-color` / `@accent-pressed-color-dark`	`#d0bcff`	`#4f378b`	`state-pressed`

5.5.1. Override from your app's `theme.css` (recommended)

Redeclare the accent variable inside the `#Constants` block of your app's `theme.css`. The framework's CSS compiler exports any `--name` declaration in `#Constants` as a `@name` theme constant, so the value flows through the same binding pass that handles runtime overrides:

```
#Constants {
    includeNativeBool: true;
    darkModeBool: true;

    /* Override the native theme's accent palette. The compiler picks
       up these declarations from #Constants and exports them as
       @accent-color / @accent-color-dark theme constants so every
       UIID bound to var(--accent-color) in the parent native theme
       picks them up at app launch - no per-UIID rule edit needed. */
    --accent-color: #ff2d95;
    --accent-color-dark: #ff2d95;
    --accent-pressed-color: #c71a75;
    --accent-pressed-color-dark: #c71a75;
    --accent-on-color: #ffffff;

    /* Material 3 RaisedButton uses a separate "container" tonal
       pair; iOS ignores these (no bindings reference them) so it's
       safe to set them unconditionally. */
    --accent-container-color: #ff2d95;
    --accent-container-color-dark: #ff2d95;
    --accent-on-container-color: #ffffff;
    --accent-on-container-color-dark: #ffffff;
}
```

Bindings that reference a constant you didn't override stay at their baked-in default, so a partial override (for example, just `--accent-color`) is fine.

5.5.2. Runtime override

For dynamic theming - in-app accent toggles, branded flavours, A/B tests - push the same constants through `UIManager.addThemeProps` after the theme has been installed:

Values can be passed with or without the leading `#` and in any case; the runtime accepts `"ff2d95"`, `"#FF2D95"`, and the 3-digit shorthand `"#f0a"` interchangeably.

`PaletteOverrideThemeScreenshotTest` in the hellocodenameone test suite exercises this path against both native themes, light + dark.

5.5.3. When the override path doesn't apply

The binding mechanism handles every accent UIID the shipped themes expose. For other widgets - or when you want to override a color the native CSS hard-codes (for example, the iOS `success` green on `Switch.selected`) - either:

- Layer a per-UIID redeclaration in your app's `theme.css` (see `Customizing in your own theme` below). Right when you want to tweak a UIID the binding vocabulary doesn't already cover.
- Or pass the specific UIID/state key directly into `UIManager.addThemeProps` (`Switch.sel#bgColor` etc.). Runtime, bypasses the binding system. Right when you need a one-off color tweak you can't anchor in CSS.

5.6. Platform-specific UIIDs

A handful of UIIDs exist only because one platform draws them differently from the other. They're still safe to override; you just want to know which platform's screen the override is going to land on.

iOS-only behavior:

- `Toolbar`, `TitleArea`, `Title` paint over the status-bar area. iOS reserves room above for the notch / status bar; in your capture this reads as a colored strip at the top of the Form. In production the device fills it with system content (signal, battery, time).
- `MultiButton` is styled as an iOS Settings row (multi-line text ramp + chevron). Compare against Material 3's denser list-item pattern.

Android-only behavior:

- `Toolbar` doesn't paint over the system status bar; the native Android status bar handles that.
- `Tabs` use Material 3 top tabs (flat, underline-by-color). iOS modern renders Tabs as a bottom-anchored pill group via the `tabPlacementInt` constant - that constant is intentionally only set in the iOS theme so behavior stays consistent on each platform.

5.7. Switching CheckBox / RadioButton glyphs

The default check / radio glyphs are Material icons drawn by `DefaultLookAndFeel`. The theme can swap them via four optional constants (Material codepoints):

Constant	Default	Notes
`@checkBoxCheckedIconInt`	`MATERIAL_CHECK_BOX` (`E834`)	Override to for example, `MATERIAL_CHECK_CIRCLE` (`E86C` / `59500`) for an iOS-style filled circle. The iOS modern theme does this.

Constant	Default	Notes
`@checkBoxUncheckedIconInt`	`MATERIAL_CHECK_BOX_OUTLINE_BLANK` (E835)	Match the empty version of whatever you chose above.
`@radioCheckedIconInt`	`MATERIAL_RADIO_BUTTON_CHECKED` (E837)	Filled circle with dot.
`@radioUncheckedIconInt`	`MATERIAL_RADIO_BUTTON_UNCHECKED` (E836)	Empty circle.

Set these in your `#Constants { … }` block; the icons rebuild on the next style refresh.

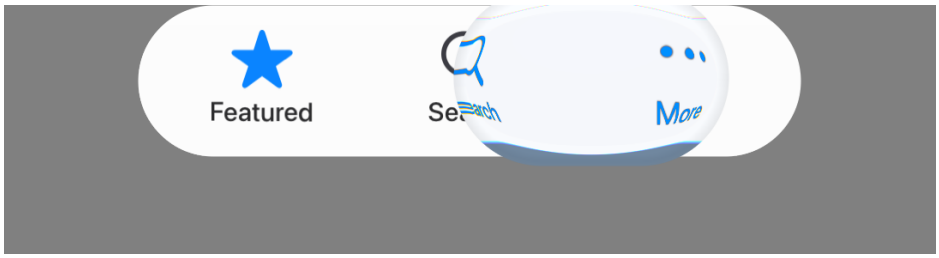
5.8. Component-specific tuning constants

Constant	Effect
`@switchTrackScaleX`, `@switchTrackScaleY`	Stretches the Switch track. Larger X = longer pill, larger Y = thicker. iOS uses 2.5 / 1.5; Material uses 3.0 / 0.9.
`@switchThumbScaleY`	Scales the Switch thumb relative to the track. iOS 1.4, Material 1.5.
`@switchThumbPaddingInt`, `@switchThumbInsetMM`	Pixels of breathing room between thumb and track.
`@switchTrackOffOutlineWidthMM`, `@switchTrackOffOutlineColor`	iOS draws a faint ring around the off-state track; Material does not.
`@tabPlacementInt`	`Component.TOP` (`0`), `Component.BOTTOM` (2), `Component.LEFT` (1), `Component.RIGHT` (3). Where the `Tabs` widget anchors its tab bar. Set in iOS modern theme to 2 for the iOS 26 bottom-bar look; unset on Android (Tabs at top).
`@tabsFillRowsBool`, `@tabsGridBool`	Distribute tabs evenly across the bar.
`@tabsAnimatedIndicatorBool`	`true` paints a thin colored underline below the currently selected tab and tweens its `x` / `width` between tabs on selection change. Both modern themes set this to `true`. Off in the framework default; opt-in for legacy themes by setting it explicitly.
`@tabsAnimatedIndicatorDurationInt`	Duration of the indicator tween in milliseconds (default `200`, matching Material 3's `NavigationBar` spec).
`@tabsAnimatedIndicatorThicknessMm`	Indicator underline thickness in millimetres (default 1).

Constant	Effect
`@pullToRefreshModernBool`	`true` switches the pull-to-refresh visual from the legacy rotating-arrow + text Label stack to a thin circular arc spinner painted directly via `Graphics.drawArc`. The arc sweep grows from 0° to ~330° proportional to the user's pull, then spins continuously once the refresh task fires. Both modern themes set this to `true`. Color comes from the `TabIndicator` UIID's fg (consistent with the animated tab indicator), falling back to the Title fg if unset.
`@pullToRefreshIndicatorDiameterMm`	Outer diameter of the modern arc spinner in millimetres (default 8). Drives the gesture threshold too — the user must pull this distance plus a small margin to fire the refresh task.
`@pullToRefreshIndicatorStrokeMm`	Stroke thickness of the modern arc spinner in millimetres (default `0.6`).
`@darkModeBool`	`true` enables `$DarkUIID` resolution when the app is in dark mode. Both modern themes set this; user themes that want dark mode also need to set it.

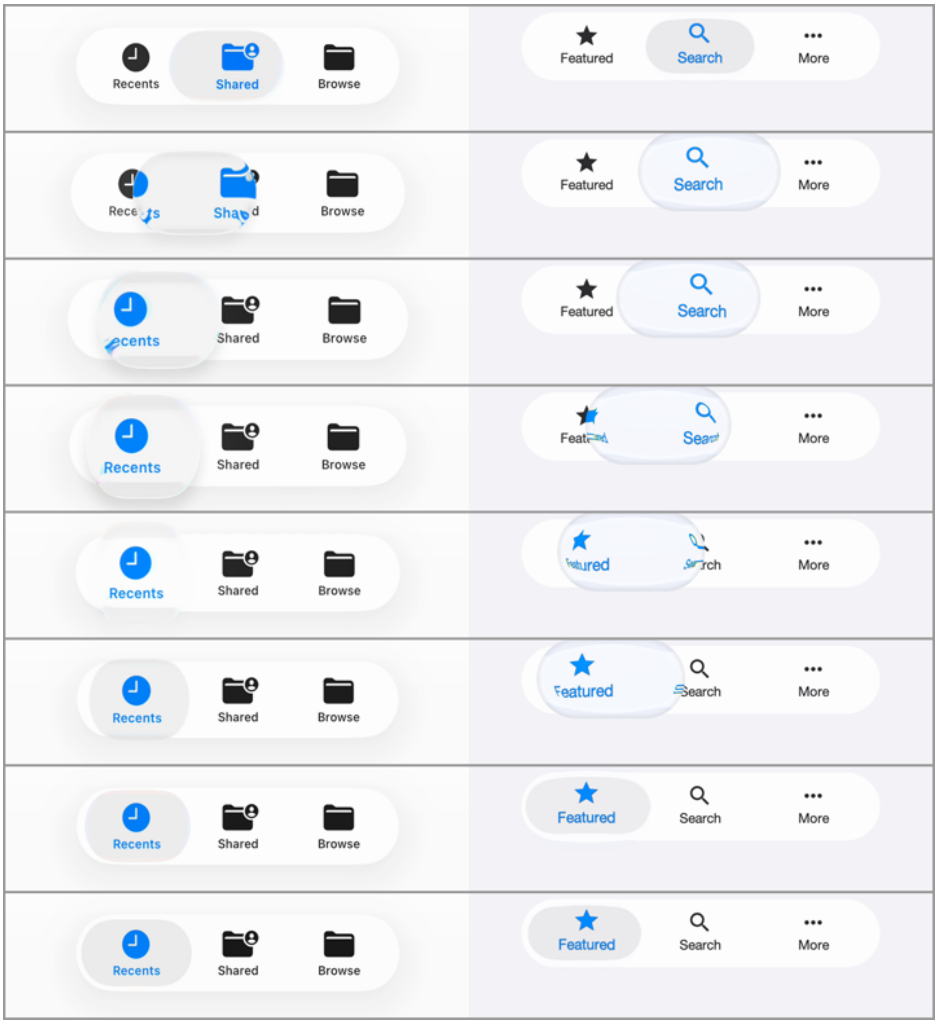
5.9. iOS 26 Liquid Glass tab selection morph

When the iOS modern theme renders `Tabs` as a bottom bar (`tabPlacementInt: 2`) the selected tab is marked by a **Liquid Glass drop** — a frosted, magnifying glass capsule painted over the (dark) icons rather than a flat highlight. On a selection change the drop springs across to the new tab, elongating while it travels and settling with a small overshoot. The selected blue is supplied by the drop itself (a luminance-keyed dark→accent tint of the glyph beneath it), so the color travels with the drop instead of snapping between tabs.



On iOS the drop is rendered live by a Metal fragment shader that samples the bar beneath it on the GPU, so the morph runs at frame rate with no screen read-back; other platforms and the simulator use the equivalent CPU lens. The effect is opt-in: the iOS modern theme sets `tabsSelectionCapsuleBool: true` and `glassMaterialBool: true`. Without the glass flag the capsule degrades to a plain translucent pill.

The morph is developed against the real OS control: the frame-by-frame comparison below shows the native iOS 26 tab bar (left) and the Codename One render (right) at matching moments of the selection travel.



The animation is controlled by a **named motion preset** plus a handful of high-level knobs. The detailed envelopes (stretch, squash, lift, lens magnification, aberration, tint timing) live inside the preset in the motion model (`TabSelectionMorph`), where they're pinned by unit test and validated frame-by-frame by the fidelity suite — a theme picks a preset and scales it rather than tuning a dozen loose constants:

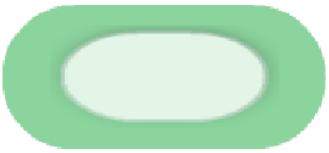
Constant	iOS modern	Effect
`@tabsSelectionCapsuleBool`	`true`	Master switch for the selection drop. Off falls back to the legacy colored underline indicator (`tabsAnimatedIndicatorBool`).
`@tabsMorphPreset`	`ios26`	The named motion preset: `ios26` (the measured Liquid Glass morph) or `subtle` (half the deformation and optics, for a calmer selection change).
`@tabsAnimatedIndicatorDurationInt`	`480`	Morph duration in milliseconds. Lower is snappier.
`@tabsMorphLensIntensityPct`	`100`	Scales the drop's **optics** (magnification, chromatic aberration, accent tint) around the preset: `0` is an optically flat drop, `200` doubles the lens strength. Geometry is unaffected.

Constant	iOS modern	Effect
`@tabsMorphSpringPct`	`100`	Scales the settle overshoot: `0` stops dead on a plain ease-out, `200` doubles the bounce.
`@tabSelLensTintColorInt`	`0x0a84ff`	The accent the dark glyph is tinted to inside the drop.

Similarly, the **Liquid Glass backdrop materials** themselves are typed, named recipes (`GlassRecipe`): a theme assigns `chrome` (edge-anchored bars), `pill` (the floating tab bar) or `panel` (buttons, dialogs, panels — the default) per UIID with `<UIID>GlassRecipe` constants, e.g. `ToolbarGlassRecipe: "chrome"`. Each recipe bundles the measured saturation/scale/offset color transform and edge optics of one native material, so similar glass surfaces can't drift apart one constant at a time.



The drop's glass cues (rim shrink, refraction, glare, edge shadow) fade out as it settles, so a resting selection reads as a calm frosted pill rather than a distorting lens. Every motion preset is validated deterministically: the fidelity suite freezes the morph at the fixed progress points 0, 10, 25, 50, 75, 90 and 100 percent, regression-compares each frame, and pins the same points numerically against the motion model in unit tests. The `Switch` thumb uses the same discipline for its liquid **droplet** morph (`switchLiquidGlassBool`, stretch/squash while sliding, frame-validated):



5.10. Customizing in your own theme

Your app's `theme.css` inherits from the installed native theme:

```
#Constants {
    includeNativeBool: true;
    darkModeBool: true;
}

/* Tweak only what's different. Everything you do not redeclare
   keeps coming from the native theme. */
RaisedButton { background-color: #d81b60; }
RaisedButton.pressed { background-color: #b71c5c; }
RaisedButton.disabled { background-color: #ffd6e2; color: #ffffff; }

@media (prefers-color-scheme: dark) {
    RaisedButton { background-color: #ff80ab; color: #4a0026; }
    RaisedButton.pressed { background-color: #f06292; }
```

```
}
```

The user's CSS is layered on top of the native theme at app launch, so refresh / restart picks the override up.

For a wholesale accent rebrand prefer redeclaring the `--accent-color` (etc.) variables in your `#Constants` block - see `Override from your app's theme.css` above. That single declaration fans out through every UIID bound to the variable in the native theme, no per-UIID rules required. Per-UIID redeclaration here remains the right choice for tweaking non-accent properties (typography, spacing, surface fills) the binding vocabulary doesn't cover.

5.11. Inheriting from a native UIID

`cn1-derive` lets a custom UIID start from one of the native theme's UIIDs and refine it:

```
DangerButton {
    cn1-derive: RaisedButton;
    background-color: #d32f2f;
}
DangerButton.pressed {
    cn1-derive: RaisedButton;
    background-color: #b71c1c;
}
```

`cn1-derive` works best when the relationship is **child refines parent**. Avoid deriving across unrelated UIIDs (for example, a TitleArea that derives from Toolbar) - inline the properties instead.

5.12. Translucency, glass effects, and the test harness

The iOS modern Dialog and Tabs use translucent surfaces (rgba with alpha < 1) so any backdrop reads through the widget. To exercise this in screenshot tests, the hellocodenameone fidelity tests opt-in to a diagonal-stripe textured backdrop on the form (`useTexturedBackdrop()` in `DualAppearanceBaseTest`). Translucent widgets show their see-through tint against the stripes; opaque widgets cover the stripes entirely.

CSS `backdrop-filter: blur(<length>)` and `filter: blur(<length>)` are now recognized by the compiler and persisted in the theme as the `backdropFilterBlur` and `filterBlur` Style properties (also reachable programmatically via `Style#getBackdropFilterBlurRadius()` / `Style#getFilterBlurRadius()`). Image-level blur is hardware-accelerated (`CIGaussianBlur` on iOS, `RenderScript` / `RenderEffect` on Android, JHLabs `GaussianFilter` in the simulator). Painting the blur into the component framebuffer is still being wired up; until that lands, the rgba approximation above remains useful for translucent surfaces.

Chapter 6. CSS

In this chapter this guide covers theming with CSS in Codename One.



CSS Changes don't Require a Recompile

You can change the CSS values while the simulator is running and the changes will reflect in the simulator within a few seconds

6.1. Activating CSS

Codename One applications always use the resource file. The CSS support compiles a file in CSS syntax to a Codename One resource file and adds it to the application. In runtime the CSS no longer exists and the file acts like a regular theme file.

New Maven projects enable CSS by default with `codename1.cssTheme=true` in `common/codenameone_settings.properties`. Add that property manually if you are migrating an older project where CSS support is disabled.

Once enabled your `theme.res` file will regenerate from a CSS file that resides under the `css` directory. Changes you make to the CSS file will instantly update the simulator as you save. But, there are some limits to this live update so sometimes a simulator restart would be necessary.

6.2. Supported CSS selectors

Since Codename One stylesheets are meant to be used with Codename One component hierarchies instead of XML/HTML documents, selectors work a little differently.

1. All selectors (with some specific exceptions discussed below) are interpreted as UIIDs.
2. Only 4 predefined CSS classes are supported:
   - `.pressed`: Targets the component when in "Pressed" state.
   - `.selected`: Targets the component when in "Selected" state.
   - `.unselected`: Targets the component when in "Unselected" state.
   - `.disabled`: Targets the component when in "Disabled" state.

   If no class is specified, then the selector targets "all" states of the given component.

The following are a few possible selectors you can include in your stylesheet.

1. `Button`: Defines styles for the "Button" UIID.
2. `Button.pressed`: Defines styles for the "Button" UIID's "pressed" state.
3. `Button, TextField, Form`: Defines styles for the `Button`, `TextField`, and "Form" UIIDs.

The following example creates a simple button with a border, and text aligned center. By default the button will have a transparent background, but when it's pressed, it will have a gray background:

```
Button {
    text-align: center;
    border: 1pt solid gray;
    background-color: transparent;
}

Button.pressed {
    background-color: gray;
}
```

6.2.1. Inheriting properties using `cn1-derive`

The following example defines a custom Button style named "MyButton" that inherits all the styles of Button but changes the background color to blue:

```
MyButton {
   cn1-derive: Button;
   background-color: blue;
}
```

6.3. Special selectors

6.3.1. `#device`

The `#Device` selector allows you to define which device resolutions this CSS file should target. Mutli-images generated from this style-sheet will be included variants for device resolutions in the range `(min-resolution, max-resolution)` as defined in this section. By default all resolutions are generated:

```
#Device {
    min-resolution: 120dpi;
    max-resolution: 480dpi;
    resolution: 480dpi;
}
```

6.3.2. `#constants`

The `#Constants` selector allows you to specify theme constants.

For example:

```
#Constants {
    includeNativeBool: true;
    commandBehavior: "Side";
    defaultFontSizeInt: 18;
```

```
    defaultDesktopFontSizeInt: 14;
    defaultSourceDPIInt: "0";
    menuImage: "menu.png";
}

MenuIcon {
    background-image: url(images/menu.png);
}
```

In the above example, the constants referring to an image name as a string requires that the image exists in one of the following locations:

- `res/<cssfilename>/<imageName>`
- `../res/<cssfilename>/<imageName>`
- `../../res/<cssfilename>/<imageName>`

It must also have been defined as a background image in some selector in this CSS file.

6.3.3. `Default`

The `Default` selector is special in that it will set properties on the theme's "default" element. The default element is a special UIID in Codename One from which all other UIIDs in the same theme are derived. This is a good place to set things like default fonts or background-colors.

6.4. Standard CSS properties

- `padding` (and variants)
- `margin` (and variants)
- `border` (and variants)
- `border-radius`
- `background` (Usage below)
- `background-color`
- `background-repeat`
- `background-image`
- `border-image`
- `border-image-slice`
- `font` (Usage is covered in the following font section)
- `font-family` (Usage is covered in the following font section)
- `font-style` (Usage is covered in the following font section)
- `font-size` (Usage is covered in the following font section)
- `@font-face` (Usage is covered in the following font section)

- `color`
- `text-align`
- `text-decoration`(Usage below)
- `opacity`
- `box-shadow`
- `width` (used for generating background-images and borders)
- `height` (used for generating background-images and borders)

6.5. Custom properties

`cn1-source-dpi`

Used to specify source DPI for multi-image generation of background images. Accepted values: `0` (Not multi-image), `120` (Low res), `160` (Medium Res), `320` (Very High Res), `480` (HD), Higher than `480` (2HD). If not specified, the default value will be the value of the `defaultSourceDPIInt` theme constant, if specified, or `480`, if not specified.

`cn1-background-type`

Used to explicitly specify the background-type that should be used for the class.

`cn1-9patch`

Used to explicitly specify the slices used when generating 9-piece borders. **Deprecated - Use `border-image` and `border-image-slice` for 9-piece borders.**

`cn1-derive`

Used to specify that this UIID should derive from an existing UIID.

6.6. CSS variables

As of CodenameOne 7.0, you can use variables in your CSS file via the `var()` CSS function. For example:

```
#Constants {
    --header-color: blue;
}

Header {
    color: var(--header-color, blue);
}
```

The `var()` function can be used inside property values. That is, You can't use it in property names or selectors.

The `var()` grammar follows the browser CSS form:

```
#Constants {
    --custom-property-name: #0066cc;
}

ExampleSelector {
    color: var(--custom-property-name, #0066cc);
}
```

The `<custom-property-name>` must begin with two dashes (`--`).

The `<declaration-value>` is the fallback value that will be used if the variable hasn't been defined in the CSS file. The fallback value may include commas.

Examples

Listing 5. Example defining and using a CSS variable

```
#Constants {
    --main-bg-color: red;
}

MyContainer {
    background-color: var(--main-bg-color);
}
```

Listing 6. Example using a fallback value

```
#Constants {
    --main-bg-color: red;
}

MyContainer {
    background-color: var(--main-bg-color, blue);
}
```

See the MDN docs [https://developer.mozilla.org/en-You/docs/Web/CSS/var] for more details about the CSS variable spec.

6.7. CSS properties

This section isn't as comprehensive as it should be due to the breadth of CSS.

6.7.1. Text-decoration

underline	Underlines text. For example, `text-decoration: underline;`

overline	Overlines text. For example, `text-decoration: overline;`
line-through	Strikes through text. For example, `text-decoration: line-through;`
none	No text decoration. For example, `text-decoration: none;`
cn1-3d	3D text. For example, `text-decoraton: cn1-3d;` The Quick Brown Fox Jumped
cn1-3d-lowered	3D lowered text. For example, `text-decoration: cn1-3d-lowered;` The Quick Brown Fox Jumped
cn1-3d-shadow-north	3D text with north shadow. For example, `text-decoration: cn1-3d-shadow-north;` The Quick Brown Fox Jumped

For other CSS font settings see the Fonts section [Fonts]

6.7.2. Border

This library supports the border property [https://developer.mozilla.org/en-You/docs/Web/CSS/border] and most of its variants (for example, border-width [https://developer.mozilla.org/en-You/docs/Web/CSS/border-width], border-style [https://developer.mozilla.org/en-You/docs/Web/CSS/border-style], and border-color [https://developer.mozilla.org/en-You/docs/Web/CSS/border-color]. It will try to use native Codename One styles for generating borders if possible. If the border definition is too complex, it will fall-back to generating a 9-piece image border at compile-time. This has the effect of making the resulting resource file larger, but will produce good runtime performance, and a look that's faithful to the provided CSS.

The algorithm used to determine whether to use a native border or to generate a 9-piece image, is complex, but the following guidelines may help you if you wish to design borders that can be rendered natively in CN1:

- Non-pixel units `border-width`. (Except with the `cn1-round-border` and `cn1-pill-border` styles)
- Using the `border-radius` directive.
- Using `box-shadow` (Unless using `cn1-round-border` or `cn1-pill-border` styles)
- Using a background gradient in combination with a border or any kind
- Using a different `border-width`, `border-style`, or `border-color` for different sides of the border
- Using a `filter`



You can open the resulting theme file in the designer and inspect it to see if an image was generated

Generating the image triggers slower CSS compilation and a larger binary, so tune the CSS so it avoids this fallback.

Round borders

Rounded borders can be achieved in a few different ways. The easiest methods are:

- **The `cn1-round-border` style**. This will render a circular round border in the background **natively**. That is, this doesn't require generation of an image border
- **The `cn1-pill-border` style**. This will render a pill-shaped border in the background natively. This also doesn't require generation of an image border
- **The `border-radius` property**. This will round the corners of the border. If the style can be achieved using the `RoundRectBorder` in CodenameOne, then it will use that border. If not, this will cause the style to be generated as an image border

A border generated from `border-radius` follows the CSS box model, so the radius rounds the component without changing its size. A radius larger than the component scales down to fit it. This differs from a `RoundRectBorder` you create in Java, which grows the component to twice the radius so the corners are always drawn in full. Use `cn1-pill-border` when a shape has to stay a pill at any height.

Examples using `cn1-round-border`

```
RoundBorder {
    border: 1px #3399ff cn1-round-border;
    text-align: center;
    margin: 2mm;
    padding: 3mm;
}

RoundBorderFilled {
    background: cn1-round-border;
    background-color: #cccccc;
    text-align: center;
    margin: 2mm;
    padding: 3mm;
}
```

Examples using `cn1-pill-border`

```
PillBorder {
    border: 1pt #3399ff cn1-pill-border;
    text-align: center;
}

PillBorderFilled {
    background: cn1-pill-border;
    background-color: #3399ff;
    color: white;
    text-align: center;
}
```

Examples using `border-radius`

```
RoundRectLabel {
    background-color: red;
    border-radius: 2mm;
}
```

`cn1-pill-border` and `cn1-round-border` don't support the standard CSS `box-shadow` property. This is because the `box-shadow` property parameters don't map onto the shadow parameters for the Codename One `RoundBorder` class. To get shadows on the `cn1-pill-border`, you should use one or more of the following CSS properties:

- `cn1-box-shadow-spread`: Accepts values in any scalar unit (for example, px, mm, cm, etc.). This maps directly to the border's shadowSpread [https://www.codenameone.com/javadoc/com/codename1/ui/plaf/RoundBorder.html#shadowSpread-int-boolean-] property.
- `cn1-box-shadow-h`: Accepts values in real values or integers (not a scalar unit). This maps directly to the border's shadowX [https://www.codenameone.com/javadoc/com/codename1/ui/plaf/RoundBorder.html#shadowX-float-] property.
- `cn1-box-shadow-v`: Accepts values in real values or integers (not a scalar unit). This maps directly to the border's shadowY [https://www.codenameone.com/javadoc/com/codename1/ui/plaf/RoundBorder.html#shadowY-float-] property.
- `cn1-box-shadow-blur`: Scalar value. Maps to the border's shadowBlur [https://www.codenameone.com/javadoc/com/codename1/ui/plaf/RoundBorder.html#shadowBlur-float-] property.
- `cn1-box-shadow-color`: The shadow color
- `cn1-box-shadow-inset`: Set to `inset` to render an inner shadow instead of the default outer shadow spread.

Using the regular CSS `box-shadow` in conjunction with `border-radius` will cause a 9-piece border to be generated rather than mapping to the `RoundRectBorder`. If, but, you use the `cn1-box-*` properties for the shadow instead, it will use the RoundRectBorder—assuming that no other styles are specified that trigger an image border to be generated.

Codename One also exposes per-corner elliptical radius controls that map to the `RoundBorder`'s X/Y radii. You can set them directly with `cn1-border-top-left-radius-x / cn1-border-top-left-radius-y` (and the equivalent `top-right`, `bottom-left`, and `bottom-right` pairs) to fine tune horizontal and vertical curvature independently. The CSS parser automatically populates these properties when you use standard `border-radius` syntax, including the longhand declarations and the `border-radius: <x-radii> / <y-radii>` shorthand:

```
EllipticalBorder {
    border-radius: 2mm 4mm 6mm 1mm / 1mm 3mm 5mm 7mm;
    cn1-box-shadow-spread: 1.5mm;
    cn1-box-shadow-inset: inset;
}
```

In the example above, the four horizontal radii (`2mm 4mm 6mm 1mm`) populate the `cn1-border--radius-x` properties clockwise from the top-left corner. The four values after the slash fill the matching `cn1-border--radius-y` entries. Setting `cn1-box-shadow-inset: inset;` converts the shadow into an inset glow that follows the same elliptical curvature.

6.7.3. Background

The `background` property supports most standard CSS values [https://developer.mozilla.org/en-You/docs/Web/CSS/background] for setting the background color, or background image.



9-piece Image borders always take precedence over background settings in Codename One. If your background directive seems to have no effect, it's likely because the theme has specified a 9-piece image border for the UIID. You can disable the image border using a directive like `border: none`

Background images

See Images [Images]

Gradients

The full CSS gradient range is natively supported: `linear-gradient`, `radial-gradient`, `conic-gradient`, `repeating-linear-gradient`, and `repeating-radial-gradient` — all with arbitrary angles, unlimited multi-stop colors (with optional position hints), and full radial shape/extent control. The compiled theme resource file carries a compact descriptor and the gradient is rendered at runtime by the platform-native graphics API (Java2D on JavaSE, `LinearGradient/RadialGradient/SweepGradient` on Android, Core Graphics / Core Image on iOS).

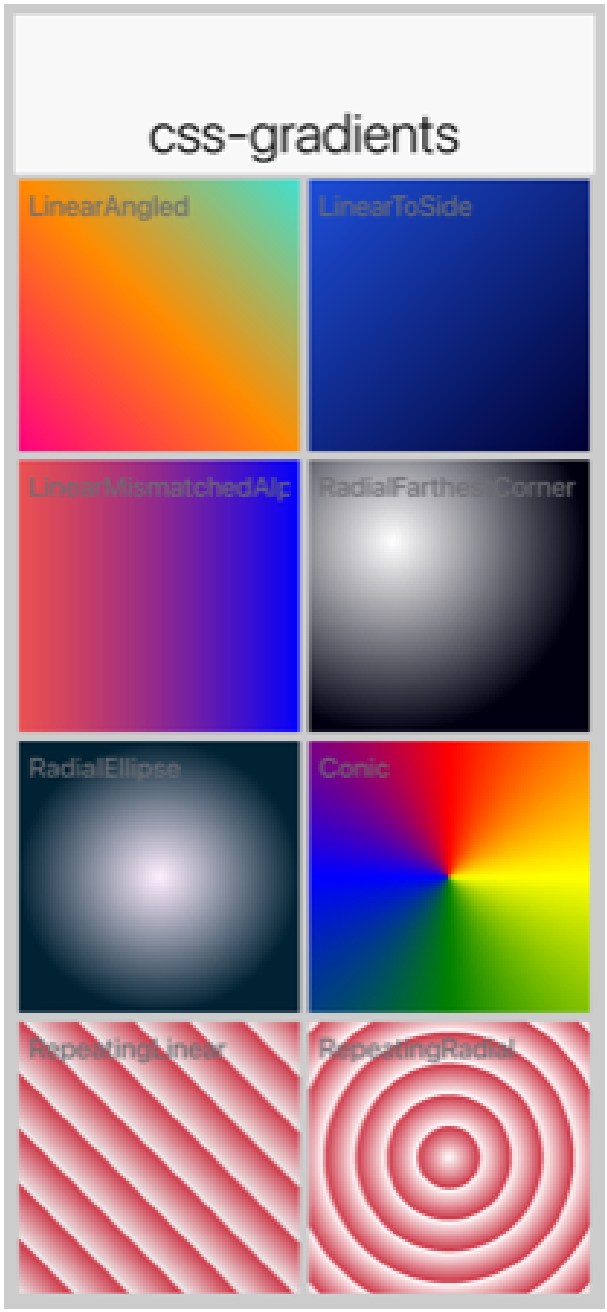


Figure 95. CSS gradient functions rendered side-by-side from the framework's screenshot test (top to bottom: linear at an arbitrary angle, linear `to <side>`*, linear with mismatched alphas, radial* `farthest-corner`*, radial ellipse, conic, repeating-linear, repeating-radial).*

`linear-gradient`

Any angle in degrees, radians, or `turn`, or the canonical `to <side>` / `to <side1> <side2>` directions. Two or more stops, each with optional position percentage; positions left blank between fixed anchors are autodistributed:

```
LinearGradientA { background: linear-gradient(0deg, #cccccc, #666666); }
LinearGradientB { background: linear-gradient(to top, #cccccc, #666666); }
LinearGradientC { background: linear-gradient(45deg, #eaeaea, #666666); }
LinearGradientD { background: linear-gradient(135deg, #ff0080 0%, #ff8c00 50%, #40e0d0 100%); }
LinearGradientE { background: linear-gradient(to bottom right, #ff0066, #000033); }
LinearGradientF { background: linear-gradient(90deg, rgba(255, 0, 0, 0.6), blue); }
```

`radial-gradient`

Full CSS radial syntax: `circle` or `ellipse`, any of the four extent keywords (`closest-side` / `closest-corner` / `farthest-side` / `farthest-corner`, defaulting to `farthest-corner`), explicit radii as percentages, an optional `at <position>` clause with side keywords or percentages, and two or more multi-stop colors:

```
RadialGradientA { background: radial-gradient(circle, gray, white); }
RadialGradientB { background: radial-gradient(ellipse closest-side at 25% 75%, #ffffff, #000000); }
RadialGradientC { background: radial-gradient(circle farthest-corner at right, #ffffee, #006666 60%, #000011 100%); }
```

`conic-gradient`

Sweep / pie-style gradient. Optional `from <angle>` (CSS convention: 0° points up, sweep is clockwise) and `at <position>` prefix, followed by the stop list:

```
ConicGradientA { background: conic-gradient(red, yellow, green, blue, red); }
ConicGradientB { background: conic-gradient(from 45deg at 50% 50%, #ff0066 0%, #ffcc00 25%, #00cc66 50%, #0066ff 75%,
#ff0066 100%); }
```

`repeating-linear-gradient` / `repeating-radial-gradient`

Identical syntax to their non-repeating counterparts. The stop pattern tiles outward to fill the bounding box, ideal for stripes and rings:

```
MyStripe { background: repeating-linear-gradient(45deg, #eeeeee 0%, #eeeeee 10%, #cccccc 10%, #cccccc 20%); }
MyTarget { background: repeating-radial-gradient(circle at center, #ffffff 0%, #ffffff 20%, #cc3333 20%, #cc3333 40%); }
```

filter and backdrop-filter

CSS `filter` and `backdrop-filter` accept a chain of functions. Two storage forms land on the corresponding `Style`:

- `blur(<length>)` → `filterBlurRadius` / `backdropFilterBlurRadius` (a single pixel value).
- `brightness`, `contrast`, `grayscale`, `hue-rotate`, `invert`, `opacity`, `saturate`, `sepia` → `filterColorMatrix` / `backdropFilterColorMatrix` (a 4×5 color matrix; a multi-function chain composes into a single matrix in CSS order):

```
MyOverlay {
    background: rgba(0, 0, 0, 0.4);
    backdrop-filter: blur(12px);
}

MyBlurredImage {
    filter: blur(4px);
}

MyFaded {
```

```
    filter: brightness(0.7) contrast(1.15);
}

MyGrayscale {
    filter: grayscale(1);
}

MySepia {
    filter: sepia(0.8) saturate(1.1);
}
```

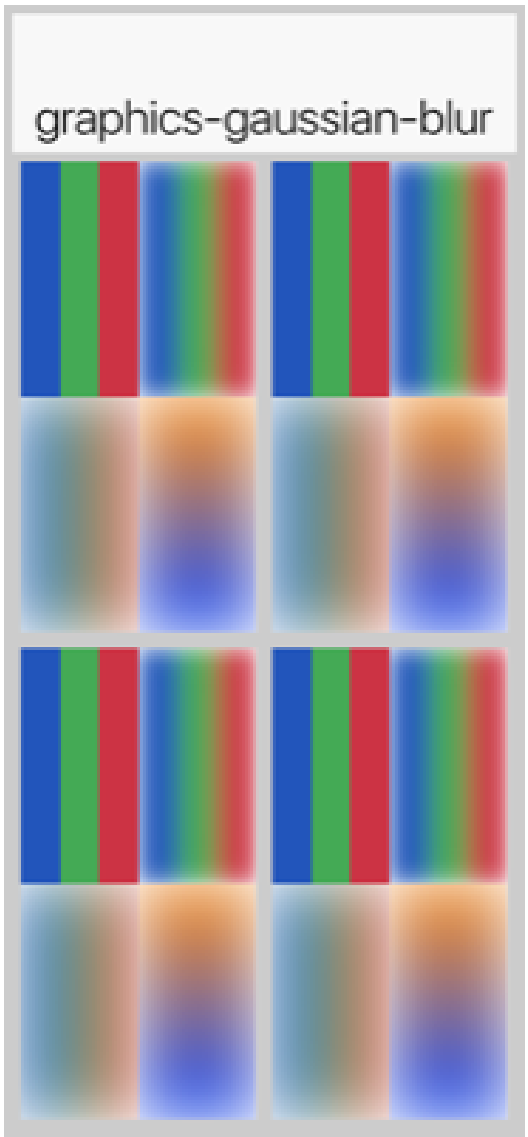


Figure 96. `filter: blur()` applied at a graphics primitive level — the framework's screenshot test renders RGB stripes and a gradient, then blurs both via `Graphics.gaussianBlur(⋯)`. Component-level paint-time integration of `filter:` declarations is in progress; until it lands, set the radius / matrix on a `Style` and consume them via the corresponding `Graphics` primitives manually.

`filter` applies to the component's own painted content; `backdrop-filter` applies to whatever is painted behind. The radii / matrices are also exposed on `Style` (`getFilterBlurRadius()`, `getFilterColorMatrix()`, `getBackdropFilterBlurRadius()`, `getBackdropFilterColorMatrix()`) so they can be set programmatically. Hardware blur is used where available (Core Image on iOS, RenderScript/RenderEffect on Android, JHLabs `GaussianFilter` on JavaSE simulator); ports without

a fast path fall back to a software Gaussian.

6.7.4. cn1-background-type

It also supports some special Codename One values, which are identifiers with a "cn1-" prefix. The following special values are available. They map to the standard Codename One values you discussed in the theming chapter:

- `cn1-image-scaled`
- `cn1-image-scaled-fill`
- `cn1-image-scaled-fit`
- `cn1-image-tile-both`
- `cn1-image-tile-valign-left`
- `cn1-image-tile-valign-center`
- `cn1-image-tile-valign-right`
- `cn1-image-tile-halign-top`
- `cn1-image-tile-halign-center`
- `cn1-image-tile-halign-bottom`
- `cn1-image-align-bottom`
- `cn1-image-align-left`
- `cn1-image-align-right`
- `cn1-image-align-center`
- `cn1-image-align-top-left`
- `cn1-image-align-top-right`
- `cn1-image-align-bottom-left`
- `cn1-image-align-bottom-right`
- `cn1-image-border`
- `cn1-none`
- `cn1-round-border`
- `cn1-pill-border`

6.8. Images

Images are supported as both "inputs" of the stylesheet, and as outputs to the compiled resource file. "Input" images are specified via the `background-image` property in a selector. "Output" images are always saved as multi-images inside the resource file.

6.8.1. Image DPI and device densities

To appropriately size the image, the CSS compiler needs to know what the source density of the

image is. For example, if an image is 160×160 pixels with a source density of 160dpi (that is, medium density - or the same as an iPhone 3G), then the resulting multi-image will be sized at 160×160 for medium density devices and 320×320 on high density devices (for example, iPhone 4S Retina) - which will result in the same perceived size to the user of 1×1 inch.

However, if the image has a source density of 320dpi, then the resulting multi-image would be 80×80 pixels on medium density devices and 160×160 pixels on high density devices.

Some images have this density information embedded in the image itself so that the CSS processor will know how to resize the image. But, it's better to explicitly document your intentions by including the `cn1-source-dpi` property as follows:

```
SomeStyle {
    background-image: url(images/my-image.png);
    cn1-source-dpi: 160;
}
```



`cn1-source-dpi` values are meant to fall into threshold ranges. Values less than or equal to 120, are interpreted as low density. 121 - 160 are medium density (iPhone 3GS). 161 - 320, high density (iPhone 4S). 321 - 480 == HD. 481 and higher == 2HD. In general, you should try to use images that are one of these DPIs exactly: 160, 320, or 480, then images will be scaled up or down to the other densities accordingly.

6.8.2. Multi-Images vs regular images

By default all images are imported as multi-images (unless you define the `defaultSourceDPIInt` theme constant). If you want to import an image as a "regular" image, you can set `cn1-source-dpi` to `0`. For example:

```
SomeStyle {
    background-image: url(images/my-image.png);
    cn1-source-dpi: 0;
}
```



You can change the default source DPI for the whole stylesheet by adding `defaultSourceDPIInt: 0` to the theme constants. For example:

```
#Constants {
  defaultSourceDPIInt: 0;
}
```

Most application templates in Codename One Initializr [https://www.codenameone.com/initializr] include this constant by default.

6.8.3. Multi-Images as inputs

If you've already generated images in all the appropriate sizes for all densities, you can provide them in the same file structure used by the Codename One XML resource files: The image path is a directory that contains images named after the density that they're intended to be used for. The possible names include:

- `verylow.png`
- `low.png`
- `medium.png`
- `high.png`
- `veryhigh.png`
- `560.png`
- `hd.png`
- `2hd.png`
- `4k.png`

For example, Given the CSS directives:

```
MyStyle {
    background-image: url(images/mymultiimage.png);
}
```

The files would look like:

```
css/
 +--- mycssfile.css
 +--- images/
       +--- mymultiimage.png/
             +--- verylow.png
             +--- low.png
             +--- medium.png
              ... etc.
```



Multi-image inputs are supported for local URLs. You can't use remote (for example, `http://`) urls with multi-image inputs

6.8.4. Image constants

Theme constants can be images. The convention is to suffix the constant name with "Image" so that it will be treated as an image. Also to the standard `url()` notation for specifying a constant image, you can provide a simple string name of the image, and the CSS processor will try to find an image by that name specified as a background image for one of the styles. If it can't find one, it will look inside a special directory named "res" (located in the same directory as the CSS stylesheet), inside

which it will look for a directory named the same as the stylesheet, inside which it will look for a directory with the specified multi-image. This directory structure is the same as used for Codename One's XML resources directory.

For example, In the CSS file "mycssfile.css":

```
radioSelectedFocusImage: "Radio_btn_Press.png";
```

Will look for a directory located at `res/mycssfile.css/Radio_btn_Press.png/` with the following images:

- `verylow.png`
- `low.png`
- `medium.png`
- `high.png`
- `veryhigh.png`
- `560.png`
- `hd.png`
- `2hd.png`
- `4k.png`

It will then create a multi-image from these images and include them in the resource file.

6.9. Image recipes

6.9.1. Import multiple images in single selector

It's quite useful to be able to embed images inside the resource file that's generated from the CSS stylesheet so that you can access the images using the `Resources.getImage()` method in your app and set it as an icon on a button or label. In this case, it's easier to create a dummy style that you don't intend to use and include multiple images in the background-image property like so:

```
Images {
    background-image: url(images/NowLogo.png),
        url(images/Username-icon.png),
        url(images/Password-icon.png),
        url(images/Name-icon.png),
        url(images/Email-icon.png),
        url(images/SeaIce.png),
        url(images/Back-icon.png),
        url(images/Source-icon.png),
        url(images/Date-icon.png),
        url(images/Arrow-right.png),
        url(images/Share-icon.png),
```

```
        url(images/Text-icon.png),
        url(images/Comments-icon.png),
        url(images/RedPlanet.png),
        url(images/News-icon.png),
        url(images/Channels-icon.png),
        url(images/Bookmarks-icon.png),
        url(images/Overview-icon.png),
        url(images/Calendar-icon.png),
        url(images/Timeline-icon.png),
        url(images/Profile-icon.png),
        url(images/Widgets-icon.png),
        url(images/Settings-icon.png),
        url(images/Bookmark-icon.png);
}
```

Then in Java, you might do something like:

```
Resources theme = Resources.openLayered("/theme");

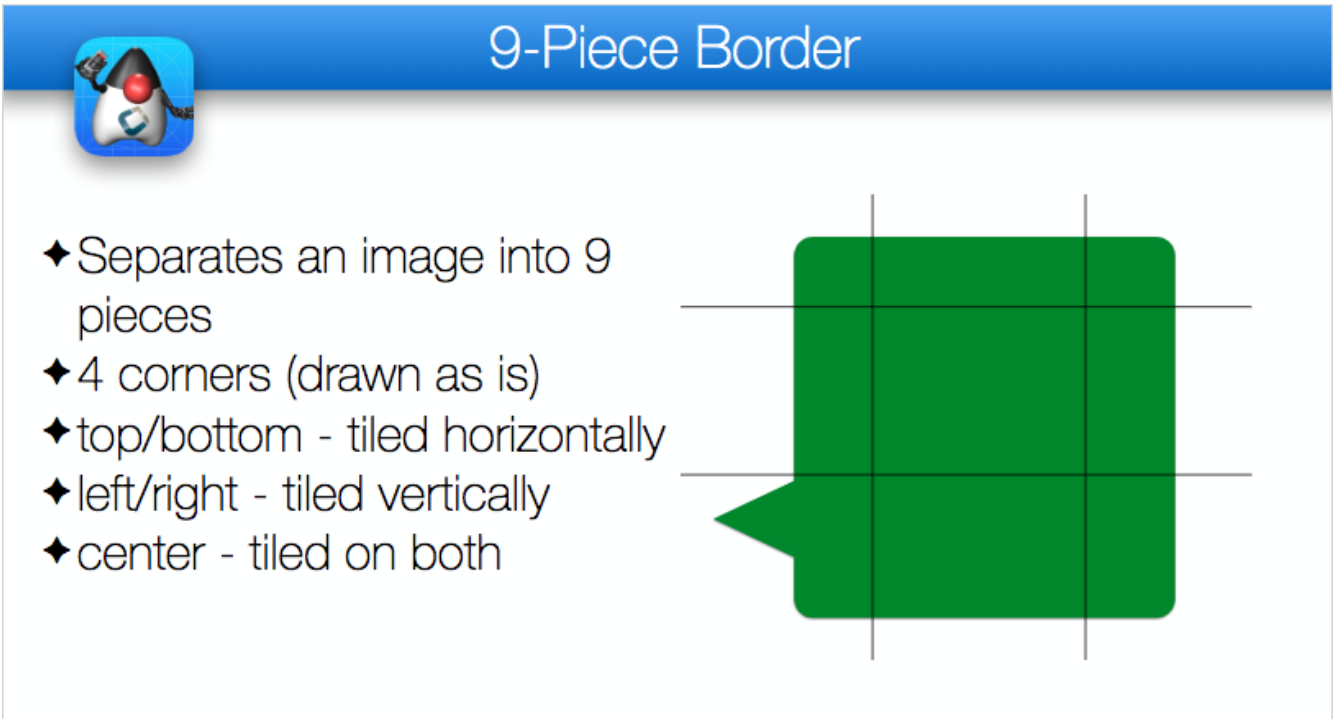
Label bookmark = new Label(theme.getImage("Bookmark-icon.png"));
```

6.9.2. Loading images from URLs

You can also load images from remote URLs. The compiled guide fixture uses a local asset so the build is deterministic and doesn't depend on network access; application CSS can use the same `url(…)` syntax with an HTTPS URL:

```
ImagesRemote {
    background-image: url(images/remote-logo.png);
}
```

6.9.3. Generating 9-Piece Image borders



9-Piece image borders can be created using the `image-border` and `image-border-slice` properties.

For example:

```
NinePiece {
    border-image: url(images/dashbg_landscape.png);
}
```

In the above example you omitted the `border-image-slice` property, so it defaults to "40%," which means that the image is sliced 40% from the top, 40% from the bottom, 40% from the left, and 40% from the right.

If you want more specific "slice" points, you can add the `border-image-slice` property. For example:

```
NinePiece {
    border-image: url(images/dashbg_landscape.png);
    border-image-slice: 10% 49%;
}

NinePiece2 {
    border-image: url(images/dashbg_landscape.png);
    border-image-slice: 10% 49% 20%;
}

NinePiece3 {
    border-image: url(images/dashbg_landscape.png);
    border-image-slice: 10% 30% 40% 20%;
}

NinePiece4 {
```

```
    border-image: url(images/dashbg_landscape.png);
    border-image-slice: 10%;
}
```

6.9.4. Image backgrounds

Component backgrounds in Codename One are a common source of confusion for newcomers because there are 3 different properties that can be used to define what a component's background looks like, and they have priorities:

1. Background Color - You can specify an RGB color to be used as the background for a component.
2. Background Image - You can specify an image to be used as the background for a component. Codename One includes settings to define how the image is treated, for example, scale/fill, tile, etc. If a background image is specified, it will override the background color setting - unless the image has transparent regions.
3. Image Border - You can define a 9-piece image border which will effectively cover the entire background of the component. If an image border is specified, it will override the background image of the component.

A common scenario you may run into is trying to set the background color of a component and seeing no change when you preview the form, because the style had an image background defined - which overrides the background color change.

The potential for confusion is mitigated somewhat, but still exists when using CSS. You can make your intentions explicit by adding the `cn1-background-type` property to your style. Possible values include:

- `cn1-image-scaled`
- `cn1-image-scaled-fill`
- `cn1-image-scaled-fit`
- `cn1-image-tile-both`
- `cn1-image-tile-valign-left`
- `cn1-image-tile-valign-center`
- `cn1-image-tile-valign-right`
- `cn1-image-tile-halign-top`
- `cn1-image-tile-halign-center`
- `cn1-image-tile-halign-bottom`
- `cn1-image-align-bottom`
- `cn1-image-align-left`
- `cn1-image-align-right`
- `cn1-image-align-center`
- `cn1-image-align-top-left`

- cn1-image-align-top-right
- cn1-image-align-bottom-left
- cn1-image-align-bottom-right
- cn1-image-border
- cn1-none
- none

Example setting background Image to scale fill

```
MyContainer {
    background-image: url(images/myimage.png);
    cn1-background-type: cn1-image-scaled-fill;
}
```

6.10. Image compression

CN1 resource files support both PNG and JPEG images, but PNG is the default. Multi-images that are generated by the CSS compiler will be PNG if they include alpha transparency, and JPEG otherwise. This is to try to reduce the file size as much as possible while not sacrificing quality.

6.11. Fonts

This library supports the font [https://developer.mozilla.org/en/docs/Web/CSS/font], font-size [https://developer.mozilla.org/en/docs/Web/CSS/font-size], font-family [https://developer.mozilla.org/en/docs/Web/CSS/font-family], font-style [https://developer.mozilla.org/en/docs/Web/CSS/font-style], font-weight [https://developer.mozilla.org/en/docs/Web/CSS/font-weight], and text-decoration [https://developer.mozilla.org/en/docs/Web/CSS/text-decoration] properties, as well as the @font-face [https://developer.mozilla.org/en/docs/Web/CSS/@font-face] CSS "at" rule for including TrueType and OpenType fonts.

6.11.1. `font-family`

By default, CN1's native fonts [https://www.codenameone.com/blog/good-looking-by-default-native-fonts-simulator-detection-more.html] are used. The appropriate native font is selected for the provided font-weight and font-style properties. You can also explicitly specify the native font you wish to use in the `font-family` property. For example:

```
SideCommand {
    font-family:  "native:MainThin";
}
```

If you omit the `font-family` directive altogether, it will use `native:MainRegular`. The following native fonts are available:

1. `native:MainThin`

2. `native:MainLight`
3. `native:MainRegular`
4. `native:MainBold`
5. `native:MainBlack`
6. `native:ItalicThin`
7. `native:ItalicLight`
8. `native:ItalicRegular`
9. `native:ItalicBold`
10. `native:ItalicBlack`

Using bundled fonts

If you want to use a font other than the built-in fonts, you'll need to define the font using the `@font-face` rule. For example:

```
@font-face {
    font-family: "GuideDemoFont";
    src: url(res/GuideDemoFont-Regular.ttf);
}
```

Both TrueType (`.ttf`) and OpenType (`.otf`) files work, upper-case or lower-case. Every platform loads both formats through its own font API, so an OpenType font needs no conversion before you bundle it. Web-only formats such as `.woff` aren't supported, and the CSS compiler fails the build on one rather than letting it reach the device.



The compiler reads each font at build time, so a file it can't parse, or one with no PostScript name, fails the build. That second case is worth knowing about: iOS resolves fonts by their PostScript name, so a font without one renders everywhere except on an iOS device.

Then you'll be able to reference the font using the specified `font-family` in any CSS element. For example:

```
MyLabel {
    font-family: "GuideDemoFont";
}
```

Where to put the font file

A relative `src` URL is resolved against the directory that holds the CSS file. You can keep font files directly beside `theme.css`, or in any subdirectory of it, whichever you prefer:

```
@font-face {
```

```
    font-family: "GuideRootFont";
    src: url(GuideRootFont.ttf);
}

@font-face {
    font-family: "GuideDemoFont Bold";
    src: url(res/GuideDemoFont-Bold.ttf);
}
```

The font does have to live somewhere under that directory, because only that directory is copied into the build. A font reached through `../`, or one that doesn't exist, fails the build rather than producing an app whose text falls back to the system font. Two `@font-face` rules naming different files that share a file name also fail, since fonts are deployed by file name alone and one would overwrite the other. Pointing two families at the same file is fine.

Family names, weights and styles

A `font-family` name that contains spaces must be quoted, both in the `@font-face` rule and wherever you reference it. An unquoted name is parsed as a list of separate identifiers, so `font-family: GuideDemoFont Bold` registers the family as `GuideDemoFont` and collides with your regular weight.

`font-weight` and `font-style` select between the built-in `native:` fonts, but they have no effect once `font-family` resolves to a `@font-face` rule. Declare one `@font-face` per weight and style you need, each with its own family name, as in the example above, then reference the right family from each UIID.

To change the base font of an entire theme, set `font-family` on the special `Default` selector, then override the UIIDs that need a bold or italic face:

```
Default {
    font-family: "GuideRootFont";
}

Title {
    font-family: "GuideDemoFont Bold";
}
```

Remote and GitHub-hosted fonts

The `@font-face` directive's `src` property will accept both local and remote URLs. The guide fixture below uses a local font so the demo build remains offline and repeatable; application CSS can replace the URL with an HTTPS font URL:

```
@font-face {
    font-family: "GuideDownloadedFont";
    src: url(res/GuideDemoFont-Regular.ttf);
}
```

In this case, it will download the `myfont.ttf` file into the build directory alongside the merged CSS file, and reuse that copy on later builds so that it doesn't have to make a network request every time. A `mvn clean` discards the cache and the next build downloads the font again.

Fonts are automatically copied next to the compiled `theme.res` when the CSS file is compiled, so that they're distributed with the app and available at runtime. The copy uses the font's file name only, which means two `@font-face` rules that point at identically named files in different directories will collide. The copy is also skipped when a file of that name is already there, so run `mvn clean` after you replace a font file with a different one of the same name.

GitHub URLs

Fonts hosted on GitHub are accessible using a special `github:` protocol to make it easier to reference such fonts. The compiled guide fixture uses a bundled icon font for offline validation; application CSS can use the same rule shape with a `github:` URL.

```
@font-face {
    font-family: "GuideIconFont";
    src: url(res/GuideIconFont.ttf);
}
```



If a third-party font repository moves or changes licensing, vendor the font into your project and reference it locally so builds stay reproducible.

6.11.2. `font-size`

It's best practice to size your fonts using millimetres (`rem`) (or another "real-world" measurement unit such as inches (`in`), centimetres (`cm`), millimetres (`mm`). This will allow the font to be sized appropriate for all display densities. If you specify size in pixels (`px`), it will treat it the same as if you sized it in points (`pt`), where 1pt == 1/72 inches (one seventy-second of an inch).

If you size your font in percentage units (for example, `150%`) it will set the font size relative to the medium font size of the platform. This is different than the standard behaviour of a web browser, which would size it relative to the parent element's font size.



`font-size: 150%` is the same as `font-size: 1.5rem`.

You can use system fonts, true type fonts, and native fonts in your CSS stylesheet. True Type fonts need to be defined in a `@font-face` directive before they can be referenced. True-type fonts and native fonts have the advantage that you can specify their sizes in generic terms (for example, `small`, `medium`, `large`) and in more specific units such as millimeters (`mm`) or pixels (`px`).

Normalizing Default Font Size

When trying to make a design look "good" across multiple platforms it can be difficult to deal with the differing default font sizes on different platforms. You may spend hours tweaking your UI to look perfect on iPhone X, only to find out that the fonts are too small when viewed

on an android device. You've now added theme constants to explicitly set the default font size in "screen-independent-pixels."

Note In this case, 1 screen-independent-pixel is defined as 1/160th of an inch on a device, and 1/96th of an inch on desktop. These values correspond to Android's definition on device, and Windows' definition on the desktop.

If you add the following to your stylesheet, it will set the default font size to 18 screen-independent pixels (or 18/160th of an inch), which corresponds to the Android native default "medium" font size:

```
#Constants {
 defaultFontSizeInt: 18;
}
```



A value of 18 here gives best results across devices.

On the desktop, you may find that 18 is too big. You can define a default font size for tablet and desktop using `defaultDesktopFontSizeInt` and `defaultTabletFontSizeInt` respectively. A `defaultDesktopFontSizeInt` value gives results that match the macOS default font size:

```
#Constants {
 defaultFontSizeInt: 18;
 defaultDesktopFontSizeInt: 14;
}
```

How density is determined per platform

The conversion from "screen-independent pixels" or `mm` to physical pixels uses the device's PPI. The two mobile ports compute it differently, which is worth keeping in mind when chasing per-platform sizing differences:

- **Android:** uses `density x 160` from `DisplayMetrics`, where `density` is the bucketed scale factor reported by Android (typically `1.0`, `1.5`, `2.0`, `3.0`, `3.5`, `4.0`). Because the buckets are coarse, a phone whose physical density falls between two buckets will report the lower one, so mm-based measurements may render slightly smaller than their nominal physical size.
- **iOS:** uses an explicit per-device PPI lookup keyed on the screen's pixel dimensions (for example, `460ppi` for iPhone 12 / 13 / 14 / 15 / 16, `458ppi` for the Plus and Pro Max models, `326ppi` for older `2×` devices). This is exact for catalogued devices. Any iPhone whose resolution isn't in the table falls through to a default fallback (now `460ppi`, since Apple has held that density steady for every non-Plus iPhone since the iPhone 12).

Net result: the same `defaultFontSizeInt: 18` may render at slightly different absolute physical sizes on an iPhone vs. An Android device of similar physical size—typically the iOS rendering

is the larger of the two when both are at the platform's default text size. If you need cross-platform pixel-perfect sizing, use platform overrides (`@iOS-platform-font-size`, etc., via media queries) rather than relying on the conversion math alone.



When a new iPhone model ships with a resolution that isn't yet in the iOS PPI table, mm-based measurements will use the fallback `460ppi`, which is correct to within 1-2% for every modern iPhone. If you discover a model that should use a different value (for example a future Plus/Max device that returns to `458ppi`), file an issue so the table can be updated.

6.11.3. `text-decoration`

See the text-decoration section [Supported-Properties#text-decoration] in the "Supported Properties" page.

6.11.4. Some sample CSS directives

```
@font-face {
    font-family: "GuideDemoFont";
    src: url(res/GuideDemoFont-Regular.ttf);
}

@font-face {
    font-family: "GuideDemoFont-Bold";
    src: url(res/GuideDemoFont-Bold.ttf);
}

PlainText0p5mm { font-size: 0.5mm; }
PlainText1mm { font-size: 1mm; }
PlainText2mm { font-size: 2mm; }
PlainText5mm { font-size: 5mm; }
PlainText10mm { font-size: 10mm; }
PlainText50mm { font-size: 50mm; }
PlainTextSmall { font-size: small; }
PlainTextMedium { font-size: medium; }
PlainTextLarge { font-size: large; }
PlainText3pt { font-size: 3pt; }
PlainText6pt { font-size: 6pt; }
PlainText12pt { font-size: 12pt; }
PlainText20pt { font-size: 20pt; }
PlainText36pt { font-size: 36pt; }

BoldText { font-weight: bold; }
BoldText1mm { font-weight: bold; font-size: 1mm; }
BoldText2mm { font-weight: bold; font-size: 2mm; }
BoldText3mm { font-weight: bold; font-size: 3mm; }
BoldText5mm { font-weight: bold; font-size: 5mm; }
```

```
ItalicText { font-style: italic; }
ItalicText3mm { font-style: italic; font-size: 3mm; }
ItalicBoldText { font-style: italic; font-weight: bold; }

PlainTextUnderline { text-decoration: underline; }
BoldTextUnderline { text-decoration: underline; font-weight: bold; }
ItalicTextUnderline { text-decoration: underline; font-style: italic; }

PlainText3d {
    text-decoration: cn1-3d;
    color: white;
    background-color: #3399ff;
}

BoldText3d {
    text-decoration: cn1-3d;
    font-weight: bold;
    color: white;
    background-color: #3399ff;
}

ItalicText3d {
    text-decoration: cn1-3d;
    font-style: italic;
    color: white;
    background-color: #3399ff;
}

PlainText3dLowered {
    text-decoration: cn1-3d-lowered;
    color: black;
    background-color: #3399ff;
}

BoldText3dLowered {
    text-decoration: cn1-3d-lowered;
    font-weight: bold;
    color: black;
    background-color: #3399ff;
}

ItalicText3dLowered {
    text-decoration: cn1-3d-lowered;
    font-style: italic;
    color: black;
    background-color: #3399ff;
}

PlainText3dShadow {
    text-decoration: cn1-3d-shadow-north;
    color: white;
```

```
    background-color: #3399ff;
}

BoldText3dShadow {
    text-decoration: cn1-3d-shadow-north;
    font-weight: bold;
    color: white;
    background-color: #3399ff;
}

ItalicText3dShadow {
    text-decoration: cn1-3d-shadow-north;
    font-style: italic;
    color: white;
    background-color: #3399ff;
}

MainThin {
    font-size: 200%;
    background: radial-gradient(circle at top left, yellow, blue 100%);
}

MainRegular0001 {
    font-family: "native:MainRegular";
    color: blue;
    border: 1px cn1-pill-border blue;
    padding: 2mm;
}

MainRegular0001.pressed {
    font-family: "native:MainRegular";
    background: cn1-pill-border blue;
    color: white;
    border: 1px solid white;
    padding: 2mm;
}

Heading {
    font-size: 4mm;
    font-family: "GuideDemoFont-Bold";
    color: black;
    padding: 2mm;
    text-align: center;
}
```

6.12. Media queries

You can use media queries to target styles to specific platforms, devices, and device densities. The following media queries are supported:

1. `platform-xxx` - Target a specific platform. For example, `platform-and`, `platform-ios`, `platform-mac`, `platform-win`.
2. `density-xxx` - Target a specific device density. For example, `density–low`, `density-low`, `density-medium`, `density-high`, `density–high`, `density-hd`, `density-2hd`, and `density-560`.
3. `device-xxx` - Target a specific device type. For example, `device-desktop`, `device-tablet`, `device-phone`, `device-tv` (Apple TV / Android TV), and `device-watch` (Apple Watch / Wear OS). The `device-tv` and `device-watch` variants are selected at runtime when `Display.isTV()` / `Display.isWatch()` returns `true`, letting you adapt styling for the 10-foot TV UI or the small wearable screen.

Also to the Codename One specific media tokens above, the CSS compilers also recognize standard dark-mode media queries using `prefers-color-scheme: dark`. Rules inside these blocks are compiled into `$Dark` UIIDs automatically (for example, `Button` becomes `$DarkButton`, `Button.selected` becomes `$DarkButton.selected`).

Listing 7. Example: Dark-mode overrides with standard CSS media query syntax

```
Button {
    color: #222222;
}

@media (prefers-color-scheme: dark) {
    Button {
        color: #f0f0f0;
        background-color: #000000;
    }

    Button.selected {
        color: #ff0000;
    }
}
```

Listing 8. Example: Different font colors on Android and iOS. On Android, labels will appear green. On iOS, they will appear red. On all other platforms, they will appear black:

```
Label {
    color: black;
}

@media platform-and {
    Label {
        color: green;
    }
}

@media platform-ios {
    Label {
        color: red;
    }
```

```
}
```

Listing 9. Example: Different font colors based on device density. On lower densities, labels will be green. On higher densities, labels will be red.

```
Label {
    color: black;
}

@media density-very-low, density-low, density-medium, density-high {
    Label {
        color: green;
    }
}

@media density-very-high, density-h2, density-2hd, density-560 {
    Label {
        color: red;
    }
}
```

Listing 10. Example: Different label colors based on device type.

```
Label {
    color: black;
}

@media device-desktop {
    Label {
        color: green;
    }
}

@media device-tablet, device-phone {
    Label {
        color: red;
    }
}
```



When deploying your app using the JavaScript port, it will use a platform name derived from the "UserAgent" string in the browser, rather than the result of `Display.getPlatformName()`, which is used for other ports. When running on Android, then, the platform will be "and." When running on iOS, the platform will be "ios." Etc.

6.12.1. Compound media queries

You can combine multiple media queries together, separated by a comma. Queries of the same type

are "OR"ed together. Queries of different types are "AND"ed together. For example if you have a media query that specifies two different device densities (for example, `density-low` and `density-high`) the query will match **both** devices with low density and high density. But, if the query specifies a device density and a platform (for example, `density-low` and `platform-and`), then it will match a device if it matches the platform **and** the density.

Listing 11. Example: Targeting styles to Android devices with high density

```
@media platform-and, density-high {
    Label {
        color: green;
    }
}
```

Listing 12. Example: Targeting styles to iOS devices with high or low density

```
@media platform-ios, density-high, density-low {
    Label {
        color: red;
    }
}
```

Listing 13. Example: Targeting Mac Desktop:

```
@media device-desktop, platform-mac {
    Label {
        color: blue;
    }
}
```

6.12.2. Order or precedence

The order of precedence when applying styles differs slightly from the way styles would be applied in standard CSS. The order of precedence is as follows:

1. Styles defined inside `@media` blocks will always take precedence over styles defined outside of `@media` blocks.
2. `@media` blocks with more query matches will take precedence over blocks with fewer query matches. For example, A media block matching density, platform, and device will take precedence over a block that only matches the density and platform.
3. If the same style is defined in two media blocks which contain the same number of query matches, then the order precedence is `platform`, `device`, `density` in decreasing order. That is, the block that matches on platform will take precedence over the block that matches on density.
4. If the same style is defined in two media blocks with identical query matches, then the order of precedence is undefined.

6.12.3. Font scaling constants

Sometimes you may find that fonts are coming out too large or too small across the board on certain types of devices. You can use standard media queries to customize font sizes, but you can also use `font-scaling` constants to **scale** font sizes for the entire stylesheet based on platform, device, and/or density. Sometimes you may find this approach easier.

For example, consider the following simple stylesheet that defines a font size of 2mm on labels:

```
Label {
 font-size: 3mm;
}
```

During testing, perhaps you find that, on desktop, the fonts are a little bit too small. In this case, you can apply a font-scale constant that only applies to the desktop:

```
#Constants {
 device-desktop-font-scale: "1.5";
}

Label {
 font-size: 3mm;
}
```

Now, on most devices the Label style will have `3mm` fonts. However, on desktop, it will have `4.5mm` fonts.

The above would be equivalent to:

```
Label {
 font-size: 3mm;
}

@media device-desktop {
 Label {
 font-size: 4.5mm;
 }
}
```

Listing 14. Example: Font-scaling based on device, platform, and density

```
#Constants {
 device-phone-font-scale: "1.5";
 device-tablet-font-scale: "1.2";
 device-desktop-font-scale: "1.4";
 platform-ios-font-scale: "0.9";
 density-low-font-scale: "1.2";
```

```
 platform-ios-density-low-font-scale: "1.3";
}
```



All matching `font-scale` constants will be applied to the styles. If you define 3 font-scale constants that all match the current runtime environment, they will all be applied. For example, If there are 3 matching font-scale constants with `2.0`, `3.0`, and `4.0`, then fonts will be scaled by 2*3*4=24!

Chapter 7. The components of Codename One

This chapter covers the components of Codename One. It doesn't cover every component, but it goes deeper than the Javadocs [https://www.codenameone.com/javadoc/].

7.1. Container

The Codename One container is a base class for many high-level components. A container can hold other components.

Every component has a parent container. That parent can be `null` when the component isn't inside another container or when it's a top-level container. A container can have many children.

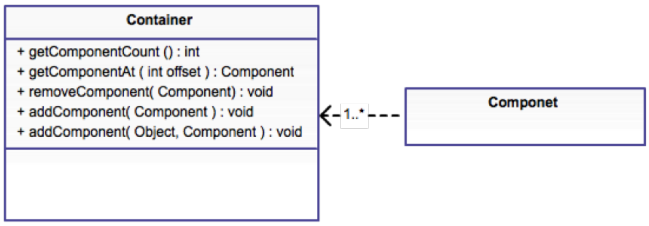


Figure 97. Component-Container relationship expressed as UML

Components are arranged in containers with layout managers. A layout manager decides how to arrange components inside the container.

You can read more about layout managers in the basics section [https://www.codenameone.com/manual/basics.html#component-container-hierarchy].

7.1.1. Composite components

Codename One components share a generic inheritance hierarchy. For example, Button [https://www.codenameone.com/javadoc/com/codename1/ui/Button.html] derives from Label [https://www.codenameone.com/javadoc/com/codename1/ui/Label.html] and thus receives all its abilities.

Some components are composites and derive from the Container [https://www.codenameone.com/javadoc/com/codename1/ui/Container.html] class. For example, the MultiButton [https://www.codenameone.com/javadoc/com/codename1/components/MultiButton.html] is a composite button that derives from `Container` but behaves like a `Button`. Most of the time this feels seamless, but a few details matter.

- Don't use the `Container`-derived methods on such a composite component (for example, `add` and `remove`).
- You can't cast it to the type it relates to. For example, you can't cast `MultiButton` to `Button`.
- Events can be more nuanced. For example, if you rely on ActionEvent.getSource() [https://www.codenameone.com/javadoc/com/codename1/ui/events/ActionEvent.html#getSource--] or ActionEvent.getComponent() [https://www.codenameone.com/javadoc/com/codename1/ui/events/ActionEvent.html#getComponent--], they may not behave as expected. For a `MultiButton`, they return

the underlying `Button`. To work around that, use ActionEvent.getActualComponent() [https://www.codenameone.com/javadoc/com/codename1/ui/events/ActionEvent.html#getActualComponent--].

Lead Component

Codename One includes a feature for creating composite components called "lead components." It lets components like `MultiButton` act like a single component while still containing multiple components.

Lead components work by assigning one component as the "leader." That leader determines the style state for every component in the hierarchy. If a `Container` is led by a `Button`, the button determines whether the selected or pressed state applies to the entire hierarchy.

This means a single `Component` can contain multiple nested `UIID`s. For example, `MultiButton` has `UIID`s such as `MultiLine1`, which you can customize with APIs such as setUIIDLine1 [https://www.codenameone.com/javadoc/com/codename1/components/MultiButton.html#setUIIDLine1-java.lang.String-].

The lead component also handles events from a single source. Clicking another component in the hierarchy sends the event to the leading `Button`, which can make action events route to a different target than the apparent click target. That's why `getActualComponent()` exists.

You can learn more about lead components in here [https://www.codenameone.com/manual/misc-features.html#lead-component-section].

7.2. Form

Form [https://www.codenameone.com/javadoc/com/codename1/ui/Form.html] is the top-level container in Codename One. `Form` derives from `Container` and is the element the app shows. Only one form can be visible at any given time. You can get the visible `Form` with the code:

```
Form currentForm = Display.getInstance().getCurrent();
```

A form is a container with a title, a content area, and an optional menu or menu-bar area. When you call methods such as `add` or `remove` on a form, you're invoking something that maps to this:

`Form` is effectively a Container [https://www.codenameone.com/javadoc/com/codename1/ui/Container.html] with a border layout. The north section holds the title area, the south section holds the optional menu bar, and the center stretches as the content pane. Place all components in the content pane.



Figure 98. Form layout graphic

You can see that every `Form` has space allocated for the title area. If you don't set the title, it won't show up, but it will still be there. The same isn't always true for the menu bar, which can vary. Effectively, the section that matters is the content pane, so the form tries to do the right thing by pretending to be the content pane. For example, this isn't always seamless, and sometimes code needs to invoke `getContentPane()` to work directly with the container.



A good example for such a case is layout animations. Animating the form might not produce the right results. When in doubt, it's easier to use `getContentPane()` instead of working with the `Form` directly.

As you can see from the graphic, `Form` has two layers that live on top of the content pane/title. The first is the layered pane, which allows you to place "always on top" components. The layered pane is added implicitly when you invoke `getLayeredPane()`.



You still need to place components using layout managers to get them to appear in the right place when using the layered pane.

7.2.1. Safe areas

Modern phones often include display cut-outs, rounded corners, or persistent gesture/navigation indicators that overlap the physical screen edges. These intrusions are no longer limited to iOS—many recent Android devices behave the same way—so Codename One provides a "safe area" API to ensure your UI remains visible regardless of the device. The safe area is the portion of the display that's guaranteed to be unobstructed.



Figure 99. Tabs with safe-area padding enabled



Figure 100. Tabs without safe-area padding

Automatic Safe-Area handling

`Form` calculates the safe region automatically, and core UI components that anchor themselves to the screen edges apply that information without more work:

- `Toolbar` and the status bar placeholder respect the top safe margins so that titles and action buttons remain legible around notches.

- `Tabs`, `Sheet`, and `FloatingActionButton` instances mark their internal containers as safe areas so navigation controls are padded above gesture/navigation bars.
- UI fragment templates can opt-in to safe areas declaratively using the `safeArea` flag, ensuring generated components stay within the padded region.

This means that many applications will work on devices such as the iPhone X/14 family or Android devices with edge-to-edge displays. You need to intervene when you use custom containers that you position flush against a screen edge or when you perform your own painting.

Marking your own containers as safe

If you create a container that should always stay clear of the unsafe portions of the screen (for example, a bottom navigation bar, an on-screen joystick, or a floating tool palette), enable safe-area padding explicitly:

```
Form form = new Form(new BorderLayout());

Container bottomBar = new Container(BoxLayout.x());
bottomBar.setSafeArea(true);
bottomBar.addAll(new Button("Home"), new Button("Search"), new Button("Profile"));

form.add(BorderLayout.SOUTH, bottomBar);
form.show();
```

Safe-area padding is applied when the container **doesn't** have a scrollable parent. For scrollable content, you can assume the user can scroll the component into view instead.

Most layouts never need to know where the safe area begins, but if you draw manually (for example, inside `paint()` or on the glass pane) you can query it directly:

```
Form form = Display.getInstance().getCurrent();
Rectangle safe = form.getSafeArea();

Graphics g = null; // e.g. inside paint()
g.setClip(safe.getX(), safe.getY(), safe.getWidth(), safe.getHeight());
// Custom drawing code that should avoid the notch/gesture areas
```

The rectangle returned by `Form#getSafeArea()` is updated automatically whenever the OS reports a change (rotation, multitasking gestures, showing or hiding the system navigation area, etc.). In unusual situations where you adjust the safe-area root yourself (for example, when animating a container in from off-screen) you can force a recalculation by calling `Form#setSafeAreaChanged()`.

Safe-Area roots and advanced layouts

Safe-area padding is calculated relative to a "safe-area root." Forms are roots by default, but you can mark any container as a root using `Container#setSafeAreaRoot(true)` when you need precise control, for example, when preparing a side menu that starts off-screen and slides in:

```
Container drawer = new Container(BoxLayout.y());
drawer.setSafeAreaRoot(true); // Ensure safe margins apply before the drawer is visible
drawer.setSafeArea(true);
```

Marking the drawer as both a root and a safe area prevents a jump the moment it becomes visible, because the safe padding is already applied while it's off-screen.

Remember that safe areas apply across platforms. Always verify your screens on actual devices (or in the Codename One simulator with a device skin that exposes cut-outs) to make sure critical UI elements remain inside the padded region.

The second layer is the glass pane which allows you to draw arbitrary things on top of everything. The order in the image is indeed right:

1. `ContentPane` is lowest
2. `LayeredPane` is second
3. `GlassPane` is painted last



It's important to notice that a layered pane is on top of the `ContentPane` and doesn't stretch to the title. A `GlassPane` stretches all the way but with a "lightweight" title area, for example, the Toolbar API [https://www.codenameone.com/javadoc/com/codename1/ui/Toolbar.html].

The `GlassPane` allows developers to overlay UI on top of existing UI and paint as they see fit. This is useful for things that provide notification but don't want to intrude with application functionality.

7.3. Dialog

A Dialog [https://www.codenameone.com/javadoc/com/codename1/ui/Dialog.html] is a special kind of `Form` that can occupy a portion of the screen. It also has the more functionality of the modal `show` method.

When showing a dialog you have two basic options: modeless and modal:

- Modal dialogs (the default) block the current EDT thread until the dialog is dismissed. To understand how they do it, read about `invokeAndBlock`.
  Modal dialogs are a useful way to prompt the user since the code can assume the user responded in the next line of execution. This promotes a linear and intuitive way of writing code.
- Modeless dialogs return, so a call to show such a dialog can't assume anything in the next line of execution. This is useful for features such as progress indicators where you aren't waiting for user input.

For example: a modal dialog can be expressed as such:

```
if(Dialog.show("Click Yes Or No", "Select one", "Yes", "No")) {
    // user clicked yes
```

```
} else {
    // user clicked no
}
```

Notice that during the `show` call above the execution of the next line was "paused" until you got a response from the user and once the response was returned you could proceed directly.



All usage of `Dialog` must be within the Event Dispatch Thread (the default thread of Codename One). This is true for modal dialogs. The `Dialog` class knows how to "block the EDT" without blocking it.

To learn more about `invokeAndBlock` which is the workhorse behind the modal dialog functionality check out the EDT section [https://www.codenameone.com/manual/edt.html].

The `Dialog` class contains many static helper methods to show user notifications, but also allows a developer to create a `Dialog` instance, add information to its content pane and show the dialog.



Dialogs contain a `ContentPane` like `Form`.

When showing a dialog in this way, you can either ask Codename One to position the dialog in a specific location based on the BorderLayout [https://www.codenameone.com/javadoc/com/codename1/ui/layouts/BorderLayout.html] concept of locations, or position it by spacing it in pixels from the four edges of the screen.

For example: you could do something like this to show a simple modal `Dialog`:

```
Dialog d = new Dialog("Title");
d.setLayout(new BorderLayout());
d.add(BorderLayout.CENTER, new SpanLabel("Dialog Body", "DialogBody"));
d.showPacked(BorderLayout.SOUTH, true);
```

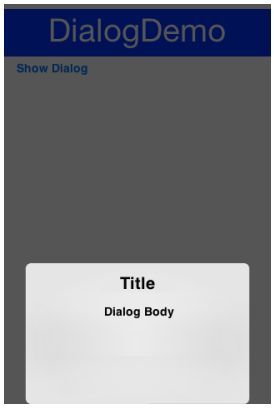


Figure 101. Custom modal Dialog in the south position



You can turn the code above to a modless `Dialog` by flipping the boolean `true` argument to `false`.

You can position a `Dialog` by determining the space from the edges for example: with this code you

can occupy the bottom portion of the screen:

```
Dialog d = new Dialog("Title");
d.setLayout(new BorderLayout());
d.add(BorderLayout.CENTER, new SpanLabel("Dialog Body", "DialogBody"));
d.show(hi.getHeight() / 2, 0, 0, 0);
```

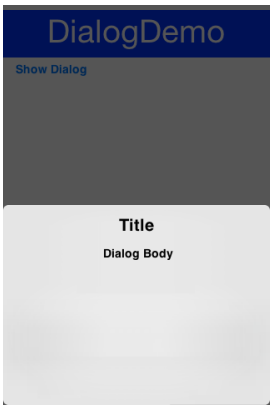


Figure 102. Custom Dialog positioned absolutely



`hi` is the name of the parent `Form` in the sample above.

7.3.1. Styling dialogs

It's important to style a `Dialog` using getDialogStyle() [https://www.codenameone.com/javadoc/com/codename1/ui/Dialog.html#getDialogStyle--] or setDialogUIID [https://www.codenameone.com/javadoc/com/codename1/ui/Dialog.html#setDialogUIID-java.lang.String-] methods rather than styling the dialog object directly.

The reason for this is that the `Dialog` is a `Form` that takes up the whole screen. The `Form` that's visible behind the `Dialog` is rendered as a screenshot. Customizing the actual `UIID` of the `Dialog` won't produce the desired results.

7.3.2. Tint and blurring

By default a `Dialog` uses a platform specific tint color when it's showing for example: notice the background in the image below is tinted:

```
Form hi = new Form("Tint Dialog", new BoxLayout(BoxLayout.Y_AXIS));
Button showDialog = new Button("Tint");
showDialog.addActionListener((e) -> Dialog.show("Tint", "Is On/* omitted */", "OK", null));
hi.add(showDialog);
hi.show();
```

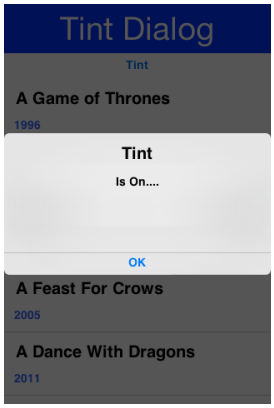


Figure 103. Dialog with tinted background

The tint color can be manipulated on the parent form, you can set it to any AARRGGBB value to set any color using the `setTintColor` method. Notice that this is invoked on the parent form and not on the `Dialog`!



This is an AARRGGBB value and not an RRGGBB value! This means that 0 will be transparent.

You can also manipulate this default value globally using the theme constant `tintColor`. The sample below tints the background in green:

```
Form hi = new Form("Tint Dialog", new BoxLayout(BoxLayout.Y_AXIS));
hi.setTintColor(0x7700ff00);
Button showDialog = new Button("Tint");
showDialog.addActionListener((e) -> Dialog.show("Tint", "Is On/* omitted */", "OK", null));
hi.add(showDialog);
hi.show();
```

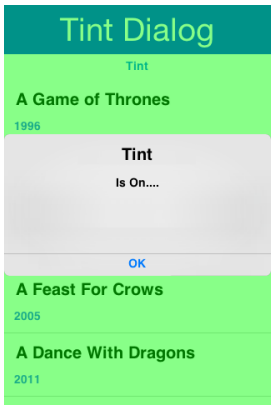


Figure 104. Dialog with green tinted background

You can apply Gaussian blur to the background of a dialog to highlight the foreground further and produce an attractive effect. You can use the `setDefaultBlurBackgroundRadius` to apply this globally, you can use the theme constant `dialogBlurRadiusInt` to do the same or you can do this on a per `Dialog` basis using `setBlurBackgroundRadius`.



Not all device types support blur you can test if your device supports it using `Display.getInstnace().isGaussianBlurSupported()`. If blur isn't supported the blur

setting will be ignored:

```
Form hi = new Form("Blur Dialog", new BoxLayout(BoxLayout.Y_AXIS));
Dialog.setDefaultBlurBackgroundRadius(8);
Button showDialog = new Button("Blur");
showDialog.addActionListener((e) -> Dialog.show("Blur", "Is On/* omitted */", "OK", null));
hi.add(showDialog);
hi.show();
```

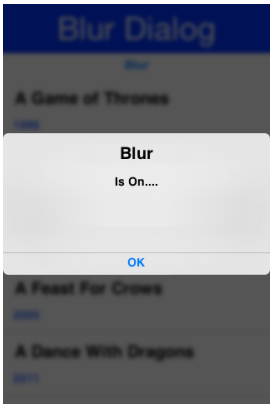


Figure 105. The blur effect coupled with the OS default tint

It might be a bit hard to notice the blur effect with the tinting so here is the same code with tinting disabled:

```
hi.setTintColor(0);
```

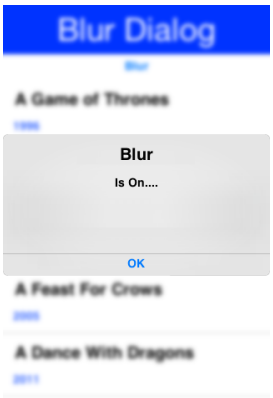


Figure 106. The blur effect is more pronounced when the tint is disabled

7.3.3. Popup dialog

A popup dialog is a common mobile paradigm showing a `Dialog` that points at a specific component. It's a standard `Dialog` that's shown in a unique way:

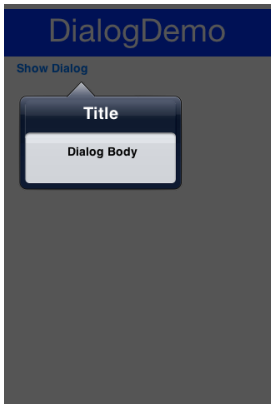


Figure 107. Popup Dialog

The popup dialog accepts a Component [https://www.codenameone.com/javadoc/com/codename1/ui/Component.html] or Rectangle [https://www.codenameone.com/javadoc/com/codename1/ui/geom/Rectangle.html] to point at and handles the rest.

Styling the arrow of the popup dialog

When Codename One was young you needed a popup arrow implementation but your low-level graphics API was pretty basic. As a workaround you created a version of the 9-piece image border that supported pointing arrows at a component.

Today Codename One supports pointing an arrow from the `RoundRectBorder` class. This is implicit for the `PopupDialog` UI. This allows for better customization of the border (color etc.) and it looks better on newer displays. It also works on all OSs. Right now the iOS theme has the old image border approach.



This will change with a future update where all OSes will align and iOS will use the lightweight popup too



You can make all OSes act the same way by overriding the `PopupDialog` UIID and defining its style to `RoundRectBorder`

The new `RoundRectBorder` support works by setting the track component property on border. When that's done the border implicitly points to the right location.

If you still need deeper customization of the arrow you can still use the old 9-piece border functionality illustrated below.

Legacy 9-Piece Border arrow

One of the harder aspects of a popup dialog is the construction of the theme elements required for arrow styling. To get that sort of behavior you will need a custom image border and 4 arrows pointing in each direction that will be overlaid with the border.



The sizes of the arrow images should be similarly proportioned and fit within the image borders whitespace. The block image of the dialog should have empty pixels in the sides to reserve space for the arrow. For example: if the arrows are all 32×32 pixels then the `PopupDialog` image should have 32 pixels of transparent pixels

around it.

You will need to define the following theme constants for the arrow to work:

Then style the `PopupDialog` UIID with the image for the `Dialog` itself.

7.4. InteractionDialog

Dialogs in Codename One can be modal or modeless, the former blocks the calling thread and the latter doesn't. For example, there is another definition to those terms: A modal dialog blocks access to the rest of the UI while a modeless dialog "floats" on top of the UI.

In that sense, all dialogs in Codename One are modal; they block the parent form since they're effectively forms that show the "parent" in their background. InteractionDialog [https://www.codenameone.com/javadoc/com/codename1/components/InteractionDialog.html] has an API that's like the Dialog [https://www.codenameone.com/javadoc/com/codename1/ui/Dialog.html] API but, unlike dialog, it never blocks anything. Neither the calling thread nor the UI.



`InteractionDialog` isn't a `Dialog` since it doesn't share the same inheritance hierarchy. For example, it acts and "feels" like a `Dialog` although it's a `Container` in the `LayeredPane`.

`InteractionDialog` is a container that's positioned within the layered pane. Notice that because of that design, you can have one such dialog at the moment and, if you add something else to the layered pane, you might run into trouble.

Using the interaction dialog is pretty trivial and like dialog:

```
InteractionDialog dlg = new InteractionDialog("Hello");
dlg.setLayout(new BorderLayout());
dlg.add(BorderLayout.CENTER, new Label("Hello Dialog"));
Button close = new Button("Close");
close.addActionListener((ee) -> dlg.dispose());
dlg.addComponent(BorderLayout.SOUTH, close);
Dimension pre = dlg.getContentPane().getPreferredSize();
int displayWidth = Display.getInstance().getDisplayWidth();
int dialogWidth = Math.max(pre.getWidth() + pre.getWidth() / 6, displayWidth * 2 / 3);
dlg.show(0, 0, displayWidth - dialogWidth, 0);
```

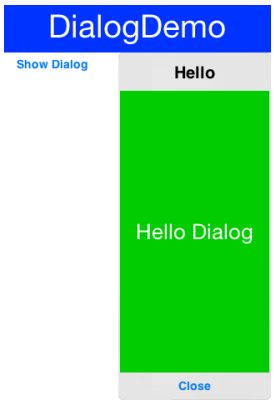


Figure 108. Interaction Dialog

This will show the dialog on the right-hand side of the screen, which is pretty useful for a floating in place dialog.



The `InteractionDialog` can be shown at absolute or popup locations. This is inherent to its use case which is "non-blocking." When using this component you need to be aware of its location.

To make popup behavior feel natural on touch devices you can call `setDisposeWhenPointerOutOfBounds(true)` so the dialog automatically dismisses as soon as the user taps outside the title or content area. Internally the dialog listens for pointer pressed/released events and will call `dispose()` for you when the interaction happens beyond its bounds, so you no longer need to wire that logic manually.

By default the dialog is placed on the form's layered pane, but you can switch between the global layered pane and form-specific layered pane using `setFormMode(boolean)`. Setting form mode to `true` keeps the dialog coupled with the showing form even when the global layered pane is used elsewhere in your app.

Finally, recent updates added animation toggles so you can fine-tune presentation. `setAnimateShow(boolean)` turns the show/dispose animations on or off, while `setRepositionAnimation(boolean)` enables a "grow/shrink" reposition effect during those animations when you want a more dynamic transition.

7.5. Label

Label [https://www.codenameone.com/javadoc/com/codename1/ui/Label.html] represents a text, icon, or both. `Label` is also the base class of `Button` which in turn is the base class for `RadioButton` & `CheckBox`. Thus the functionality of the `Label` class extends to all these components.

`Label` text can be positioned in one of 4 locations as such:

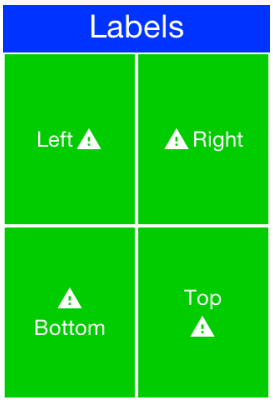


Figure 109. Label positions

`Label` allows a single line of text, line breaking is an expensive operation on mobile devices [1] and so the `Label` class doesn't support it.



SpanLabel supports many lines with a single label, notice that it does carry a performance penalty for this functionality.

Labels support tickering and the ability to end with "..." if there isn't enough space to render the label. Developers can determine the placement of the label to its icon in a few powerful ways.

7.5.1. Label gap

The gap between the label text & the icon defaults to 2 pixels due to legacy settings. The `setGap` method of `Label` accepts a gap size in pixels.

Two pixels is low for most cases & it's hard to customize for each `Label`.

You can use the theme constant `labelGap` which is a floating point value you can specify in millimeters that will allow you to determine the default gap for a label. You can also customize this manually using the method `Label.setDefaultGap(int)` which determines the default gap in pixels.

7.5.2. Autosizing labels

One of the common requests you received over the years is a way to let text "fit" into the allocated space so the font will match almost the width available. In some designs this is important but it's also tricky. Measuring the width of a String is an expensive operation on some OSes., there is no other way other than trial & error to find the "best size."

Still although something is "slow" you might still want to use it for some cases, this isn't something you should use in a renderer, infinite scroll etc. and recommend minimizing the usage of this feature as much as possible.

This feature is applicable to `Label` and its subclasses (for example: `Button`), with components such as `TextArea` (for example: `SpanButton`) the choice between shrinking and line break would require some complex logic.

To activate this feature use `setAutoSizeMode(true)` for example:

```
Form hi = new Form("AutoSize", BoxLayout.y());

Label a = new Label("Short Text");
a.setAutoSizeMode(true);
Label b = new Label("Much Longer Text than the previous linenull");
b.setAutoSizeMode(true);
Label c = new Label("MUCH MUCH MUCH Much Longer Text than the previous line by a pretty big marginnull");
c.setAutoSizeMode(true);

Label a1 = new Button("Short Text");
a1.setAutoSizeMode(true);
Label b1 = new Button("Much Longer Text than the previous linenull");
b1.setAutoSizeMode(true);
Label c1 = new Button("MUCH MUCH MUCH Much Longer Text than the previous line by a pretty big marginnull");
c1.setAutoSizeMode(true);
hi.addAll(a, b, c, a1, b1, c1);

hi.show();
```



Figure 110. Automatically sizes the fonts of the buttons/labels based on text and available space

7.6. TextField and TextArea

The TextField [https://www.codenameone.com/javadoc/com/codename1/ui/TextField.html] class derives from the TextArea [https://www.codenameone.com/javadoc/com/codename1/ui/TextArea.html] class, and both are used for text input in Codename One.

`TextArea` defaults to multi-line input and `TextField` defaults to single line input but both can be used in both cases. The main differences between `TextField` and `TextArea` are:

- Blinking cursor is rendered on `TextField`
- DataChangeListener [https://www.codenameone.com/javadoc/com/codename1/ui/events/DataChangedListener.html] is available in `TextField`. This is crucial for character by character input event tracking
- Done listener [https://www.codenameone.com/javadoc/com/codename1/ui/TextField.html#setDoneListener-com.codename1.ui.events.ActionListener-] is available in the `TextField`
- Different `UIID`



The semantic difference between `TextField` & `TextArea` dates back to the ancestor of Codename One: LWUIT. Feature phones don't have "proper" in-place editing

capabilities & thus `TextField` was introduced to allow such input.

Because it lacks the blinking cursor capability `TextArea` is often used as a multi-line label and is used internally in `SpanLabel`, `SpanButton` etc.



A common use case is to have an important text component in edit mode as you enter a `Form`. Codename One forms support this exact use case through the Form.setEditOnShow(TextArea) [https://www.codenameone.com/javadoc/com/codename1/ui/Form.html#setEditOnShow-com.codename1.ui.TextArea-] method.

`TextField` & `TextArea` support constraints for various types of input such as `NUMERIC`, `EMAIL`, `URL`, etc. Those usually affect the virtual keyboard used, but might not limit input in some platforms. For example: on iOS even with `NUMERIC` constraint you would still be able to input characters.



If you need to prevent specific types of input check out the validation section.

The following sample shows off simple text field usage:

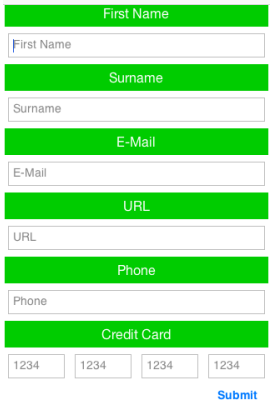


Figure 111. Simple text component sample



The Toolbar section contains an elaborate `TextField` search sample with `DataChangeListener` and rather unique styling.

7.6.1. Masking

A common use case when working with text components is the ability to "mask" input for example: in the credit card number above you would want 4 digits for each text field and don't want the user to tap Next 3 times.

Masking allows you to accept partial input in one field and implicitly move to the next, this can be used to all types of complex input thanks to the text component API. E.g with the code above you can mask the credit card input so the cursor jumps to the next field implicitly using this code:

Then implement the method `automoveToNext` as:

Notice you can invoke `stopEditing(Runnable)` where you receive a callback as editing is stopped.

7.6.2. The virtual keyboard

A common misconception for developers is assuming the virtual keyboard represents "keys." For example: developers often override the "keyEvent" callbacks which are invoked for physical keyboard typing and expect those to occur with a virtual keyboard.

This isn't the case since a virtual keyboard is a different beast. With a virtual keyboard characters typed might produce a different output due to autocorrect. Some keyboards don't even have "keys" in the traditional sense or don't type them in the traditional sense (for example: swiping).



The constraint property for the `TextField`/`TextArea` is crucial for a virtual keyboard.



When working with a virtual keyboard it's important that the parent `Container` for the `TextField`/`TextArea` is scrollable. Otherwise the component won't be reachable or the UI might be distorted when the keyboard appears.

Action Button client property

By default, the virtual keyboard on Android has a "Done" button, you can customize it to be a search icon, a send icon, or a go icon using a hint such as this:

This will adapt the icon for the action on the keys.

Next and done on iOS

You try to hide a lot of the platform differences in Codename One, input is ** different between OSes. A common reliance is the ability to send the "Done" event when the user presses the `Done` button. This button doesn't always exist for example: if there is an `Enter` button (due to multiline input) or if there is a `Next` button in that place.

To unify the behavior you slightly customized the iOS keyboard as such:

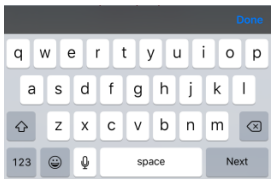


Figure 112. Next virtual keyboard with toolbar



Figure 113. Done virtual keyboard without toolbar



This works with 3rd party keyboards too...

For example, this behavior might not be desired so to block that you can do:

This will hide the toolbar for that given field.



You can customize the color of the Done button in the toolbar by setting the `ios.doneButtonColor` display property. For example: To change the color to red, you could do `Display.getInstance().setProperty("ios.doneButtonColor", String.valueOf(0xff0000))`.

7.6.3. Clearable text field

iOS has a convention where an X can be placed after the text field to clear it. Some Android apps have it but there is no native support for that as of this writing.

You can wrap a `TextField` with a clearable wrapper to get this effect on all platforms. For example: replace this:

With this:

You can also specify the size of the clear icon if you wish. This is technically a `Container` with the text field style and a button to clear the text at the edge.

7.7. TextComponent

When building input forms you sometimes want to adapt to the native OS behavior and create a UI that's a bit more distinct to the native OS. `TextField` and `TextArea` are low-level, you can create an Android style UI with such components but it might look out of place in iOS.

For example: this is how most of you would expect the UI to look on iOS and Android respectively:

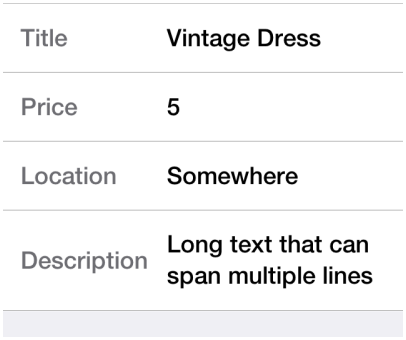


Figure 114. Text Input on iOS

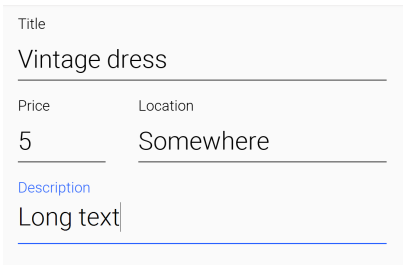


Figure 115. Text Input on Android

Doing this with text fields is possible but would require code that looks a bit different and jumps through hoops. `TextComponent` allows this exact UI without forcing developers to write OS specific code:

```
TextModeLayout tl = new TextModeLayout(3, 2);
Form f = new Form("Pixel Perfect", tl);
TextComponent title = new TextComponent().label("Title");
TextComponent price = new TextComponent().label("Price");
TextComponent location = new TextComponent().label("Location");
TextComponent description = new TextComponent().label("Description").multiline(true);

f.add(tl.createConstraint().horizontalSpan(2), title);
f.add(tl.createConstraint().widthPercentage(30), price);
f.add(tl.createConstraint().widthPercentage(70), location);
f.add(tl.createConstraint().horizontalSpan(2), description);
f.setEditOnShow(title.getField());
f.show();
```



This code uses the `TextModeLayout` which is discussed in the layouts section

The text component uses a builder approach to set various values for example:

```
TextComponent t = new TextComponent().
    text("This appears in the text field").
    hint("This is the hint").
    label("This is the label").
    multiline(true);
```

The code is pretty self-explanatory and more convenient than typical setters/getters. It automatically handles the floating hint style of animation when running on Android.

7.7.1. Error handling

The validator class supports text component and it should "work." But the cool thing is that it uses the material design convention for error handling!

If you add to the sample above a `Validator`:

You would see something that looks like this on Android:

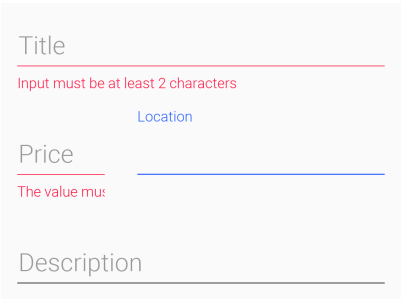


Figure 116. Error handling when the text is blank

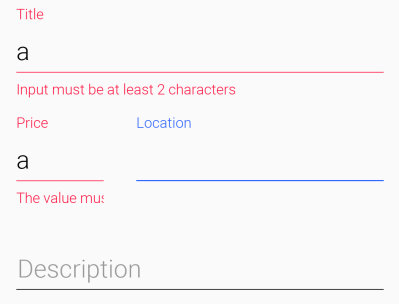


Figure 117. Error handling when there is some input (notice red title label)

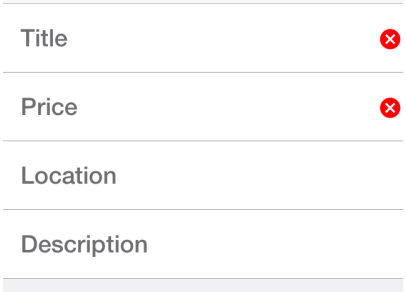


Figure 118. On iOS the situation hasn't changed much yet

The underlying system is the `errorMessage` method which you can chain like the other methods on `TextComponent` as such:

```
TextComponent tc = new TextComponent().
    label("Input Required").
    errorMessage("Input is essential in this field");
```

7.7.2. InputComponent and PickerComponent

To keep the code common and generic you use the `InputComponent` abstract base class and derive the other classes from that. `PickerComponent` is the other option.

A picker can work with your existing sample using code like this:

This produces the following which looks pretty standard:

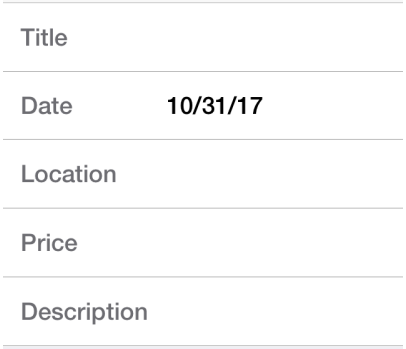


Figure 119. Picker component taking place in iOS

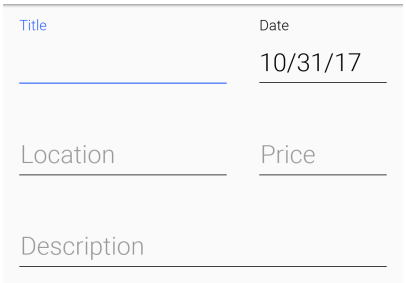


Figure 120. And in Android

The one tiny thing you should notice with the `PickerComponent` is that you don't construct the picker component using `new PickerComponent()`. Instead you use create methods such as `PickerComponent.createDate(new Date())`. The reason for that's that you have many types of pickers and it wouldn't make sense to have one constructor.

7.7.3. Underlying theme constants and UIID's

These varying looks are implemented through a combination of layouts, theme constants and UIID's. The most important UIID's are: `TextComponent`, `FloatingHint` & `TextHint`.

There are many related theme constants that can manipulate some pieces of this functionality:

- `textComponentErrorColor` a hex RGB color which defaults to null in which case this has no effect. When defined this will change the color of the border and label to the given color to match the material design styling. This implements the red border underline in cases of error and the label text color change
- `textComponentErrorLineBorderBool` toggles the material-style underline that appears on validation errors. Set it to `false` if you prefer to supply a different border when errors are shown
- `textComponentOnTopBool` toggles the on top mode which makes things look like they do on Android. This defaults to true on Android and false on other OSes. This can also be manipulated through the `onTopMode(boolean)` method in `InputComponent`, but the layout will use the theme constant
- `textComponentAnimBool` toggles the animation mode which again can be manipulated by a method in `InputComponent`. If you want to keep the UI static without the floating hint effect set this to false. Notice this defaults to true on Android
- `textComponentFieldUIID` sets the UIID of the text field to something other than `TextField` this is useful for platforms such as iOS where the look of the text field is different within the text component. This allows you to make the background of the text field transparent when it's within the `TextComponent` and make it different from the regular text field
- `inputComponentErrorMultilineBool` makes error labels multi-line by default so longer validation messages can wrap instead of being clipped

7.8. Button

Button [https://www.codenameone.com/javadoc/com/codename1/ui/Button.html] is a subclass of `Label` and as a result it inherits all its functionality, specifically icon placement, tickering, etc.

Button adds to the mix some more states such as a pressed UIID state and pressed icon.



Button supports more icon states such as rollover and disabled icon.

Button also exposes some functionality for subclasses specifically the setToggle method call which has no meaning when invoked on a Button but has a lot of implications for CheckBox & RadioButton.

Button event handling can be performed through an ActionListener [https://www.codenameone.com/javadoc/com/codename1/ui/events/ActionListener.html] or through a Command [https://www.codenameone.com/javadoc/com/codename1/ui/Command.html].



Changes in a Command won't be reflected into the Button after the command was set to the Button.

Here is a trivial hello world style Button:

```
Form hi = new Form("Button");
Button b = new Button("My Button");
hi.add(b);
b.addActionListener((e) -> Log.p("Clicked"));
```



Figure 121. Simple button in the iOS styling, notice iOS doesn't have borders on buttons...

Such a button can be styled to look like a link using code like this or by making these settings in the theme and using code such as btn.setUIID("Hyperlink"):

```
Form hi = new Form("Button");
Button b = new Button("Link Button");
b.getAllStyles().setBorder(Border.createEmpty());
b.getAllStyles().setTextDecoration(Style.TEXT_DECORATION_UNDERLINE);
hi.add(b);
b.addActionListener((e) -> Log.p("Clicked"));
```



Figure 122. Button styled to look like a link

7.8.1. Uppercase buttons

Buttons on Android's material design UI use upper case styling which isn't the case for iOS. To solve this you have the method `setCapsText(boolean)` in `Button` which has the corresponding `isCapsText`, `isCapsTextDefault` & `setCapsTextDefault(boolean)`. This is pretty core to Codename One so to prevent this from impacting everything unless you explicitly invoke `setCapsText(boolean)` the default value of `true` will apply when the UIID is `Button`, `RaisedButton` or for the built-in `Dialog` buttons.

You also have a theme constant: `capsButtonTextBool`. This constant controls caps text behavior from the theme and is set to true in the Android native theme.

7.8.2. Raised Button

Raised button is a style of button that's available on Android and used to highlight an important action within a form. To confirm with the material design UI guidelines you might want to leverage a raised button UI element on Android but use a regular button everywhere else.

First you need to know whether a raised button exists in the theme. On Android this will return true but on other OSes it will return false. A potential future update might make another platform true based on UI guidelines in other OSes.

For this purpose you have got the theme constant `hasRaisedButtonBool` which will return true on Android but will be false elsewhere. You can use it like this:

```
if(UIManager.getInstance().isThemeConstant("hasRaisedButtonBool", false)) {
    // that means we can use a raised button
}
```

To enable this you have the `RaisedButton` UIID that derives from `Button` and will act like it except for the places where `hasRaisedButtonBool` is true in which case it will look like this:

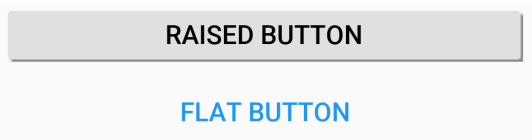


Figure 123. Raised and flat button in simulator

Notice that you can customize the colors of these buttons now since the border respects user colors...

In this case set the background color to purple and the foreground to white:



Figure 124. Purple raised button

```
Form f = new Form("Pixel Perfect", BoxLayout.y());
Button b = new Button("Raised Button", "RaisedButton");
Button r = new Button("Flat Button");
```

```
f.add(b);
f.add(r);
f.show();
```

7.8.3. Ripple effect

The ripple effect in material design highlights the location of the finger and grows as a circle to occupy the full area of the component as the user presses the button.

You can perform a ripple effect by darkening the touched area and growing that in a quick animation.

Ripple effect can be applied to any component but you have it turned on for buttons on Android which also applies to things like title commands, side menu elements etc. This might not apply at this moment to lead components like multi-buttons but that might change in the future.

`Component` has a property to enable the ripple effect `setRippleEffect(boolean)` and the corresponding `isRippleEffect()`. You can turn it on or off individually in the component level. For example, `Button` has static `setButtonRippleEffectDefault(boolean)` and `isButtonRippleEffectDefault()`. These allow you to define the default behavior for all the buttons and that can be configured through the theme constant `buttonRippleBool` which is on by default on the native Android theme.

7.9. CheckBox/RadioButton

CheckBox [https://www.codenameone.com/javadoc/com/codename1/ui/CheckBox.html] & RadioButton [https://www.codenameone.com/javadoc/com/codename1/ui/RadioButton.html] are subclasses of button that allow for either a toggle state or exclusive selection state.

Both `CheckBox` & `RadioButton` have a selected state that allows you to determine their selection.



`RadioButton` doesn't allow you to "deselect" it, the way to "deselect" a `RadioButton` is by selecting another `RadioButton`.

The `CheckBox` can be added to a `Container` like any other `Component` but the `RadioButton` must be associated with a `ButtonGroup` otherwise if you have more than one set of `RadioButton`s in the form you might have an issue.

Notice in the sample below that you associate all the radio buttons with a group but don't do anything with the group as the radio buttons keep the reference internally. You also show the opposite side functionality and icon behavior:

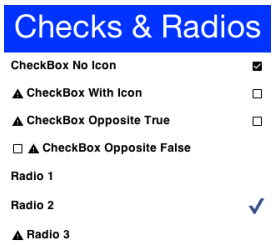


Figure 125. RadioButton & CheckBox usage

Both of these components can be displayed as toggle buttons (see the toggle button section below), or use the default check mark/filled circle appearance based on the type/OS.

7.9.1. Toggle Button

A toggle button is a button that's pressed and stays pressed. When a toggle button is pressed again it's released from the pressed state. Hence the button has a selected state to show if it's pressed or not like the `CheckBox`/`RadioButton` components in Codename One.

To turn any `CheckBox` or `RadioButton` to a toggle button use the `setToggle(true)` method. Or you can use the static `createToggle` method on both `CheckBox` and `RadioButton` to create a toggle button directly.



Invoking `setToggle(true)` implicitly converts the `UIID` to `ToggleButton` unless it was changed by the user from its original default value.

You can convert the sample above to use toggle buttons as such:



Figure 126. Toggle button converted sample

That's half the story though: to get the full effect of some cool toggle button UI's you can use a ComponentGroup [https://www.codenameone.com/javadoc/com/codename1/ui/ComponentGroup.html]. This allows you to create a button bar effect with the toggle buttons.

For example, to enclose the `CheckBox` components in a vertical `ComponentGroup` and the `RadioButton`s in a horizontal group, change the last line of the code above as such:

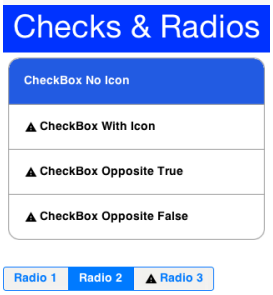


Figure 127. Toggle button converted sample wrapped in ComponentGroup

7.10. ComponentGroup

ComponentGroup [https://www.codenameone.com/javadoc/com/codename1/ui/ComponentGroup.html] is a special container that can be either horizontal or vertical (BoxLayout [https://www.codenameone.com/javadoc/com/codename1/ui/layouts/BoxLayout.html] `X_AXIS` or `Y_AXIS` respectively).

`ComponentGroup` "restyles" the elements within the group to have a `UIID` that allows you to create a "round border" effect that groups elements together.

The following code adds 4 component groups to a `Container` to show the various `UIID` changes:

```
hi.add("Three Labels").
        add(ComponentGroup.enclose(new Label("GroupElementFirst UIID"), new Label("GroupElement UIID"), new Label
("GroupElementLast UIID"))).
        add("One Label").
        add(ComponentGroup.enclose(new Label("GroupElementOnly UIID"))).
        add("Three Buttons").
        add(ComponentGroup.enclose(new Button("ButtonGroupFirst UIID"), new Button("ButtonGroup UIID"), new Button
("ButtonGroupLast UIID"))).
        add("One Button").
        add(ComponentGroup.enclose(new Button("ButtonGroupOnly UIID")));
```

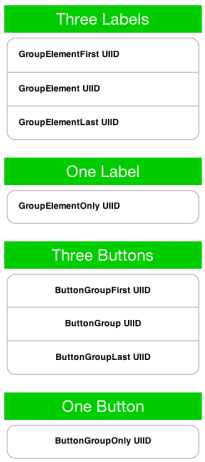


Figure 128. ComponentGroup adapts the UIID's of the components added so you can style them

Notice the following about the code above and the resulting image:

- Buttons have a different UIID than other element types. Their styling is slightly different in such

UI's so you need to pay attention to that.

- When an element is placed alone within a `ComponentGroup` its a special case `UIID`.



By default, `ComponentGroup` does **nothing**. You need to explicitly activate it in the theme by setting a theme property to true. Specifically you need to set `ComponentGroupBool` to `true` for `ComponentGroup` to do something otherwise its a box layout container! The `ComponentGroupBool` flag is true by default in the iOS native theme.

When `ComponentGroupBool` is set to true, the component group will change the styles of all components placed within it to match the element UIID given to it (GroupElement by default) with special caveats to the first/last/ elements. For example:

1. With one element in a component group it will have the UIID: `GroupElementOnly`
2. With two elements in a component group they will have the UIID's `GroupElementFirst`, `GroupElementLast`
3. With three elements in a component group they will have the UIID's `GroupElementFirst`, `GroupElement`, `GroupElementLast`
4. With four elements in a component group they will have the UIID's `GroupElementFirst`, `GroupElement`, `GroupElement`, `GroupElementLast`

This allows you to define special styles for the edges.

You can customize the `UIID` set by the component group by calling `setElementUIID` in the component group for example: `setElementUIID("ToggleButton")` for three elements result in the following `UIID`s:

`ToggleButtonFirst`, `ToggleButton`, `ToggleButtonLast`

7.11. MultiButton

MultiButton [https://www.codenameone.com/javadoc/com/codename1/components/MultiButton.html] is a composite component (lead component) that acts like a versatile Button [https://www.codenameone.com/javadoc/com/codename1/ui/Button.html]. It supports up to 4 lines of text (it doesn't automatically wrap the text), an emblem (usually navigational arrow, or check box) and an icon.

`MultiButton` can be used as a button, a `CheckBox` or a `RadioButton` for creating rich UI's.



The `MultiButton` was inspired by the aesthetics of the `UITableView` iOS component.

A common source of confusion in the `MultiButton` is the difference between the icon and the emblem, since both may have an icon image associated with them. The icon is an image representing the entry while the emblem is an optional visual representation of the action that will be undertaken when the element is pressed. Both may be used simultaneously or individually of one another:

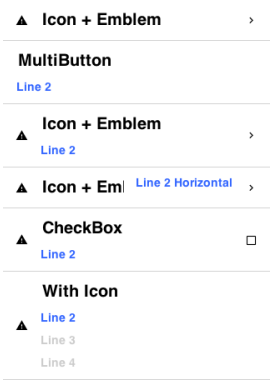


Figure 129. Multiple usage scenarios for the MultiButton

7.11.1. Styling the MultiButton

Since the `MultiButton` is a composite component setting its `UIID` will impact the top level UI.

To customize everything you need to customize the UIID's for `MultiLine1`, `MultiLine2`, `MultiLine3`, `MultiLine4` & `Emblem`.

You can customize the individual `UIID`s through the API directly using the `setIconUIID`, `setUIIDLine1`, `setUIIDLine2`, `setUIIDLine3`, `setUIIDLine4` & `setEmblemUIID`.

Recent versions also include a badge overlay that can be rendered in the corner of the main icon. Use `setBadgeText()` to display a value (for example a notification count) and `setBadgeUIID()` if you need a custom UIID instead of the default `Badge` styling. When you need to inspect or adjust the badge style programmatically, `getBadgeStyleComponent()` returns the component whose styles are applied to the badge so you can tweak padding, colors or borders before showing the `MultiButton`.

7.12. SpanButton

SpanButton [https://www.codenameone.com/javadoc/com/codename1/components/SpanButton.html] is a composite component (lead component) that looks/acts like a `Button` but can break lines rather than crop them when the text is long.

Unlike the `MultiButton` it uses the `TextArea` internally to break lines seamlessly. The `SpanButton` is far simpler than the `MultiButton` and as a result isn't as configurable:

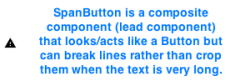


Figure 130. The SpanButton Component



`SpanButton` is slower than both `Button` and `MultiButton`. Recommend using it when there is a genuine need for its functionality.

7.13. SpanLabel

SpanLabel [https://www.codenameone.com/javadoc/com/codename1/components/SpanLabel.html] is a composite component (lead component) that looks/acts like a Label [https://www.codenameone.com/

javadoc/com/codename1/ui/Label.html] but can break lines rather than crop them when the text is long.

`SpanLabel` uses the `TextArea` internally to break lines seamlessly and so doesn't provide all the elaborate configuration options of `Label`.

One of the features of label that moved into `SpanLabel` to some extent is the ability to position the icon. For example, unlike a `Label` the icon position is determined by the layout manager of the composite so `setIconPosition` accepts a `BorderLayout` constraint:

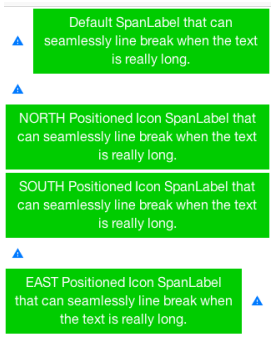


Figure 131. The SpanLabel Component



`SpanLabel` is slower than `Label`. Recommend using it when there is a genuine need for its functionality.

7.14. OnOffSwitch

The OnOffSwitch [https://www.codenameone.com/javadoc/com/codename1/components/OnOffSwitch.html] allows you to write an application where the user can swipe a switch between two states (on/off). This is a common UI paradigm in Android and iOS, although it's implemented in a radically different way in both platforms.

This is a rather elaborate component because of its unique design on iOS, but you're able to accommodate most of the small behaviors of the component into your version, and it seamlessly adapts between the Android style and the iOS style.

The image below was generated based on the default use of the `OnOffSwitch`:

```
OnOffSwitch onOff = new OnOffSwitch();
hi.add(onOff);
```

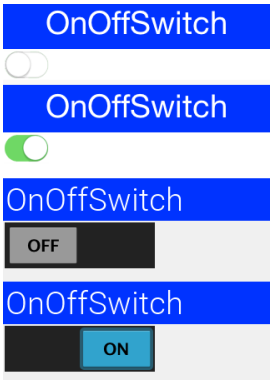


Figure 132. The OnOffSwitch component as it appears on/off on iOS (top) and on Android (bottom)

As you can understand the difference between the way iOS and Android render this component has triggered two different implementations within a single component. The Android implementation uses standard buttons and is the default for non-iOS platforms.



You can force the Android or iOS mode by using the theme constant `onOffIOSModeBool`.

7.14.1. Validation

Validation is an inherent part of text input, and the Validator [https://www.codenameone.com/javadoc/com/codename1/ui/validation/Validator.html] class allows that. You can enable validation through the `Validator` class to add constraints for a specific component. It's also possible to define components that would be enabled/disabled based on validation state and the way in which validation errors are rendered (change the components `UIID`, paint an emblem on top, etc.). A Constraint [https://www.codenameone.com/javadoc/com/codename1/ui/validation/Constraint.html] is an interface that represents validation requirements. You can define a constraint in Java or use some built-in constraints such as LengthConstraint [https://www.codenameone.com/javadoc/com/codename1/ui/validation/LengthConstraint.html], RegexConstraint [https://www.codenameone.com/javadoc/com/codename1/ui/validation/RegexConstraint.html], etc.

This sample below continues from the place where the TextField sample above stopped by adding validation to that code:

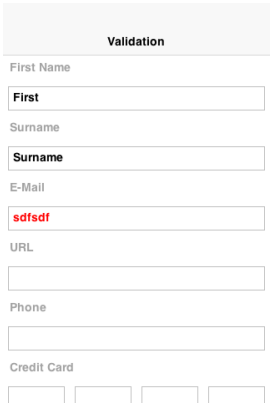


Figure 133. Validation & Regular Expressions

When the form components are bound to a model with `@Bindable`, the same constraints can be expressed as field annotations — `@Required`, `@Length`, `@Regex`, `@Email`, `@Url`, `@Numeric`, `@ExistIn`, `@Validate` — and the generated binder wires them into a `Validator` exposed through

`Binding#getValidator()`. See the Validation annotations section in the component binding chapter for the full reference.

7.15. InfiniteProgress

The InfiniteProgress [https://www.codenameone.com/javadoc/com/codename1/components/InfiniteProgress.html] indicator spins an image infinitely to show that a background process is still working.



This style of animation is often nicknamed "washing machine" as it spins endlessly.

`InfiniteProgress` can be used in one of two ways either by embedding the component into the UI through something like this:

`InfiniteProgress` can also appear over the entire screen, thus blocking all input. This tints the background while the infinite progress rotates:

```
Dialog ip = new InfiniteProgress().showInifiniteBlocking();

// do some long operation here using invokeAndBlock or do something in a separate thread and callback later
// when you are done just call

ip.dispose();
```

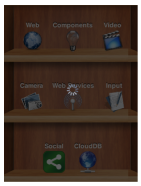


Figure 134. Infinite progress

The image used in the `InfiniteProgress` animation is defined by the native theme. You can override that definition either by defining the theme constant `infiniteImage` or by invoking the setAnimation [https://www.codenameone.com/javadoc/com/codename1/components/InfiniteProgress.html#setAnimation-com.codename1.ui.Image-] method.



Despite the name of the method `setAnimation` expects a static image that will be rotated internally. Don't use an animated image.

7.16. InfiniteScrollAdapter and InfiniteContainer

InfiniteScrollAdapter [https://www.codenameone.com/javadoc/com/codename1/components/InfiniteScrollAdapter.html] & InfiniteContainer [https://www.codenameone.com/javadoc/com/codename1/ui/InfiniteContainer.html] allow you to create a scrolling effect that "never" ends with the typical `Container`/`Component` paradigm.

The motivation behind these classes is simple, say you have a lot of data to fetch from storage or from the internet. You can fetch the data in batches and show progress sign while you do this.

Infinite scroll fetches the next batch of data dynamically as you reach the end of the `Container`. `InfiniteScrollAdapter` & `InfiniteContainer` represent two similar ways to do that task.

Let start by exploring how you can achieve this UI that fetches data from a webservice:

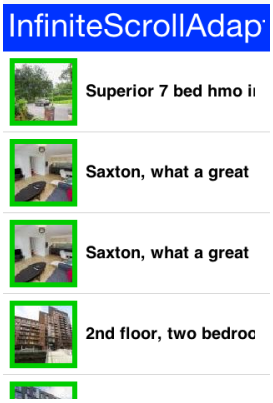


Figure 135. InfiniteScrollAdapter demo code fetching property cross data

The first step is creating the webservice call, you won't go into too much detail here as webservices & IO are discussed later in the guide:



The demo code here doesn't do any error handling! This is a bad practice and it's taken here to keep the code short and readable. Proper error handling is used in the Property Cross demo.

The `fetchPropertyData` is a simplistic tool that fetches the next page of listings for the nestoria webservice. Notice that this method is synchronous and will block the calling thread (legally) until the network operation completes.

Now that you have a webservice lets proceed to create the UI. Check out the code annotations below:



Don't violate the EDT in the callback. It's invoked on the event dispatch thread and it's crucial

7.16.1. The InfiniteContainer

InfiniteContainer [https://www.codenameone.com/javadoc/com/codename1/ui/InfiniteContainer.html] was introduced to simplify and remove some boilerplate of the `InfiniteScrollAdapter`. It takes a more traditional approach of inheriting the Container class to provide its functionality.

Unlike the `InfiniteScrollAdapter` the `InfiniteContainer` accepts an index and amount to fetch. This is useful for tracking your position but also important since the `InfiniteContainer` also implements Pull To Refresh as part of its functionality.

Converting the code above to an `InfiniteContainer` is pretty simple you moved all the code into the callback `fetchComponents` method and returned the array of `Component`s as a response.

Unlike the `InfiniteScrollAdapter` you can't use the `ContentPane` directly so you've to use a `BorderLayout` and place the `InfiniteContainer` there:

7.17. List, MultiList, renderers & models

7.17.1. InfiniteContainer/InfiniteScrollAdapter vs. List/ContainerList

Your recommendation is to always go with `Container`, `InfiniteContainer` or `InfiniteScrollAdapter`.

Recommend avoiding `List` or its subclasses/related classes specifically `ContainerList` & `MultiList`.



Recommend replacing `ComboBox` with `Picker` but that's a different discussion.

A `Container` with ~5000 nested containers within it can perform on par with a `List` and probably exceed its performance when used.

Larger sets of data are manageable on phones or tablets so the benefits for lists are dubious.

In API you found that even experienced developers experienced a great deal of pain when wrangling the Swing styled lists and their stateless approach.

Since animation, swiping and other capabilities that are so common in mobile are so hard to do with lists you see no actual reason to use them.

7.17.2. Why isn't list deprecated

You deprecated `ContainerList` which performs badly and has some inherent complexity issues. `List` has some unique use cases and is still used all over Codename One.

`MultiList` is a reasonable version of `List` that's far easier to use without most of the pains related to renderer configuration.

Some cases justify using `List` or `MultiList`, but they're rarer than usual, hence the recommendation.

7.17.3. MVC in lists

A Codename One List [https://www.codenameone.com/javadoc/com/codename1/ui/List.html] doesn't contain components, but rather arbitrary data; this seems odd at first but makes sense. If you want a list to contain components, use a Container.

The advantage of using a `List` in this way is that you can display it in many ways (for example: fixed focus positions, horizontally, etc.), and that you can have more than a million entries without performance overhead. You can also do some pretty nifty things, like filtering the list on the fly or fetching it dynamically from the Internet as the user scrolls down the list. To achieve these things the list uses two interfaces: ListModel [https://www.codenameone.com/javadoc/com/codename1/ui/list/ListModel.html] and ListCellRenderer. List [https://www.codenameone.com/javadoc/com/codename1/ui/List.html] model represents the data; its responsibility is to return the arbitrary object within the list at a given offset. Its second responsibility is to tell the list when the data changes, so the list can refresh.



Think of the model as an array of objects that can tell you when it changes.

The list renderer is like a rubber stamp that knows how to draw an object from the model, it's called many times per entry in an animated list and must be fast. Unlike standard Codename One components, it's used to draw the entry in the model and is discarded, hence it has no memory overhead, but if it takes too long to process a model value it can be a big bottleneck!



Think of the render as a translation layer that takes the "data" from the model and translates it to a visual representation.

This is all generic, but a bit too much for most, doing a list correctly requires some understanding. The main source of confusion for developers is the stateless nature of the list and the transfer of state to the model (for example: a checkbox list needs to listen to action events on the list and update the model, in order for the renderer to display that state). Once you understand that it's easy.

7.17.4. Understanding MVC

A quick recap of what MVC is:

- Model - Represents the data for the component (list), the model can tell you how many items are in it and which item resides at a given offset within the model. This differs from a simple `Vector` (or array), since all access to the model is controlled (the interface is simpler), and unlike a `Vector`/Array, the model can tell you of changes that occur within it.
- View - The view draws the content of the model. It's a "dumb" layer that has no notion of what's displayed and knows how to draw. It tracks changes in the model (the model sends events) and redraws itself when it changes.
- Controller - The controller accepts user input and performs changes to the model, which in turn cause the view to refresh.

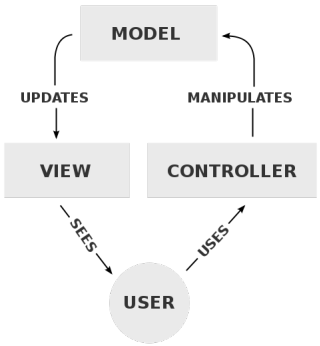


Figure 136. Typical MVC Diagram [2]

Codename One's List [https://www.codenameone.com/javadoc/com/codename1/ui/List.html] component uses the MVC paradigm in its implementation. `List` itself is the Controller (with a bit of the View mixed in). The ListCellRenderer [https://www.codenameone.com/javadoc/com/codename1/ui/list/ListCellRenderer.html] interface is the rest of the View and the ListModel [https://www.codenameone.com/javadoc/com/codename1/ui/list/ListModel.html] is (you guessed it by now) the Model.

When the list is painted, it iterates over the visible elements in the model and asks the model for the data, it then draws them using the renderer. Notice that because of this both the model and the renderer must be fast and that's hard.

Why is this useful

Since the model is a lightweight interface, it can be implemented by you and replaced in runtime if so desired, this allows many use cases:

1. A list can contain thousands of entries but load the portion visible to the user. Since the model will be queried for the elements that are visible to the user, it won't need to load the large data set into memory until the user starts scrolling down (at which point other elements may be offloaded from memory).
2. A list can cache efficiently. For example: a list can mirror data from the server into local RAM without actually downloading all the data. Data can also be mirrored from storage for better performance and discarded for better memory use.
3. The is no need for state copying. Since renderers allow you to display any object type, the list model interface can be implemented by the application's data structures (for example: persistence/network engine), which would return internal application data structures saving you the need of copying application state into a list specific data structure. Note that this advantage applies with a custom renderer which is pretty difficult to get right.
4. Using the proxy pattern you can layer logic such as filtering, sorting, caching, etc. On top of existing models without changing the model source code.
5. You can reuse generic models for many views, for example: a model that fetches data from the server can be initialized with different arguments, to fetch different data for different views. View objects in different Forms can display the same model instance in different view instances, thus they would update automatically when you change one global model.

Most of these use cases work best for lists that grow to a larger size, or represent complex data, which is what the list object is designed to do.

7.17.5. Important - lists and Layout managers

When working with lists, you want the list to handle the scrolling (otherwise it will perform badly). This means you should place the list in a non-scrollable container (no parent can be scrollable), notice that the content pane is scrollable by default, so you should disable that.

It's also recommended to place the list in the `CENTER` location of a BorderLayout [https://www.codenameone.com/javadoc/com/codename1/ui/layouts/BorderLayout.html] to produce the most effective results. For example:

7.17.6. MultiList & DefaultListModel

After this long start, here is the first sample of creating a list using the MultiList [https://www.codenameone.com/javadoc/com/codename1/ui/list/MultiList.html].

The `MultiList` is a preconfigured list that contains a ready made renderer with defaults that make sense for the most common use cases. It still retains most of the power available to the `List` component but reduces the complexity of one of the hardest things to grasp for most developers: rendering.

The full power of the `ListModel` is still available and allows you to create a million entry list with a

few lines of code. For example the objects that the model returns should always be in the form of `Map` objects and not an arbitrary object like the standard `List` allows.

Here is a simple example of a `MultiList` containing a highly popular subject matter:

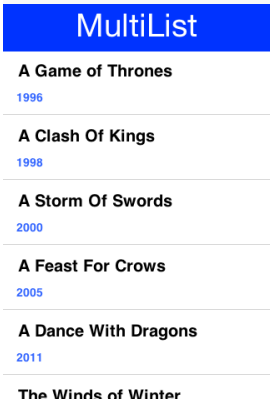


Figure 137. Basic usage of the MultiList & DefaultListModel]

`createListEntry` is trivial:

```
private Map<String, Object> createListEntry(String name, String date) {
    Map<String, Object> entry = new HashMap<>();
    entry.put("Line1", name);
    entry.put("Line2", date);
    return entry;
}
```

One major piece is missing here: the cover images for the books. A simple approach would be to place the image objects into the entries using the "icon" property as such:

```
private Map<String, Object> createListEntry(String name, String date, Image cover) {
    Map<String, Object> entry = new HashMap<>();
    entry.put("Line1", name);
    entry.put("Line2", date);
    entry.put("icon", cover);
    return entry;
}
```

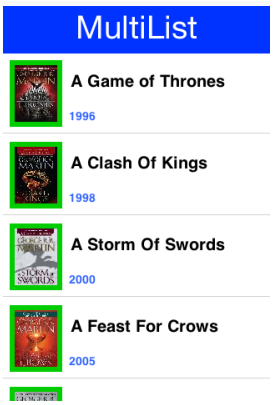


Figure 138. With cover images in place



Since the `MultiList` uses the `GenericListCellRenderer` internally you can use URLImage [https://www.codenameone.com/javadoc/com/codename1/ui/URLImage.html] to dynamically fetch the data. This is discussed in the graphics section of this guide.

Going further with the ListModel

Suppose that GRRM [http://www.georgerrmartin.com/] was prolific and wrote 1 million books. The default list model won't make much sense in that case but you would still be able to render everything in a list model.

You will fake it a bit but notice that 1M components won't be created even if you somehow scroll all the way down...

The ListModel [https://www.codenameone.com/javadoc/com/codename1/ui/list/ListModel.html] interface can be implemented by anyone in this case you did a stupid simple implementation:

You can now replace the existing model by removing all the model related logic and changing the constructor call as such:

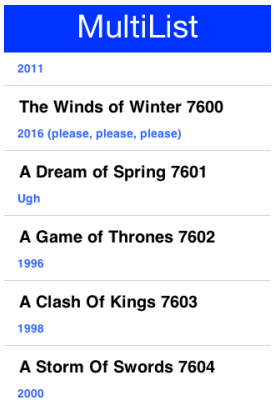


Figure 139. It took ages to scroll this far... This goes to a million...

7.17.7. List cell renderer

The Renderer is a simple interface with 2 methods:

The most simple/naive implementation may choose to implement the renderer as follows:

This will compile and work, but won't give you much, notice that you won't see the `List` selection move on the List, this is because the renderer returns a Label [https://www.codenameone.com/javadoc/com/codename1/ui/Label.html] with the same style regardless if it's selected or not.

Now make it a bit more useful:

In this renderer you set the `Label.setFocus(true)` if it's selected, calling to this method doesn't give the focus to the Label, it renders the label as selected.

Then you invoke `Label.getAllStyles().setBgTransparency(100)` to give the selection semi transparency, and `0` for full transparency if not selected.

That's still not efficient because you create a new `Label` each time the method is invoked.

To make the code tighter, keep a reference to the `Component` or extend it as DefaultListCellRenderer [https://www.codenameone.com/javadoc/com/codename1/ui/list/DefaultListCellRenderer.html] does:

Now look at a more advanced Renderer:

In this renderer you want to render a `Contact` object to the Screen, you build the `Component` in the constructor and in the getListCellRendererComponent you update the Labels' texts according to the `Contact` object.

Notice that in this renderer you return a focus `Label` with semi transparency, as mentioned before, the focus component can be modified within this method.

For example, you can change the focus `Component` to have an icon:

7.17.8. Generic list cell renderer

As part of the GUI builder work, you needed a way to customize rendering for a List, but the renderer/model approach seemed impossible to adapt to a GUI builder (it seems the Swing GUI builders had a similar issue). Your solution was to introduce the `GenericListCellRenderer`, which while introducing limitations and implementation requirements still manages to make life easier, both in the GUI builder and outside of it.

GenericListCellRenderer [https://www.codenameone.com/javadoc/com/codename1/ui/list/GenericListCellRenderer.html] is a renderer designed to be as simple to use as a `Component-Container` hierarchy, you effectively crammed most of the common renderer use cases into one class. To enable that, you need to know the content of the objects within the model, so the `GenericListCellRenderer` assumes the model contains `Map` objects. Since `Maps` can contain arbitrary data the list model is still generic and allows storing application specific data. Furthermore a `Map` can still be derived and extended to provide domain specific business logic.

The `GenericListCellRenderer` accepts two container instances (more later on why at least two, and not one), which it maps to individual `Map` entries within the model, by finding the appropriate components within the given container hierarchy. Components are mapped to the `Map` entries based on the name property of the component (`getName`/`setName`) and the key/value within the `Map`, for example:

For a model that contains a `Map` entry like this:

```
"Foo": "Bar"
"X": "Y"
"Not": "Applicable"
"Number": Integer(1)
```

A renderer will loop over the component hierarchy in the container, searching for components whose name matches Foo, X, Not, and Number, and assigning the appropriate value to them.



You can also use image objects as values, and they will be assigned to labels as expected. For example, you can't assign both an image and a text to a single label, since the key will be taken. That isn't a big problem, since two labels can be used in

> such a renderer.

To make matters even more attractive the renderer seamlessly supports list tickering when appropriate, and if a CheckBox [https://www.codenameone.com/javadoc/com/codename1/ui/CheckBox.html] appears within the renderer, it will toggle a boolean flag within the `Map` seamlessly.

One issue that crops up with this approach is that, if a value is missing from the `Map`, it's treated as empty and the component is reset.

This can pose an issue if you hardcode an image or text within the renderer and you don't want them replaced (for example: an arrow graphic on a `Label` within the renderer). The solution for this is to name the component with Fixed in the end of the name, for example: `HardcodedIconFixed`.

Naming a component within the renderer with $number will automatically set it as a counter component for the offset of the component within the list.

Styling the `GenericListCellRenderer` is slightly different, the renderer uses the `UIID` of the `Container` passed to the generic list cell renderer, and the background focus uses that same `UIID` with the word "Focus" appended to it.

Note that the generic list cell renderer will grant focus to the child components of the selected entry if they're focusable, thus changing the style of said entries. For example: a Container [https://www.codenameone.com/javadoc/com/codename1/ui/Container.html] might have a child `Label` that has one style when the parent container is unselected and another when it's selected (focused), this can be achieved by defining the label as focusable. Notice that the component will never receive direct focus, since it's still part of a renderer.

Finally, the generic list cell renderer accepts two or four instances of a Container, rather than the obvious choice of accepting one instance. This allows the renderer to treat the selected entry differently, which is important to tickering, although it's also useful for the fisheye effect [3]. Since it might not be practical to seamlessly clone the `Container` for the renderer's needs, Codename One expects the developer to provide two separate instances, they can be identical in all respects, but they must be separate instances for tickering to work. The renderer also allows for a fisheye effect, where the selected entry is actually different from the unselected entry in its structure, it also allows for a pinstripe effect, where odd/even rows have different styles (this is accomplished by providing 4 instances of the containers selected/unselected for odd/even).

The best way to learn about the generic list cell renderer and the `Map` model is by playing with them in the old GUI builder. Notice they can be used in code without any dependency on the GUI builder and can be useful at that.

Here is a simple example of a list with checkboxes that gets updated automatically:

```
list.setRenderer(new GenericListCellRenderer(createGenericRendererContainer(), createGenericRendererContainer()));
```

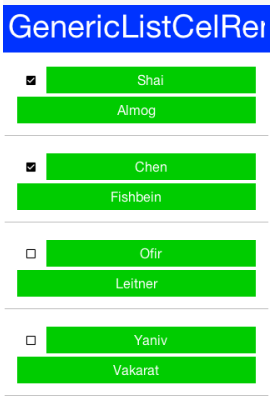


Figure 140. GenericListCellRenderer demo code

Custom UIID of entry in GenenricListCellRenderer/MultiList

With MultiList [https://www.codenameone.com/javadoc/com/codename1/ui/list/MultiList.html]/`GenenricListCellRenderer` one of the common issues is making a UI where a specific component within the list renderer has a different UIID style based on data. For example: this can be helpful to mark a label within the list as red, for instance, for a list of monetary transactions.

This can be achieved with a custom renderer, but that's a pretty difficult task.
`GenericListCellRenderer` (`MultiList` uses `GenericListCellRenderer` internally) has another option.

Normally, to build the model for a renderer of this type, you use something like:

What if you want componentName to be red? Just use:

This will apply the UIID "red" to the component, which you can then style in the theme. Notice that once you start doing this, you need to define this entry for all entries, for example:

Otherwise the component will stay red for the next entry (since the renderer acts like a rubber stamp).

Rendering prototype

Because of the rendering architecture of a `List` its pretty hard to calculate the right preferred size for such a component. The default behavior includes querying a few entries from the model then constructing their renderers to get a "sample" of the preferred size value.

As you might guess this triggers a performance penalty that's paid with every reflow of the UI. The solution is to use `setRenderingPrototype`.

`setRenderingPrototype` accepts a "fake" value that represents a reasonably large amount of data and it will be used to calculate the preferred size. For example: for a multiList that should render 2 lines of text with 20 characters and a 5mm square icon you can do something like this:

7.17.9. ComboBox

The ComboBox [https://www.codenameone.com/javadoc/com/codename1/ui/ComboBox.html] is a specialization of `List` that displays a single selected entry. When clicking that entry a popup is presented allowing the user to pick an entry from the full list of entries.



The ComboBox UI paradigm isn't as common on OSes such as iOS where there is no native equivalent to it. Recommend using either the Picker [https://www.codenameone.com/javadoc/com/codename1/ui/spinner/Picker.html] class or the AutoCompleteTextField [https://www.codenameone.com/javadoc/com/codename1/ui/AutoCompleteTextField.html].

ComboBox is notoriously hard to style as it relies on a complex dynamic of popup renderer and instantly visible renderer. The UIID for the ComboBox is ComboBox but if you set it to something else all the other UIIDs will also change their prefix. For example: the ComboBoxPopup UIID will become MyNewUIIDPopup.

The combo box defines the following UIID's by default:

- ComboBox
- ComboBoxItem
- ComboBoxFocus
- PopupContentPane
- PopupItem
- PopupFocus

The ComboBox also defines theme constants that allow some native themes to manipulate its behavior for example:

- popupTitleBool - shows the "label for" value as the title of the popup dialog
- popupCancelBodyBool - Adds a cancel button into the popup dialog
- centeredPopupBool - shows the popup dialog in the center of the screen instead of under the popup
- otherPopupRendererBool - Uses a different list cell render for the popup than the one used for the ComboBox itself. When this is false PopupItem & PopupFocus become irrelevant. Notice that the Android native theme defines this to true.

Since a ComboBox is a List you can use everything you learned about a List to build a ComboBox including models, GenericListCellRenderer etc.

For example: the demo below uses the GRRM demo data from above to build a ComboBox:

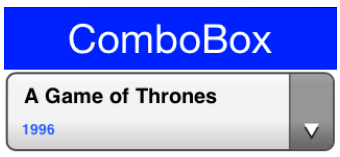


Figure 141. GRRM ComboBox

7.18. Slider

A Slider [https://www.codenameone.com/javadoc/com/codename1/ui/Slider.html] is an empty component that can be filled horizontally or vertically to allow indicating progress, setting volume etc. It can be

editable to allow the user to determine its value or none editable to relay that information to the user. It can have a thumb on top to show its current position.



Figure 142. Slider

The interesting part about the slider is that it has two separate style UIID□s, `Slider` & `SliderFull`. The `Slider` UIID is always painted and `SliderFull` is rendered on top based on the amount the `Slider` should be filled.

`Slider` is highly customizable for example: a slider can be used to replicate a 5-star rating widget as such. Notice that this slider will work when its given its preferred size otherwise more stars will appear. That's why you place it within a `FlowLayout`:

The slider itself is initialized in the code below. Notice that you can achieve almost the same result using a theme by setting the `Slider` & `SliderFull` UIID's (both in selected & unselected states).

In fact doing this in the theme might be superior as you could use one image that contains 5 stars already and that way you won't need the preferred size hack below:

```
private void initStarRankStyle(Style s, Image star) {
    s.setBackgroundType(Style.BACKGROUND_IMAGE_TILE_BOTH);
    s.setBorder(Border.createEmpty());
    s.setBgImage(star);
    s.setBgTransparency(0);
}

private Slider createStarRankSlider() {
    Slider starRank = new Slider();
    starRank.setEditable(true);
    starRank.setMinValue(0);
    starRank.setMaxValue(10);
    Font fnt = Font.createTrueTypeFont("native:MainLight", "native:MainLight").
            derive(Display.getInstance().convertToPixels(5, true), Font.STYLE_PLAIN);
    Style s = new Style(0xffff33, 0, fnt, (byte)0);
    Image fullStar = FontImage.createMaterial(FontImage.MATERIAL_STAR, s).toImage();
    s.setOpacity(100);
    s.setFgColor(0);
    Image emptyStar = FontImage.createMaterial(FontImage.MATERIAL_STAR, s).toImage();
    initStarRankStyle(starRank.getSliderEmptySelectedStyle(), emptyStar);
    initStarRankStyle(starRank.getSliderEmptyUnselectedStyle(), emptyStar);
    initStarRankStyle(starRank.getSliderFullSelectedStyle(), fullStar);
    initStarRankStyle(starRank.getSliderFullUnselectedStyle(), fullStar);
    starRank.setPreferredSize(new Dimension(fullStar.getWidth() * 5, fullStar.getHeight()));
    return starRank;
}
private void showStarPickingForm() {
  Form hi = new Form("Star Slider", new BoxLayout(BoxLayout.Y_AXIS));
  hi.add(FlowLayout.encloseCenter(createStarRankSlider()));
  hi.show();
}
```



Figure 143. Star Slider set to 5 (its between 0 - 10)



This slider goes all the way to 0 stars which is less common. You can use a `Label` to represent the first star and have the slider work between 0 - 8 values to provide 4 more stars.

7.19. Table

Table [https://www.codenameone.com/javadoc/com/codename1/ui/table/Table.html] is a composite component (but it isn't a lead component), this means it's a subclass of Container [https://www.codenameone.com/javadoc/com/codename1/ui/Container.html]. It's effectively built from many components.



`Table` is based on the TableLayout [https://www.codenameone.com/javadoc/com/codename1/ui/table/TableLayout.html] class. It's important to be familiar with that layout manager when working with `Table`.

Here is a trivial sample of using the standard table component:

```
Form hi = new Form("Table", new BorderLayout());
TableModel model = new DefaultTableModel(
        new String[] {"Col 1", "Col 2", "Col 3"},
        new Object[][] {
            {"Row 1", "Row A", "Row X"},
            {"Row 2", "Row B", "Row Y"},
            {"Row 3", "Row C", "Row Z"},
            {"Row 4", "Row D", "Row K"},
        }) {
    public boolean isCellEditable(int row, int col) {
        return col != 0;
    }
};
Table table = new Table(model);
hi.add(BorderLayout.CENTER, table);
hi.show();
```

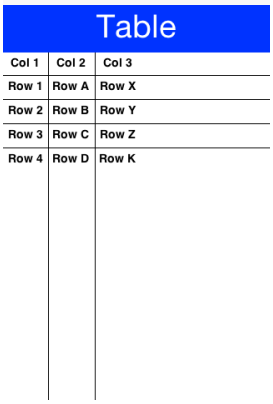


Figure 144. Simple Table usage



In the sample above the title area and first column aren't editable. The other two columns are editable.

The more "interesting" capabilities of the `Table` class can be utilized through the `TableLayout`. You can use the layout constraints (also exposed in the table class) to create spanning and elaborate UI's.

For example:

```
Form hi = new Form("Table", new BorderLayout());
TableModel model = new DefaultTableModel(
        new String[] {"Col 1", "Col 2", "Col 3"},
        new Object[][] {
            {"Row 1", "Row A", "Row X"},
            {"Row 2", "Row B can now stretch", null},
            {"Row 3", "Row C", "Row Z"},
            {"Row 4", "Row D", "Row K"},
        }) {
    public boolean isCellEditable(int row, int col) {
        return col != 0;
    }
};
Table table = new Table(model) {
    @Override
    protected TableLayout.Constraint createCellConstraint(Object value, int row, int column) {
        TableLayout.Constraint con = super.createCellConstraint(value, row, column);
        if (row == 1 && column == 1) {
            con.setHorizontalSpan(2);
        }
        con.setWidthPercentage(33);
        return con;
    }
};
hi.add(BorderLayout.CENTER, table);
hi.show();
```

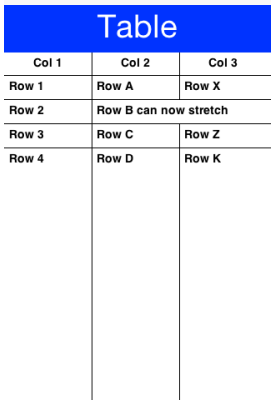


Figure 145. Table with spanning & fixed widths to 33%

To customize the table cell behavior you can derive the `Table` to create a "renderer like" widget, but unlike the list this component is "kept" and used as is. This means you can bind listeners to this component and work with it as you would with any other component in Codename One.

The example above can be extended to include far more capabilities:

```
Form hi = new Form("Table", new BorderLayout());
TableModel model = new DefaultTableModel(
        new String[] {"Col 1", "Col 2", "Col 3"},
        new Object[][] {
            {"Row 1", "Row A", "Row X"},
            {"Row 2", "Row B can now stretch", null},
            {"Row 3", "Row C", "Row Z"},
            {"Row 4", "Row D", "Row K"},
        }) {
    public boolean isCellEditable(int row, int col) {
        return col != 0;
    }
};
Table table = new Table(model) {
    @Override
    protected Component createCell(Object value, int row, int column, boolean editable) {
        Component cell;
        if (row == 1 && column == 1) {
            Picker p = new Picker();
            p.setType(Display.PICKER_TYPE_STRINGS);
            p.setStrings("Row B can now stretch", "This is a good value",
                    "So Is This", "Better than text field");
            p.setSelectedString((String)value);
            p.setUIID("TableCell");
            p.addActionListener(e -> getModel().setValueAt(row, column, p.getSelectedString()));
            cell = p;
        } else {
            cell = super.createCell(value, row, column, editable);
        }
        if (row > -1 && row % 2 == 0) {
            cell.getAllStyles().setBgColor(0xeeeeee);
            cell.getAllStyles().setBgTransparency(255);
        }
        return cell;
    }
```

```
    @Override
    protected TableLayout.Constraint createCellConstraint(Object value, int row, int column) {
        TableLayout.Constraint con = super.createCellConstraint(value, row, column);
        if (row == 1 && column == 1) {
            con.setHorizontalSpan(2);
        }
        con.setWidthPercentage(33);
        return con;
    }
};
hi.add(BorderLayout.CENTER, table);
hi.show();
```

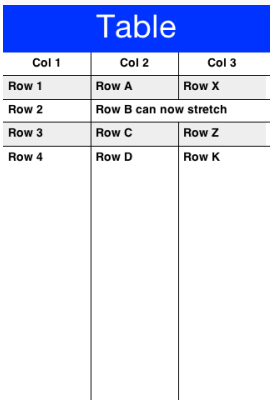


Figure 146. Table with customize cells using the pinstripe effect

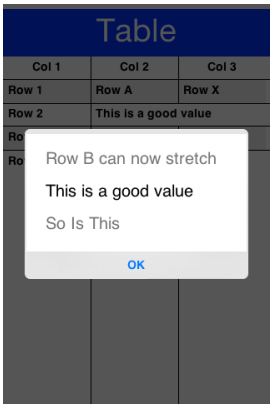


Figure 147. Picker table cell during edit

To line wrap table cells you can override the `createCell` method and return a TextArea [https://www.codenameone.com/javadoc/com/codename1/ui/TextArea.html] instead of a TextField [https://www.codenameone.com/javadoc/com/codename1/ui/TextField.html] since the `TextArea` defaults to the multi-line behavior this should work seamlessly. For example:

```
Form hi = new Form("Table", new BorderLayout());
TableModel model = new DefaultTableModel(
        new String[] {"Name", "Description", "Status"},
        new Object[][] {
            {"Row 1", "A short value", "OK"},
            {"Row 2", "A much longer value that wraps to multiple lines", "OK"},
            {"Row 3", "Another long table cell that needs wrapping", "Pending"},
        });
```

```
Table table = new Table(model) {
    @Override
    protected Component createCell(Object value, int row, int column, boolean editable) {
        if (row > -1 && column == 1) {
            TextArea cell = new TextArea(String.valueOf(value));
            cell.setUIID("TableCell");
            cell.setEditable(false);
            cell.setGrowByContent(true);
            return cell;
        }
        return super.createCell(value, row, column, editable);
    }

    @Override
    protected TableLayout.Constraint createCellConstraint(Object value, int row, int column) {
        TableLayout.Constraint con = super.createCellConstraint(value, row, column);
        con.setWidthPercentage(column == 1 ? 50 : 25);
        return con;
    }
};
hi.add(BorderLayout.CENTER, table);
hi.show();
```



Notice that you don't need to do anything else as binding to the `TextArea` is built-in to the `Table`.



You must set the column width constraint when you want multi-line to work. Otherwise the preferred size of the column might be too wide and the remaining columns might not have space left.

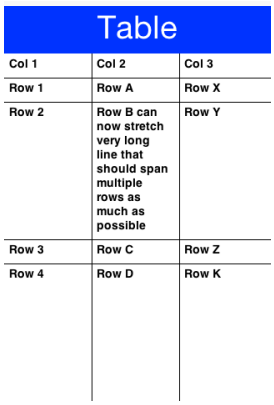


Figure 148. Multiline table cell in portrait mode

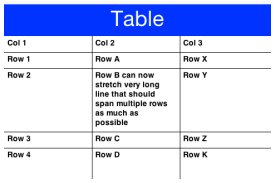


Figure 149. Multiline table cell in landscape mode. Notice the cell row count adapts seamlessly

7.19.1. Sorting tables

Sorting tables by clicking the titles is something that should work out of the box by using an API like `setSortSupported(true)`:

```
Form hi = new Form("Table", new BorderLayout());
TableModel model = new DefaultTableModel(
        new String[] {"Col 1", "Col 2", "Col 3"},
        new Object[][] {
            {"Row 1", "Row A", 1},
            {"Row 2", "Row B", 4},
            {"Row 3", "Row C", 7.5},
            {"Row 4", "Row D", 2.24},
        });
Table table = new Table(model);
table.setSortSupported(true);
hi.add(BorderLayout.CENTER, table);
hi.add(BorderLayout.NORTH, new Button("Button"));
hi.show();
```

Notice this works with numbers, Strings and might work with dates but you can support any object type by overriding the method `protected Comparator createColumnSortComparator(int column)` which should return a comparator for your custom object type in the column.

7.20. Tree

Tree [https://www.codenameone.com/javadoc/com/codename1/ui/tree/Tree.html] allows displaying hierarchical data such as folders and files in a collapsible/expandable UI. Like the Table it's a composite component (but it isn't a lead component). Like the `Table` it works in consort with a model to construct its user interface on the fly but doesn't use a stateless renderer (as `List` does).

The data of the `Tree` arrives from a model for example: this:

Will result in this:

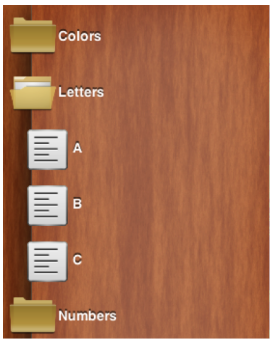


Figure 150. Tree



Since `Tree` is hierarchy based you can't have a simple model like you've for the `Table` as deep hierarchy is harder to represent with arrays.

A more practical "real world" example would be working with XML data. You can use something like this to show an XML `Tree`:



The `try(Stream)` syntax is a try with resources clogic that implicitly closes the stream.



Figure 151. XML Tree

The model for the XML hierarchy is implemented as such:

7.21. ShareButton

ShareButton [https://www.codenameone.com/javadoc/com/codename1/components/ShareButton.html] is a button you can add into the UI to let a user share an image or block of text.

The `ShareButton` uses a set of predefined share options on the simulator. On Android & iOS the `ShareButton` is mapped to the OS native sharing functionality and can share the image/text with the services configured on the device (for example: Twitter, Facebook etc.).



Sharing text is trivial but to share an image you need to save it to the FileSystemStorage [https://www.codenameone.com/javadoc/com/codename1/io/FileSystemStorage.html]. Notice that saving to Storage **won't work**!

In the sample code below you take a screenshot which is saved to FileSystemStorage [https://www.codenameone.com/javadoc/com/codename1/io/FileSystemStorage.html] for sharing:

```
Form hi = new Form("ShareButton");
ShareButton sb = new ShareButton();
sb.setText("Share Screenshot");
hi.add(sb);

Image screenshot = Image.createImage(hi.getWidth(), hi.getHeight());
hi.revalidate();
hi.setVisible(true);
hi.paintComponent(screenshot.getGraphics(), true);

String imageFile = FileSystemStorage.getInstance().getAppHomePath() + "screenshot.png";
try(OutputStream os = FileSystemStorage.getInstance().openOutputStream(imageFile);) {
```

```
    ImageIO.getImageIO().save(screenshot, os, ImageIO.FORMAT_PNG, 1);
} catch(IOException err) {
    Log.e(err);
}
sb.setImageToShare(imageFile, "image/png");
```

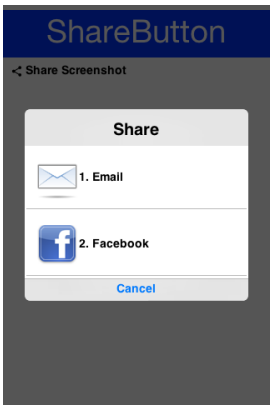


Figure 152. The share button running on the simulator

`ShareButton` behaves differently on the device…

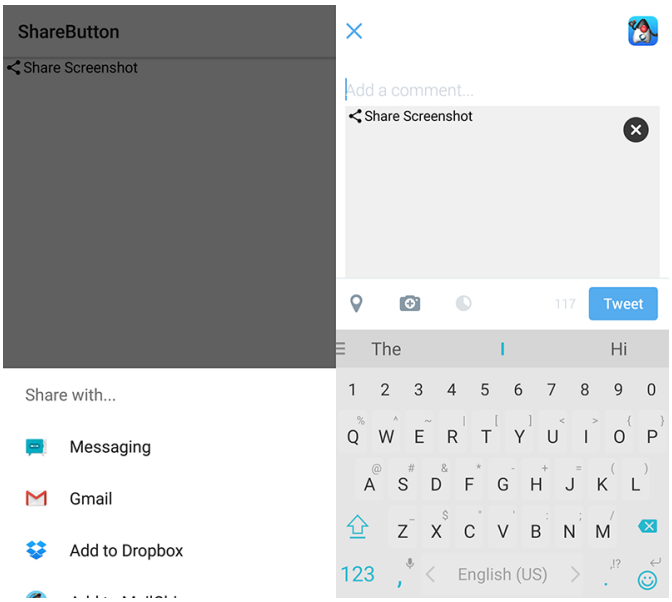


Figure 153. The share button running on the Android device and screenshot sent into twitter

The `ShareButton` features some share service classes to allow plugging in more share services. For example, this functionality is relevant to devices where native sharing isn't supported. This code isn't used on iOS/Android…

7.21.1. Share Result Callback

The share API can report what the user did with the share sheet. Register a `ShareResultListener` on either `ShareButton` or directly with `Display.share(…)` and you receive a single `ShareResult` describing the outcome:

- `SHARED_TO(packageName)` — the user picked a destination. `packageName` is the chosen target's identifier: an Android package name (for example, `com.whatsapp`) or, on iOS, a `UIActivityType`

such as `com.apple.UIKit.activity.PostToTwitter`. It may be `null` when the platform doesn't expose the selection (for example, older Android, Web Share API).

- `DISMISSED` — the user cancelled without picking a target. iOS reports this reliably; Android's chooser has no public dismissal signal, so on Android the listener simply doesn't fire on cancel.
- `FAILED` — the share couldn't be completed. `getError()` may carry a short platform-supplied message.

The listener is always invoked on the EDT, exactly once per share request.

```
ShareButton sb = new ShareButton();
sb.setTextToShare("Check this out!");
sb.setShareResultListener(result -> {
    if (result.isSharedTo()) {
        Log.p("Shared to " + result.getPackageName());
    } else if (result.isDismissed()) {
        Log.p("User dismissed the share sheet");
    } else if (result.isFailed()) {
        Log.p("Share failed: " + result.getError());
    }
});
form.add(sb);
```

If you call `Display` directly instead of using `ShareButton`, pass the listener as the final argument to the new overload:

When the share is dispatched through the non-native fallback dialog (platforms without `isNativeShareSupported`), the listener still fires: it reports `DISMISSED` when the user picks the Cancel button, and the underlying `ShareService` implementations report `SHARED_TO` once they call `ShareService.finish()`. Custom `ShareService` subclasses can call `deliverResult(ShareResult.failed("…"))` from inside `share(…)` to publish a failure explicitly; otherwise `finish()` falls back to a default `SHARED_TO(commandName)`.

7.22. iOS Share Extension Authoring Helper

An iOS **share extension** is a separate app target that the system shows inside `UIActivityViewController` when the user shares from another app (Safari, Photos, etc.). It runs in a sandboxed process and persists payloads to the host app via an **App Group** shared `NSUserDefaults` suite.

Codename One's build pipeline already wires `.ios.appext` zip archives under `src/main/resources` into the generated Xcode project. The `IOSShareExtensionBuilder` helper generates that archive for you so you don't have to create the extension target in Xcode first.

7.22.1. What the helper produces

For a single call the helper writes four files at the root of the extension bundle:

- `Info.plist` — declares the `NSExtension` dictionary, activation rules (`SupportsText` / `SupportsWebURL` / `SupportsImage`), and the principal class (`ShareViewController`).
- `<ExtensionName>.entitlements` — declares `com.apple.security.application-groups` with the configured App Group identifier.
- `ShareViewController.swift` — a `SLComposeServiceViewController` subclass that extracts text / URL / image attachments from `extensionContext`, packs them into a dictionary, and writes that dictionary to `UserDefaults(suiteName: appGroupId)` under the key `cn1.shareExtension.payload`.
- `buildSettings.properties` — Xcode build setting overrides (deployment target, Swift version, entitlements path, plist path) picked up by the iOS build pipeline.

7.22.2. Generating the bundle

Call the builder from a Maven plugin, an Ant task or a one-shot `main`:

The next iOS build picks up the `.ios.appext` archive automatically; no Xcode steps are required.

If you prefer a directory layout (for example, to commit the generated sources or to feed a future `ios/app_extensions/` pipeline) use `writeTo(File)` instead of `writeAppext(File)` — the file contents are identical.

7.22.3. Reading the payload from the host app

The host app reads the most recent payload from the same App Group at startup or when it resumes:

```
// Inside an iOS native interface
NSUserDefaults* shared =
    [[NSUserDefaults alloc] initWithSuiteName:@"group.com.example.myapp.shared"];
NSDictionary* payload = [shared dictionaryForKey:@"cn1.shareExtension.payload"];
```

The payload dictionary contains:

- `text` — the user-composed text (`String`).
- `items` — an array of `{ kind, value }` dictionaries where `kind` is `"text"`, `"url"` or `"image"`. For images, `value` is a file URL inside the App Group container.
- `timestamp` — a `Double` UNIX time.

Expose this to your CN1 code via a small native interface and clear the key once handled so the next share is unambiguous.

7.22.4. Activation rules and validation

The builder validates its inputs before writing anything:

- `extensionName` must be a non-empty identifier (letters, digits, _, -). It becomes the Xcode target name and the `.appex` bundle name. The extension's bundle id is set to `<hostBundleId>.<extensionName>` by the iOS build pipeline.

- `appGroupId` must start with `group.` (an Apple requirement). Create the matching App Group in your Apple Developer account and add it to both the host app's and the extension's provisioning profile.
- At least one of `acceptText`, `acceptURLs`, `acceptImages` must be enabled, otherwise iOS will never present your extension.

The default deployment target is iOS 12.0; override with `setDeploymentTarget("13.0")` if you need newer APIs in your customized controller.

7.22.5. Customising the generated controller

The generated Swift controller covers the common case (read attachments, persist to App Group, finish the request). If you need bespoke logic — custom UI, server upload, image transcoding — treat the helper's output as a starting point: run it once, copy the directory under `ios/app_extensions/MyShareExtension/`, edit `ShareViewController.swift`, and zip it back into `.ios.appext` (or wait for the directory-based pipeline tracked by issue #3427).



Provisioning profiles for the extension target aren't generated by the helper. You still need to provision the extension's bundle id (`<hostBundleId>.<extensionName>`) and add the App Group capability to it on the Apple Developer portal, alongside the host app.

7.23. Tabs

The Tabs [https://www.codenameone.com/javadoc/com/codename1/ui/Tabs.html] `Container` arranges components into groups within "tabbed" containers. `Tabs` is a container type that allows leafing through its children using labeled toggle buttons. The tabs can be placed in many different ways (top, bottom, left or right) with the default being determined by the platform. This class also allows swiping between components to leaf between said tabs (for this purpose the tabs themselves can also be hidden).

Since `Tabs` are a `Container` its a common mistake to try and add a `Tab` using the `add` method. That method won't work since a `Tab` can have both an `Image` and text String associated with it:

```
Form hi = new Form("Tabs", new BorderLayout());

Tabs t = new Tabs();
Style s = UIManager.getInstance().getComponentStyle("Tab");
FontImage icon1 = FontImage.createMaterial(FontImage.MATERIAL_QUESTION_ANSWER, s);

Container container1 = BoxLayout.encloseY(new Label("Label1"), new Label("Label2"));
t.addTab("Tab1", icon1, container1);
t.addTab("Tab2", new SpanLabel("Some text directly in the tab"));

hi.add(BorderLayout.CENTER, t);
```

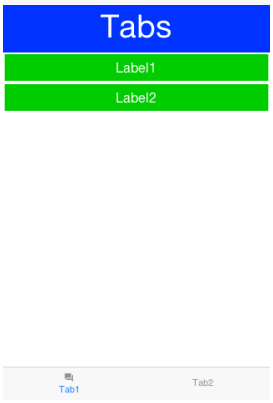


Figure 154. Simple usage of Tabs

A common usage for Tabs is the swipe to proceed effect which is common in iOS applications. In the code below you use RadioButton [https://www.codenameone.com/javadoc/com/codename1/ui/RadioButton.html] and LayeredLayout [https://www.codenameone.com/javadoc/com/codename1/ui/layouts/LayeredLayout.html] with hidden tabs to produce that effect:

```
Form hi = new Form("Swipe Tabs", new LayeredLayout());
Tabs t = new Tabs();
t.hideTabs();

Style s = UIManager.getInstance().getComponentStyle("Button");
FontImage radioEmptyImage = FontImage.createMaterial(FontImage.MATERIAL_RADIO_BUTTON_UNCHECKED, s);
FontImage radioFullImage = FontImage.createMaterial(FontImage.MATERIAL_RADIO_BUTTON_CHECKED, s);
((DefaultLookAndFeel)UIManager.getInstance().getLookAndFeel()).setRadioButtonImages(radioFullImage, radioEmptyImage,
radioFullImage, radioEmptyImage);

Container container1 = BoxLayout.encloseY(new Label("Swipe the tab to see more"),
        new Label("You can put anything here"));
t.addTab("Tab1", container1);
t.addTab("Tab2", new SpanLabel("Some text directly in the tab"));

RadioButton firstTab = new RadioButton("");
RadioButton secondTab = new RadioButton("");
firstTab.setUIID("Container");
secondTab.setUIID("Container");
new ButtonGroup(firstTab, secondTab);
firstTab.setSelected(true);
Container tabsFlow = FlowLayout.encloseCenter(firstTab, secondTab);

hi.add(t);
hi.add(BorderLayout.south(tabsFlow));

t.addSelectionListener((i1, i2) -> {
    switch(i2) {
        case 0:
            if(!firstTab.isSelected()) {
                firstTab.setSelected(true);
            }
            break;
        case 1:
            if(!secondTab.isSelected()) {
                secondTab.setSelected(true);
            }
            break;
    }
});
```



Figure 155. Swipeable Tabs with an iOS carousel effect page 1

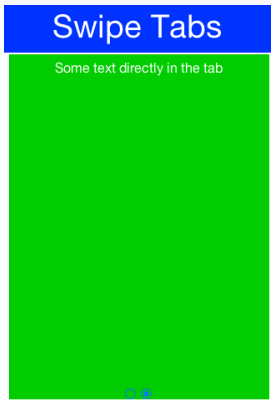


Figure 156. Swipeable Tabs with an iOS carousel effect page 2



Notice that you used `setRadioButtonImages` to explicitly set the radio button images to the look you want for the carousel.

7.23.1. Animated tab indicator

Modern Material 3 (`NavigationBar`) and iOS 26 tab bars animate a small underline between tabs when the user changes selection. `Tabs` ships this as an opt-in feature gated by the `tabsAnimatedIndicatorBool` theme constant; the iOS Modern and Android Material themes turn it on by default, so apps shipping against those themes get the effect for free.

To enable in a custom theme:

```
#Constants {
    tabsAnimatedIndicatorBool: true;
    tabsAnimatedIndicatorDurationInt: 200;    /* tween duration in ms */
    tabsAnimatedIndicatorThicknessMm: 1;      /* underline thickness  */
}

TabIndicator {
    /* The indicator picks up its color from this UIID's fg. If the
       UIID isn't defined or has fgColor == 0, the indicator falls
       back to the currently-selected tab's fgColor. */
    color: #007aff;
    background-color: transparent;
    padding: 0;
```

```
    margin: 0;
}
```

To toggle programmatically (for example, add it to a Tabs instance whose theme hasn't enabled it):

```
Tabs tabs = new Tabs();
tabs.setAnimatedIndicator(true);
```

The indicator animates its `x` / `width` from the previously selected tab's bounds to the new selection's bounds using a `Motion.createEaseInOutMotion` over the configured duration (200ms default, matching Material 3's spec). Rapid double-taps start the animation from the **current interpolated position** rather than from a stale baseline, so the indicator chains cleanly.

7.24. MediaManager & MediaPlayer



`MediaPlayer` is a **peer component**, understanding this is crucial if your application depends on such a component. You can learn about peer components and their issues here [https://www.codenameone.com/manual/advanced-topics.html#native-peer-components].

The MediaPlayer [https://www.codenameone.com/javadoc/com/codename1/components/MediaPlayer.html] allows you to control video playback. To use the `MediaPlayer` you need to first load the `Media` object from the MediaManager [https://www.codenameone.com/javadoc/com/codename1/media/MediaManager.html].

The `MediaManager` is the core class responsible for media interaction in Codename One.



You should also check out the Capture [https://www.codenameone.com/javadoc/com/codename1/capture/Capture.html] class for things that aren't covered by the `MediaManager`.

In the demo code below you use the gallery functionality to pick a video from the device's video gallery:

```
final Form hi = new Form("MediaPlayer", new BorderLayout());
hi.setToolbar(new Toolbar());
Style s = UIManager.getInstance().getComponentStyle("Title");
FontImage icon = FontImage.createMaterial(FontImage.MATERIAL_VIDEO_LIBRARY, s);
hi.getToolbar().addCommandToRightBar("", icon, (evt) -> {
    Display.getInstance().openGallery((e) -> {
        if(e != null && e.getSource() != null) {
            String file = (String)e.getSource();
            try {
                Media video = MediaManager.createMedia(file, true);
                hi.removeAll();
                hi.add(BorderLayout.CENTER, new MediaPlayer(video));
                hi.revalidate();
            } catch(IOException err) {
```

```
                Log.e(err);
            }
        }
    }, Display.GALLERY_VIDEO);
});
hi.show();
```



Figure 157. Video playback running on the simulator

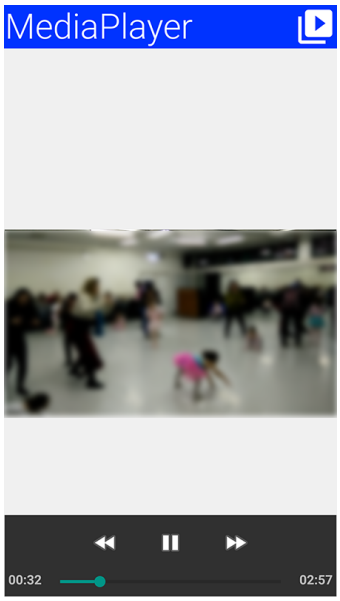


Figure 158. Video playback running on an Android device. Notice the native playback controls that appear when the video is tapped



Video playback in the simulator will work with JavaFX enabled. This is the default for Java 8 or newer so recommend using that.

7.25. ImageViewer

The ImageViewer [https://www.codenameone.com/javadoc/com/codename1/components/ImageViewer.html] allows you to inspect, zoom and pan into an image. It also allows swiping between images if you have a set of images (using an image list model).



The `ImageViewer` is a complex rich component designed for user interaction. If you want to display an image use Label [https://www.codenameone.com/javadoc/com/codename1/ui/Label.html] if you want the image to scale seamlessly use ScaleImageLabel [https://www.codenameone.com/javadoc/com/codename1/components/ScaleImageLabel.html].

You can use the `ImageViewer` as a tool to view a single image which allows you to zoom in/out to that image as such:



You can simulate pinch to zoom on the simulator by dragging the right button away from the top left corner to zoom in and towards the top left corner to zoom out. On Mac touchpads you can drag two fingers to achieve that.



Figure 159. ImageViewer as the demo loads with the image from the default icon



Figure 160. ImageViewer zoomed in

You can work with a list of images to produce a swiping effect for the image viewer where you can swipe from one image to the next and also zoom in/out on a specific image:

```
Form hi = new Form("ImageViewer", new BorderLayout());

Image red = Image.createImage(100, 100, 0xffff0000);
Image green = Image.createImage(100, 100, 0xff00ff00);
Image blue = Image.createImage(100, 100, 0xff0000ff);
Image gray = Image.createImage(100, 100, 0xffcccccc);

ImageViewer iv = new ImageViewer(red);
iv.setImageList(new DefaultListModel<>(red, green, blue, gray));
hi.add(BorderLayout.CENTER, iv);
```

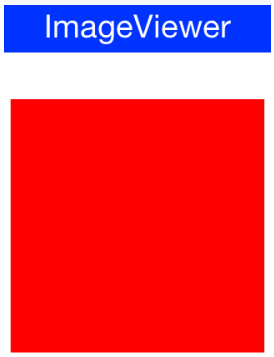


Figure 161. An ImageViewer with many elements is indistinguishable from a single ImageViewer except for swipe

Notice that you use a ListModel [https://www.codenameone.com/javadoc/com/codename1/ui/list/ListModel.html] to allow swiping between images.

`ImageViewer` also supports optional side arrows (material font icons) and an optional thumbnail strip for direct image navigation:

When arrows are enabled, tapping near the left/right edge will move to the previous/next image. When thumbnails are enabled, tapping a thumbnail jumps directly to that image.

All three options can be configured with theme constants (using the `ImageViewer` UIID prefix):

- `imageviewerNavigationArrowsBool=true|false`
- `imageviewerThumbnailsBool=true|false`
- `imageviewerThumbnailBarHeightMM=<floating point millimeters>`



`EncodedImage`s aren't always fully loaded and so when you swipe if the images are large you might see delays!

You can dynamically download images directly into the `ImageViewer` with a custom list model like this:

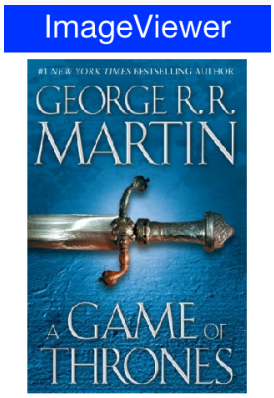


Figure 162. Dynamically fetching an image URL from the internet [4]

This fetches the images in the URL asynchronously and fires a data change event when the data arrives to automatically refresh the `ImageViewer` when that happens.

7.26. ScaleImageLabel & ScaleImageButton

ScaleImageLabel [https://www.codenameone.com/javadoc/com/codename1/components/ScaleImageLabel.html] & ScaleImageButton [https://www.codenameone.com/javadoc/com/codename1/components/ScaleImageButton.html] allow you to position an image that will grow/shrink to fit available space. In that sense they differ from Label & Button which keeps the image at the same size.



The default UIID of `ScaleImageLabel` is `Label`, but the default UIID of `ScaleImageButton` is `ScaleImageButton`. The reasoning for the difference is that the `Button` UIID includes a border and a lot of legacy.

You can use `ScaleImageLabel`/`ScaleImageButton` interchangeably. The major difference between these components is the buttons ability to handle click events/focus.

Here is a simple example that also shows the difference between the `scale to fill` and `scale to fit` modes:



Figure 163. ScaleImageLabel/Button, the top row includes scale to fit versions (the default) whereas the bottom row includes the scale to fill versions



When styling these components keep in mind that changing attributes such as background behavior might cause an issue with their functionality or might not work.

7.27. Toolbar

The Toolbar [https://www.codenameone.com/javadoc/com/codename1/ui/Toolbar.html] API provides deep customization of the title bar area with more flexibility for example: placing a TextField [https://www.codenameone.com/javadoc/com/codename1/ui/TextField.html] for search or buttons in arbitrary title area positions. The `Toolbar` API replicates some native functionality available on Android/iOS and integrates with features such as the side menu to provide fine grained control over the title area behavior.

The `Toolbar` needs to be installed into the `Form` in order for it to work. You can setup the Toolbar in one of these three ways:

1. `form.setToolbar(new Toolbar());` - allows you to activate the `Toolbar` to a specific `Form` and not for the entire application
2. `Toolbar.setGlobalToolbar(true);` - enables the `Toolbar` for all the forms in the app
3. Theme constant `globalToobarBool` - this is equivalent to `Toolbar.setGlobalToolbar(true);`

The basic functionality of the `Toolbar` includes the ability to add a command to the following 4 places:

- Left side of the title - `addCommandToLeftBar`
- Right side of the title - `addCommandToRightBar`
- Side menu bar (the drawer that opens when you click the icon on the top left or swipe the screen from left to right) - `addCommandToSideMenu`
- Overflow menu (the menu that opens when you tap the 3 vertical dots in the top right corner) - `addCommandToOverflowMenu`

The code below provides a brief overview of these options:



Figure 164. The Toolbar

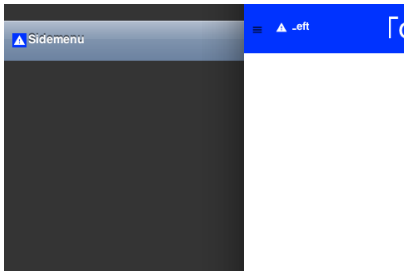


Figure 165. The default sidemenu of the Toolbar



Figure 166. The overflow menu of the Toolbar

You can set a title with a `String` but if you would want the component to be a text field or a multi line label you can use `setTitleComponent(Component)` which allows you to install any component into the title area.



The code below demonstrates searching using custom code but the Toolbar also has built-in support for search covered in the next section

The customization of the title area allows for some pretty powerful UI effects for example: the code below allows searching dynamically within a set of entries and uses some neat tricks:

```
Toolbar.setGlobalToolbar(true);
Style s = UIManager.getInstance().getComponentStyle("Title");

Form hi = new Form("Toolbar", new BoxLayout(BoxLayout.Y_AXIS));
TextField searchField = new TextField("", "Toolbar Search"); ①
searchField.getHintLabel().setUIID("Title");
searchField.setUIID("Title");
```

```
searchField.getAllStyles().setAlignment(Component.LEFT);
hi.getToolbar().setTitleComponent(searchField);
FontImage searchIcon = FontImage.createMaterial(FontImage.MATERIAL_SEARCH, s);
searchField.addDataChangeListener((i1, i2) -> { ②
    String t = searchField.getText();
    if(t.length() < 1) {
        for(Component cmp : hi.getContentPane()) {
            cmp.setHidden(false);
            cmp.setVisible(true);
        }
    } else {
        t = t.toLowerCase();
        for(Component cmp : hi.getContentPane()) {
            String val = null;
            if(cmp instanceof Label) {
                val = ((Label)cmp).getText();
            } else {
                if(cmp instanceof TextArea) {
                    val = ((TextArea)cmp).getText();
                } else {
                    val = (String)cmp.getPropertyValue("text");
                }
            }
            boolean show = val != null && val.toLowerCase().indexOf(t) > -1;
            cmp.setHidden(!show);
            cmp.setVisible(show); ③
        }
    }
    hi.getContentPane().animateLayout(250);
});
hi.getToolbar().addCommandToRightBar("", searchIcon, (e) -> { ④
    searchField.startEditingAsync();
});

hi.add("A Game of Thrones").
        add("A Clash Of Kings").
        add("A Storm Of Swords").
        add("A Feast For Crows").
        add("A Dance With Dragons").
        add("The Winds of Winter").
        add("A Dream of Spring");
hi.show();
```

① You use a `TextField` the whole time and style it to make it (and its hint) look like a regular title. An alternative way is to replace the title component dynamically.

② You use the `DataChangeListener` to update the search results as you type them.

③ Hidden & Visible use the opposite flag values to say similar things (for example: when hidden is set to false you would want to set visible to true).
Visible indicates whether a component can be seen. It will still occupy the physical space on the

screen even when it isn't visible. Hidden will remove the space occupied by the component from the screen, but some code might still try to paint it., visible is redundant but you use it with hidden for good measure.

④ The search button is totally unnecessary here. You can click the `TextField`!
For example, that isn't intuitive to most users so a button is added to start editing.



Figure 167. Search field within the toolbar



Figure 168. Search field after typing a couple of letters

7.27.1. Search mode

While you can implement search manually, using the built-in search offers a simpler and uniform UI.

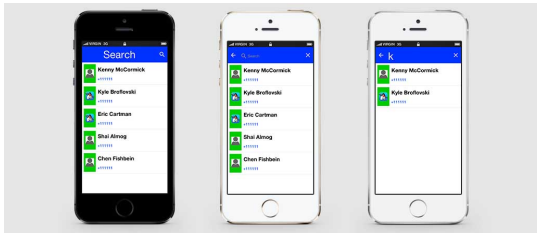


Figure 169. built-in toolbar search functionality

You can customize the appearance of the search bar by using the UIID's: `ToolbarSearch`, `TextFieldSearch` & `TextHintSearch`.

In the sample below you fetch all the contacts from the device and enable search through them, notice it expects and image called `duke.png` which is the default Codename One icon renamed and placed in the src folder:

7.27.2. South Component

A common feature in side menu bar is the ability to add a component to the "south" part of the side menu.

Notice that this feature works with the on-top and permanent versions of the side menu and not with the legacy versions:

This places the component below the side menu bar. Notice that this component controls its entire UIID & is separate from the `SideNavigationPanel` UIID so if you set that component you might want to place it within a container that has the `SideNavigationPanel` UIID so it will blend with the rest of the UI.

7.27.3. Title animations

Modern UI's often animate the title upon scrolling to balance the highly functional smaller title advantage with the gorgeous large image based title. This is pretty easy to do with the Toolbar API through the Title animation API.

The code below shows off an attractive title based on a book by GRRM on top of text [5] that's scrollable. As the text is scrolled the title fades out:

```
Toolbar.setGlobalToolbar(true);

Form hi = new Form("Toolbar", new BoxLayout(BoxLayout.Y_AXIS));
EncodedImage placeholder = EncodedImage.createFromImage(Image.createImage(hi.getWidth(), hi.getWidth() / 5, 0xffff0000),
true);
URLImage background = URLImage.createToStorage(placeholder, "400px-AGameOfThrones.jpg",
        "http://awoiaf.westeros.org/images/thumb/9/93/AGameOfThrones.jpg/400px-AGameOfThrones.jpg");
background.fetch();
Style stitle = hi.getToolbar().getTitleComponent().getUnselectedStyle();
stitle.setBgImage(background);
stitle.setBackgroundType(Style.BACKGROUND_IMAGE_SCALED_FILL);
stitle.setPaddingUnit(Style.UNIT_TYPE_DIPS, Style.UNIT_TYPE_DIPS, Style.UNIT_TYPE_DIPS, Style.UNIT_TYPE_DIPS);
stitle.setPaddingTop(15);
SpanButton credit = new SpanButton("This excerpt is from A Wiki Of Ice And Fire. Please check it out by clicking here!");
credit.addActionListener((e) -> Display.getInstance().execute("http://awoiaf.westeros.org/index.php/A_Game_of_Thrones"));
hi.add(new SpanLabel("A Game of Thrones is the first of seven planned novels in A Song of Ice and Fire, an epic fantasy
series by American author George R. R. Martin. It was first published on 6 August 1996. The novel was nominated for the
1998 Nebula Award and the 1997 World Fantasy Award,[1] and won the 1997 Locus Award.[2] The novella Blood of the Dragon,
comprising the Daenerys Targaryen chapters from the novel, won the 1997 Hugo Award for Best Novella. ")).
        add(new Label("Plot introduction", "Heading")).
        add(new SpanLabel("A Game of Thrones is set in the Seven Kingdoms of Westeros, a land reminiscent of Medieval
Europe. In Westeros the seasons last for years, sometimes decades, at a time.\n\n" +
            "Fifteen years prior to the novel, the Seven Kingdoms were torn apart by a civil war, known alternately as
\"Robert's Rebellion\" and the \"War of the Usurper.\" Prince Rhaegar Targaryen kidnapped Lyanna Stark, arousing the ire
of her family and of her betrothed, Lord Robert Baratheon (the war's titular rebel). The Mad King, Aerys II Targaryen,
had Lyanna's father and eldest brother executed when they demanded her safe return. Her second brother, Eddard, joined
his boyhood friend Robert Baratheon and Jon Arryn, with whom they had been fostered as children, in declaring war against
the ruling Targaryen dynasty, securing the allegiances of House Tully and House Arryn through a network of dynastic
marriages (Lord Eddard to Catelyn Tully and Lord Arryn to Lysa Tully). The powerful House Tyrell continued to support the
King, but House Lannister and House Martell both stalled due to insults against their houses by the Targaryens. The civil
war climaxed with the Battle of the Trident, when Prince Rhaegar was killed in battle by Robert Baratheon. The Lannisters
finally agreed to support King Aerys, but then brutallynull ")).
        add(credit);

ComponentAnimation title = hi.getToolbar().getTitleComponent().createStyleAnimation("Title", 200);
hi.getAnimationManager().onTitleScrollAnimation(title);
```

```
hi.show();
```

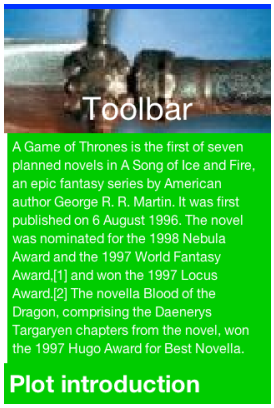


Figure 170. The Toolbar starts with the large URLImage fetched from the web



Figure 171. As you scroll down the image fades and the title shrinks in size returning to the default UIID look



Figure 172. As scrolling continues the title reaches standard size

Most the code above creates the "look" of the application. The key piece of code above is this:

```
ComponentAnimation title = hi.getToolbar().getTitleComponent().createStyleAnimation("Title", 200);
hi.getAnimationManager().onTitleScrollAnimation(title);
```

In the first line you create a style animation that will translate the style from the current settings to the destination UIID (the first argument) within 200 pixels of scrolling. You then bind this animation to the title scrolling animation event.

7.28. BrowserComponent & WebBrowser



`BrowserComponent` is a **peer component**, understanding this is crucial if your application depends on such a component. You can learn about peer components and their issues here [https://www.codenameone.com/manual/advanced-topics.html#native-peer-components].

The WebBrowser [https://www.codenameone.com/javadoc/com/codename1/components/WebBrowser.html] component shows the native device web browser when supported by the device and the HTMLComponent [https://www.codenameone.com/javadoc/com/codename1/ui/html/HTMLComponent.html] when the web browser isn't supported on the given device. If you intend to target smartphones you should use the BrowserComponent [https://www.codenameone.com/javadoc/com/codename1/ui/BrowserComponent.html] directly instead of the `WebBrowser`.

The `BrowserComponent` can point at an arbitrary URL to load it:

```
Form hi = new Form("Browser", new BorderLayout());
BrowserComponent browser = new BrowserComponent();
browser.setURL("https://www.codenameone.com/");
hi.add(BorderLayout.CENTER, browser);
```



Figure 173. Browser Component showing the Codename One website on the simulator



The scrollbars appear in the simulator, device versions of the browser component act differently and support touch scrolling.



A `BrowserComponent` should be in the center of a BorderLayout. Otherwise its preferred size might be zero before the HTML finishes loading/layout in the native layer and layout might be wrong as a result.

You can use `WebBrowser` and `BrowserComponent` interchangeably for most basic usage. For example, if you need access to JavaScript or native browser functionality then there is no use in going through

the WebBrowser abstraction.

The BrowserComponent has full support for executing local web pages from within the jar. The basic support uses the jar:/// URL as such:

```
BrowserComponent wb = new BrowserComponent();
wb.setURL("jar:///Page.html");
```



On Android a native indicator might show up when the web page is loading. This can be disabled using the Display.getInstance().setProperty("WebLoadingHidden", "true"); call. You need to invoke this once.

7.28.1. Troubleshooting "Failed to create CEF browser" on Linux

In the Simulator, BrowserComponent is rendered by the bundled CEF [https://bitbucket.org/chromiumembedded/java-cef] (Chromium Embedded Framework) port. On Linux, BrowserComponent.isNativeBrowserSupported() may return true and yet creating the component throws RuntimeException: Failed to create CEF browser, with a root cause similar to:

```
java.lang.UnsatisfiedLinkError: .../codenameone-cef-*-linux64.zip-extracted/lib/linux64/libjcef.so:
    .../lib/linux64/libjawt.so: version `SUNWprivate_1.1' not found
    (required by .../lib/linux64/libjcef.so)
```

The native CEF library (libjcef.so) is loaded with a sibling libjawt.so shipped inside the CEF archive. That copy of libjawt.so expects symbols (SUNWprivate_1.1) provided by the JDK's own libjawt.so and the matching libjvm.so. When the JVM running the Simulator doesn't expose those symbols at load time — typically because the dynamic linker doesn't have the JDK's native library directories on its search path — the load fails before any browser is created. This is purely a Simulator/desktop concern; the same code runs unchanged on Android and iOS.

The fix is to launch the IDE (and therefore the Simulator JVM) with JAVA_HOME and LD_LIBRARY_PATH pointing at a supported JDK. For example, on Linux Mint / Ubuntu wrap the IDE startup script:

```
#!/bin/bash
export JAVA_HOME=/usr/lib/jvm/java-17-openjdk-amd64
export LD_LIBRARY_PATH=$JAVA_HOME/lib:$JAVA_HOME/lib/server${LD_LIBRARY_PATH:+:$LD_LIBRARY_PATH}
exec /path/to/your/ide "$@"
```

After relaunching the IDE through this wrapper, BrowserComponent instantiation succeeds in the Simulator. Any JDK from 11 through 25 that's supported for running the Simulator works — the important part is that LD_LIBRARY_PATH resolves to that JDK's lib and lib/server directories so the bundled libjcef.so finds a compatible libjawt/libjvm pair.



If BrowserComponent instantiation fails on Linux, log the environment with CN.getProperty("java.home", "<unset>"), CN.getProperty("JAVA_HOME", "<unset>") and CN.getProperty("LD_LIBRARY_PATH", "<unset>"). An <unset> value for

> `LD_LIBRARY_PATH` while the IDE was launched from a desktop shortcut is the typical fingerprint of this problem — desktop launchers don't source your shell profile.



This issue doesn't affect packaged native applications, only the desktop Simulator running on Linux.

7.28.2. BrowserComponent hierarchy

When Codename One packages applications into native apps it hides a lot of details to make the process simpler. One of the things hidden is the fact that you aren't dealing with a JAR anymore, so `getResource`/`getResourceAsStream` are problematic... Both of these APIs support hierarchies and a concept of package relativity both of which might not be supported on all OSes.

Codename One has its own getResourceAsSteam in the Display [https://www.codenameone.com/javadoc/com/codename1/ui/Display.html] class and that works fine, but it requires that all files be in the src root directory.



That's why recommend that you place files inside res files. A resource file allows you to add arbitrary data files and you can have as many resource files as you need.

For web developers this isn't enough since hierarchies are used often to represent the various dependencies, this means that many links & references are relative. To work with such hierarchies place all your resources in a hierarchy under the HTML package in the project source directory (`src/html`). The build server will `tar` the entire content of that package and add an `html.tar` file into the native package. This `tar` is seamlessly extracted on the device when you actually need the resources and with new application versions (not on every launch). Assuming the resources are under the HTML root package they can be displayed with code like this:

```
try {
    browserComponent.setURLHierarchy("/htmlFile.html");
} catch(IOException err) {
    /* omitted */
}
```

Notice that the path is relative to the HTML directory and starts with / but inside the HTML files you should use relative (not absolute) paths.

Also notice that an `IOException` can be thrown due to the process of untarring. Its unlikely to happen but is entirely possible.

7.28.3. NavigationCallback

At the core of the `BrowserComponent` you have the BrowserNavigationCallback [https://www.codenameone.com/javadoc/com/codename1/ui/events/BrowserNavigationCallback.html]. It might not seem like the most important interface within the browser but it's the "glue" that allows the JavaScript code to communicate back into the Java layer.

You can bind a BrowserNavigationCallback by invoking setBrowserNavigationCallback on the BrowserComponent. At that point with every navigation within the browser the callback will get invoked.



The shouldNavigate method from the BrowserNavigationCallback is invoked in a native thread and **NOT ON THE EDT**!
it's crucial that this method returns and that it won't do any changes on the UI.

The shouldNavigate indicates to the native code whether navigation should proceed or not. For example: if a user clicks a specific link you might choose to do something in the Java code so you can return false and block the navigation. You can invoke https://www.codenameone.com/javadoc/com/codename1/ui/Display.html#callSerially java.lang.Runnable-[callSerially] to do the actual task in the Java side:

```
Form hi = new Form("BrowserComponent", new BorderLayout());
BrowserComponent bc = new BrowserComponent();
bc.setPage( "<html lang=\"en\">\n" +
            "    <head>\n" +
            "        <meta charset=\"utf-8\">\n" +
            "        <script>\n" +
            "          function  fnc(message) {\n" +
            "         document.write(message);\n" +
            "            };\n" +
            "        </script>\n" +
            "    </head>\n" +
            "    <body >\n" +
            "        <p><a href=\"http://click\">Demo</a></p>\n" +
            "    </body>\n" +
            "</html>", null);
hi.add(BorderLayout.CENTER, bc);
bc.setBrowserNavigationCallback((url) -> {
    if(url.startsWith("http://click")) {
        Display.getInstance().callSerially(() -> bc.execute("fnc('<p>You clicked!</p>')"));
        return false;
    }
    return true;
});
```



Figure 174. Before the link is clicked for the "shouldNavigate" call



Figure 175. After the link is clicked for the "shouldNavigate" call



The JavaScript Bridge is implemented on top of the BrowserNavigationCallback.

7.28.4. JavaScript



The JavaScript bridge is sometimes confused with the JavaScript Port. The JavaScript bridge allows you to communicate with JavaScript from Java (and vice versa). The JavaScript port allows you to compile the Codename One application into a JavaScript application that runs in a standard web browser without code changes (think GWT without source changes and with thread support).+ You discuss the JavaScript port further later in the guide.

Use the `BrowserComponent` API to interact with JavaScript. It replaces the deprecated com.codename1.JavaScript package [https://www.codenameone.com/javadoc/com/codename1/JavaScript/package-summary.html].

What was wrong with the old API

The old API provided a synchronous wrapper around an inherently asynchronous process, and made extensive use of `invokeAndBlock()` underneath the covers. This resulted in a nice API with high-level abstractions that played with a synchronous programming model, but it came with a price-tag in performance, complexity, and predictability. Let's take a simple example, getting a reference to the "window" object:

This code looks harmless enough, but this is actually expensive. It issues a command to the `BrowserComponent`, and uses `invokeAndBlock()` to wait for the command to go through and send back a response. `invokeAndBlock()` is a magical tool that allows you to "block" without blocking the EDT, but it has its costs, and shouldn't be overused. Most of the Codename One APIs that use `invokeAndBlock()` show this in their name. For example: `Component.animateLayoutAndWait()`. This provides you the expectation that this call could take some time, and helps to alert you to the underlying cost.

The problem with the `ctx.get("window")` call is that it looks the same as a call to `Map.get(key)`. There's no sign that this call is expensive and could take time. One call like this probably isn't a big deal, but it doesn't take long before you have dozens or even hundreds of calls like this littered throughout your codebase, and they can be hard to pick out.

The new API

The new API fully embraces the asynchronous nature of JavaScript. It uses callbacks instead of return values, and provides convenience wrappers with the appropriate "AndWait()" naming convention to allow for synchronous usage. Let's look at a simple example:



In all the sample code below, you can assume that variables named `bc` represent an instance of BrowserComponent [https://www.codenameone.com/javadoc/com/codename1/ui/BrowserComponent.html]:

This code should output "The result was 7" to the console. It's fully asynchronous, so you can include this code anywhere without worrying about it "bogging down" your code. The full signature of this form of the execute() [https://www.codenameone.com/javadoc/com/codename1/ui/BrowserComponent.html#execute-java.lang.String-com.codename1.util.SuccessCallback-] method is:

The first parameter is a JavaScript expression. This JavaScript **MUST** call either `callback.onSuccess(result)` or `callback.onError(message, errCode)` at some point in order for your callback to be called.

The second parameter is your callback that's executed from the JavaScript side, when `callback.onSuccess(res)` is called. The callback takes a single parameter of type JSRef [https://www.codenameone.com/javadoc/com/codename1/ui/BrowserComponent.JSRef.html] which is a generic wrapper around a JavaScript variable. JSRef has accessors to retrieve the value as some primitive types. For example: `getBoolean()`, `getDouble()`, `getInt()`, `toString()`, and it provides some introspection through the `getType()` method.



It's worth noting that the callback method can take a single parameter. If you need to pass many parameters, you may consider including them in a single string which you parse in your callback.

Synchronous wrappers

As mentioned before, the new API also provides an `executeAndWait()` wrapper for `execute()` that will work synchronously. It, as its name suggests, uses `invokeAndBlock` under the hood so as not to block the EDT while it's waiting.

For example:

Prints `The result was 7`.



When using the `andWait()` variant, it's critical that your JavaScript calls your callback method at some point - otherwise it will block **indefinitely**. You provide variants of executeAndWait() that include a timeout in case you want to hedge against this possibility.

Multi-use callbacks

The callbacks you pass to `execute()` and `executeAndWait()` are single-use callbacks. You can't, for example, store the `callback` variable on the JavaScript side for later use (for example: to respond to a button click event). If you need a "multi-use" callback, you should use the `addJSCallback()` method instead. Its usage looks identical to `execute()`, the difference is that the callback will live on after its first use. For example: Consider the following code:

The above example, assumes that jQuery is loaded in the webpage that you're interacting with, and you're adding a click handler to a button with ID "somebutton." The click handler calls your callback.

If you run this example, the first time the button is clicked, you'll see "Button was clicked" printed to the console as expected. For example, the 2nd time, you'll get an exception. This is because the callback passed to `execute()` is single-use.

You need to change this code to use the `addJSCallback()` method as follows:

Now it will work no matter how many times the button is clicked.

Passing parameters to JavaScript

Often, the JavaScript expressions that you execute will include parameters from your Java code. Escaping these parameters is tricky at worst, and annoying at best. For example: If you're passing a string, you need to make sure that it escapes quotes and new lines or it will cause the JavaScript to have a syntax error. You provide variants of `execute()` and addJSCallback() [https://www.codenameone.com/javadoc/com/codename1/ui/BrowserComponent.html#addJSCallback-java.lang.String-com.codename1.util.SuccessCallback-] that allow you to pass your parameters and have them automatically escaped.

For example, suppose you want to pass a string with text to set in a textarea within the webpage. You can do something like:

The gist is that you embed placeholders in the JavaScript expression that are replaced by the corresponding entry in an array of parameters. The `${0}` placeholder is replaced by the first item in the parameters array, the ${1} placeholder is replaced by the 2nd, etc.

Proxy objects

The new API also includes a JSProxy [https://www.codenameone.com/javadoc/com/codename1/ui/BrowserComponent.JSProxy.html] class that encapsulates a JavaScript object simplify the getting and setting of properties on JavaScript objects - and the calling of their methods. It provides essentially three core methods, along with many variants of each to allow for async or synchronous usages, parameters, and timeouts.

For example: You might want to create a proxy for the window.location [https://developer.mozilla.org/en-You/docs/Web/API/Window/location] object so that you can access its properties more from Java:

Then you can retrieve its properties using the `get()` method:

Or synchronously:

You can also set its properties:

And call its methods:

Legacy JSObject support

This section describes the now deprecated `JSObject` approach. It's here for reference by developers working with older code. You suggest using the new API when starting a new project.

`BrowserComponent` can communicate with the HTML code using JavaScript calls. For example: you can create HTML like this:

```
Form hi = new Form("BrowserComponent", new BorderLayout());
BrowserComponent bc = new BrowserComponent();
bc.setPage( "<html lang=\"en\">\n" +
            "    <head>\n" +
            "        <meta charset=\"utf-8\">\n" +
            "        <script>\n" +
            "          function  fnc(message) {\n" +
```

```
            "         document.write(message);\n" +
            "            };\n" +
            "        </script>\n" +
            "    </head>\n" +
            "    <body >\n" +
            "        <p>Demo</p>\n" +
            "    </body>\n" +
            "</html>", null);
TextField tf = new TextField();
hi.add(BorderLayout.CENTER, bc).
        add(BorderLayout.SOUTH, tf);
bc.addWebEventListener("onLoad", (e) ->  bc.execute("fnc('<p>Hello World</p>')"));
tf.addActionListener((e) -> bc.execute("fnc('<p>" + tf.getText() +"</p>')"));
hi.show();
```

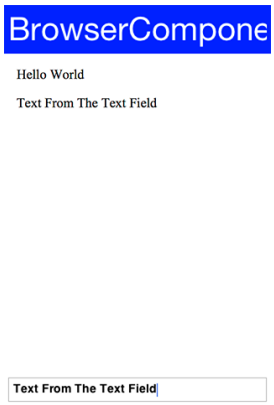


Figure 176. JavaScript code was invoked to append text into the browser image above



Notice that opening an alert in an embedded native browser might not work

You use the `execute` method above to execute custom JavaScript code. You also have an `executeAndReturnString` method that allows you to receive a response value from the JavaScript side.

Coupled with `shouldNavigate` you can effectively do everything which is what the JavaScript Bridge tries to do.

The JavaScript bridge

While it's possible to build everything on top of `execute` and `shouldNavigate`, both of these methods have their limits. That's why Codename One introduced the JavaScript package, it allows you to communicate with JavaScript using intuitive code/syntax.

The JavascriptContext [https://www.codenameone.com/javadoc/com/codename1/JavaScript/JavascriptContext.html] class lays the foundation by enabling you to call JavaScript code directly from Java. It provides automatic type conversion between Java and JavaScript types as follows:

Table 7. Java to JavaScript

Java Type	JavaScript Type
`String`	`String`

Java Type	JavaScript Type
`Double/Integer/Float/Long`	`Number`
`Boolean`	`Boolean`
`JSObject`	`Object`
`null`	`null`
`Other`	Not Allowed

Table 8. JavaScript to Java

JavaScript Type	Java Type
`String`	`String`
`Number`	`Double`
`Boolean`	`Boolean`
`Object`	`JSObject`
`Function`	`JSObject`
`Array`	`JSObject`
`null`	`null`
`undefined`	`null`



This conversion table is more verbose than necessary, since JavaScript functions and arrays are, in fact Objects themselves, so those rows are redundant. All JavaScript objects are converted to JSObject [https://www.codenameone.com/javadoc/com/codename1/JavaScript/JSObject.html].

You can access JavaScript variables from the context by using code like this:



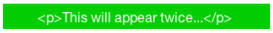


Figure 177. The contents were copied from the DOM and placed in the south position of the form

Notice that when you work with numeric values or anything related to the types mentioned above your code must be aware of the typing. For example: in this case the type is `Double` and not `String`:

You can also query the context for objects and change their value for example:

This code effectively navigates to the Codename One home page by fetching the DOM's window object and setting its `location` property to https://www.codenameone.com/ [https://www.codenameone.com/].

7.28.5. Cordova/PhoneGap integration

PhoneGap was one of the first web app packager tools in the market: a tool that's effectively a browser component within a native wrapper coupled with native access APIs, and Cordova is the open source extension of this popular project.

Codename One supports embedding PhoneGap/Cordova applications directly into Codename One applications. This is easy to do with the `BrowserComponent` and JavaScript integration. The main aspect that this integration requires is support for Cordova plugins & its JavaScript APIs.

The effort to integrate Cordova/PhoneGap support into Codename One is handled within an open-source GitHub project here [https://github.com/codenameone/CN1Cordova]. The chief benefits of picking Codename One rather than using Cordova directly are:

- Build Cloud
- Better Native Code Support
- Better Protection Of IP
- IDE Integration Java - JavaScript - HTML
- Easy, doesn't Require A Mac, Automates Certificates/Signing
- Migration To Java

This is discussed further in the original announcement [https://www.codenameone.com/blog/phonegap-cordova-compatibility-for-codename-one.html].

7.29. Text editors

`RichTextArea` and `CodeEditor` are introduced here as components, but editing is broad enough to have its own chapter. See Chapter 8, *Rich Text and Code Editing* for rich HTML, syntax highlighting, completion, diagnostics, touch and desktop selection, scrolling, bidirectional text, input methods, backend selection, and the port-level `TextInputClient` contract.

7.30. AutoCompleteTextField

The AutoCompleteTextField [https://www.codenameone.com/javadoc/com/codename1/ui/AutoCompleteTextField.html] allows you to write text into a text field and select a completion entry from the list in a similar way to a search engine.

This is easy to incorporate into your code, replace your usage of TextField [https://www.codenameone.com/javadoc/com/codename1/ui/TextField.html] with `AutoCompleteTextField` and define the data that the autocomplete should work from. A default implementation accepts a `String` array or a ListModel [https://www.codenameone.com/javadoc/com/codename1/ui/list/ListModel.html] for completion strings, this can work well for a "small" set of thousands (or tens of thousands) of entries.

For example: This is a trivial use case that can work well for smaller sample sizes:

```
Form hi = new Form("Auto Complete", new BoxLayout(BoxLayout.Y_AXIS));
AutoCompleteTextField ac = new AutoCompleteTextField("Short", "Shock", "Sholder", "Shrek");
ac.setMinimumElementsShownInPopup(5);
hi.add(ac);
```

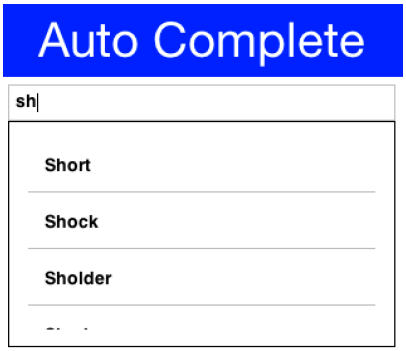


Figure 178. Autocomplete Text Field

For example, if you wish to query a database or a web service you will need to derive the class and perform more advanced filtering by overriding the `filter` method:



Figure 179. Autocomplete Text Field with a webservice

7.30.1. Using images in AutoCompleteTextField

One question that comes up frequently is "How do you customize the results of the auto complete field"?

This sounds difficult to most people as you can work with Strings so how do you represent more data or format the date correctly?

The answer is actually pretty simple, you still need to work with Strings because autocomplete is fundamentally a text field. For example, that doesn't preclude your custom renderer from fetching data that might be placed in a different location and associated with the result.

The following source code presents an autocomplete text field with images in the completion popup and two lines for every entry:

```
final String[] characters = { "Tyrion Lannister", "Jaime Lannister", "Cersei Lannister", "Daenerys Targaryen",
    "Jon Snow", "Petyr Baelish", "Jorah Mormont", "Sansa Stark", "Arya Stark", "Theon Greyjoy"
    // snipped the rest for clarity
};

Form current = new Form("AutoComplete", BoxLayout.y());

AutoCompleteTextField ac = new AutoCompleteTextField(characters);

final int size = Display.getInstance().convertToPixels(7);
final EncodedImage placeholder = EncodedImage.createFromImage(Image.createImage(size, size, 0xffcccccc), true);
```

```
final String[] actors = { "Peter Dinklage", "Nikolaj Coster-Waldau", "Lena Headey"}; ①
final Image[] pictures = {
    URLImage.createToStorage(placeholder, "tyrion","http://i.lv3.hbo.com/assets/images/series/game-of-
thrones/character/s5/tyrion-lannister-512x512.jpg"),
    URLImage.createToStorage(placeholder, "jaime","http://i.lv3.hbo.com/assets/images/series/game-of-
thrones/character/s5/jamie-lannister-512x512.jpg"),
    URLImage.createToStorage(placeholder, "cersei","http://i.lv3.hbo.com/assets/images/series/game-of-
thrones/character/s5/cersei-lannister-512x512.jpg")
};

ac.setCompletionRenderer(new ListCellRenderer() {
    private final Label focus = new Label(); ②
    private final Label line1 = new Label(characters[0]);
    private final Label line2 = new Label(actors[0]);
    private final Label icon = new Label(pictures[0]);
    private final Container selection = BorderLayout.center(
            BoxLayout.encloseY(line1, line2)).add(BorderLayout.EAST, icon);

    @Override
    public Component getListCellRendererComponent(com.codename1.ui.List list, Object value, int index, boolean
isSelected) {
        for(int iter = 0 ; iter < characters.length ; iter++) {
            if(characters[iter].equals(value)) {
                line1.setText(characters[iter]);
                if(actors.length > iter) {
                    line2.setText(actors[iter]);
                    icon.setIcon(pictures[iter]);
                } else {
                    line2.setText(""); ③
                    icon.setIcon(placeholder);
                }
                break;
            }
        }
        return selection;
    }

    @Override
    public Component getListFocusComponent(com.codename1.ui.List list) {
        return focus;
    }
});
current.add(ac);

current.show();
```

① you have duplicate arrays that are partial for clarity. This is a separate list of data element but you can fetch the more data from anywhere

② You create the renderer UI instantly in the fields with the helper methods for wrapping elements which is pretty cool & terse

③ In a renderer it's important to always set the value if you don't have a value in place

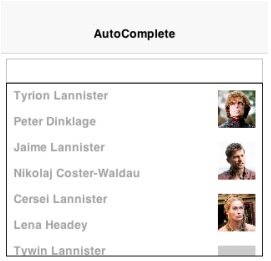


Figure 180. Auto complete with images

7.31. Picker

Picker [https://www.codenameone.com/javadoc/com/codename1/ui/spinner/Picker.html] occupies the limbo between native widget and lightweight widget. Picker is more like `TextField`/`TextArea` in the sense that it's a Codename One widget that calls the native code during editing.

The reasoning for this is the highly native UX and functionality related to this widget type which should be obvious from the screenshots below.

At this time there are 4 types of pickers:

- Time
- Date & Time
- Date
- Strings

If a platform doesn't support native pickers an internal fallback implementation is used. This is the implementation you always use in the simulator so assume different behavior when building for the device.



While Android supports Date, Time native pickers it doesn't support the Date & Time native picker UX and will fall back in that case.

The sample below includes al picker types:

```
Form hi = new Form("Picker", new BoxLayout(BoxLayout.Y_AXIS));
Picker datePicker = new Picker();
datePicker.setType(Display.PICKER_TYPE_DATE);
Picker dateTimePicker = new Picker();
dateTimePicker.setType(Display.PICKER_TYPE_DATE_AND_TIME);
Picker timePicker = new Picker();
timePicker.setType(Display.PICKER_TYPE_TIME);
Picker stringPicker = new Picker();
stringPicker.setType(Display.PICKER_TYPE_STRINGS);
Picker durationPicker = new Picker();
durationPicker.setType(Display.PICKER_TYPE_DURATION);
Picker minuteDurationPicker = new Picker();
minuteDurationPicker.setType(Display.PICKER_TYPE_DURATION_MINUTES);
Picker hourDurationPicker = new Picker();
hourDurationPicker.setType(Display.PICKER_TYPE_DURATION_HOURS);

datePicker.setDate(new Date());
dateTimePicker.setDate(new Date());
timePicker.setTime(10 * 60); // 10:00AM = Minutes since midnight
stringPicker.setStrings("A Game of Thrones", "A Clash Of Kings", "A Storm Of Swords", "A Feast For Crows",
        "A Dance With Dragons", "The Winds of Winter", "A Dream of Spring");
stringPicker.setSelectedString("A Game of Thrones");

hi.add(datePicker).add(dateTimePicker).add(timePicker)
  .add(stringPicker).add(durationPicker)
  .add(minuteDurationPicker).add(hourDurationPicker);
hi.show();
```

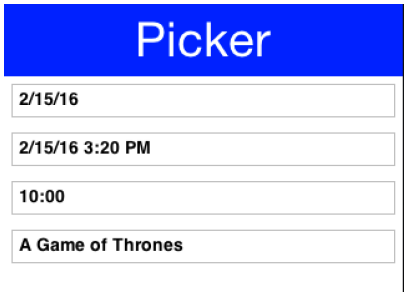


Figure 181. The various picker components

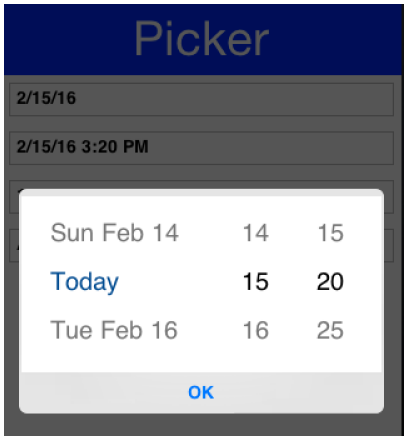


Figure 182. The date & time picker on the simulator

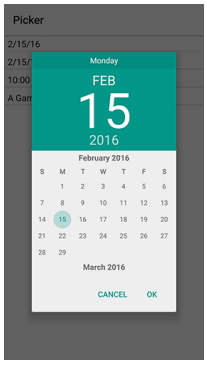


Figure 183. The date picker component on the Android device

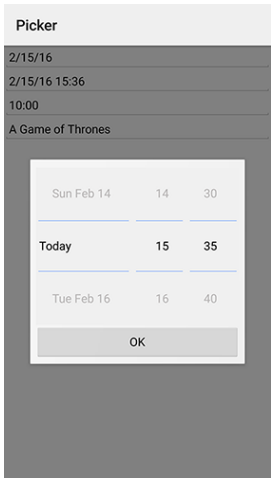


Figure 184. Date & time picker on Android. Notice it didn't use a built-in widget since there is none

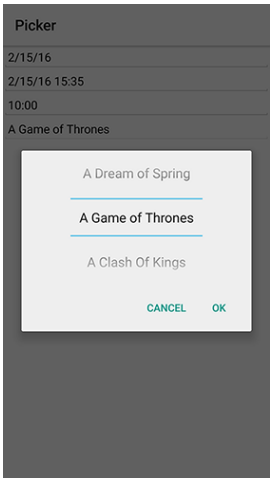


Figure 185. String picker on the native Android device

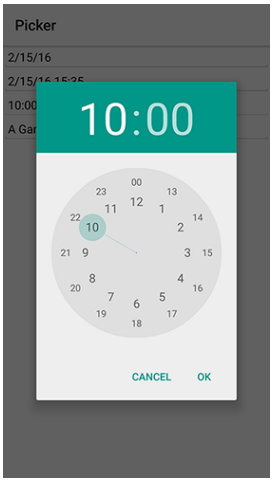


Figure 186. Time picker on the Android device

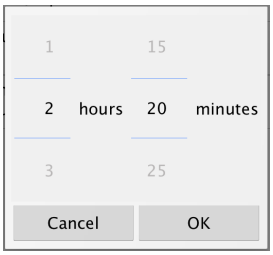


Figure 187. Duration picker on Android device

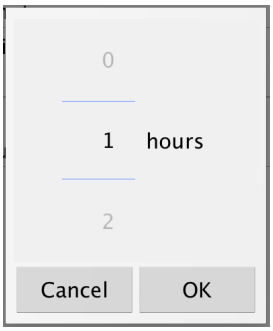


Figure 188. Hours Duration picker on Android device

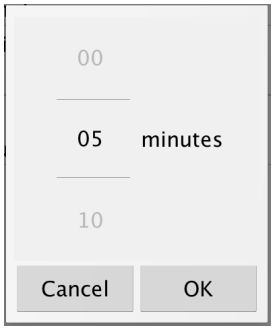


Figure 189. Minutes Duration picker on Android device

The text displayed by the picker on selection is generated automatically by the `updateValue()` method. You can override it to display a custom formatted value and call `setText(String)` with the correct display string.

A common use case is to format date values based on a specific appearance and `Picker` has built-in support for a custom display formatter. Just use the `setFormatter(SimpleDateFormat)` method and set the appearance for the field.

When using lightweight picker mode (`setUseLightweightPopup(true)`), you can add custom quick-action buttons to the popup. This is useful for actions like setting the date to "Today" or "+7 Days" without scrolling the wheels manually:

Button placement options are:

- `Picker.LightweightPopupButtonPlacement.BETWEEN_CANCEL_AND_DONE` (default)
- `Picker.LightweightPopupButtonPlacement.ABOVE_SPINNER`
- `Picker.LightweightPopupButtonPlacement.BELOW_SPINNER`

7.32. SwipeableContainer

The SwipeableContainer [https://www.codenameone.com/javadoc/com/codename1/ui/SwipeableContainer.html] allows you to place a component such as a MultiButton [https://www.codenameone.com/javadoc/com/codename1/components/MultiButton.html] on top of more "options" that can be exposed by swiping the component to the side.

This swipe gesture is commonly used in touch interfaces to expose features such as delete, edit etc. It’s trivial to use this component by determining the components placed on top and bottom (the revealed component):

You can combine some demos above including the Slider stars demo to rank GRRM’s books in an interactive way:

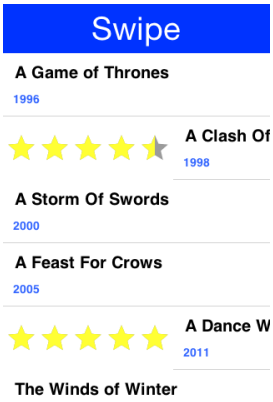


Figure 190. SwipableContainer showing a common use case of ranking on swipe

7.33. EmbeddedContainer

EmbeddedContainer [https://www.codenameone.com/javadoc/com/codename1/ui/util/EmbeddedContainer.html] solves a problem that exists within the GUI builder and the class makes no sense outside of the context of the GUI builder. The necessity for `EmbeddedContainer` came about due to iPhone inspired designs that relied on tabs (iPhone style tabs at the bottom of the screen) where different features of the application are within a different tab.

This didn't mesh well with the GUI builder navigation logic and so it needed a rethink. The aim was to reuse GUI as much as possible while still enjoying the advantage of navigation being managed for you.

Android does this with Activities and the iPhone itself has a view controller, both approaches are problematic for Codename One. The problem is that you have what's effectively two incompatible hierarchies to mix and match.

The Component/Container hierarchy is powerful enough to represent such a UI but you needed a "marker" to show to the UIBuilder [https://www.codenameone.com/javadoc/com/codename1/ui/util/UIBuilder.html] where a "root" component exists so navigation occurs within the given "root." Here `EmbeddedContainer` comes into play, its a simple container that can contain another GUI from the GUI builder. Nothing else. You can place it in any form of UI and effectively have the UI change appropriately and navigation would default to "sensible values."

Navigation replaces the content of the embedded container; it finds the embedded container based on the component that broadcast the event. If you want to navigate manually use the showContainer() method which accepts a component, you can give any component that's under the `EmbeddedContainer` you want to replace and Codename One will be smart enough to replace that component.

The nice part about using the `EmbeddedContainer` is that the resulting UI can be refactored to provide a more traditional form based UI without duplicating effort and can be adapted to a more tablet oriented UI (with a sidebar) again without much effort.

7.34. MapComponent



The MapComponent uses a somewhat outdated tiling API which isn't as rich as

> modern native maps. Recommend using the GoogleMap's Native cn1lib [https://github.com/codenameone/codenameone-google-maps/] to integrate native mapping functionality into the Codename One app.

The MapComponent [https://www.codenameone.com/javadoc/com/codename1/maps/MapComponent.html] uses the OpenStreetMap webservice by default to display a navigatable map.

The code was contributed by Roman Kamyk and was originally used for a LWUIT application.



Figure 191. Map Component

The screenshot above was produced using the following code:

The example below shows how to integrate the MapComponent [https://www.codenameone.com/javadoc/com/codename1/maps/MapComponent.html] with the Google Location [https://www.codenameone.com/javadoc/com/codename1/location/Location.html] API. Make sure to get your secret API key from the Google Location [https://www.codenameone.com/javadoc/com/codename1/location/Location.html] data API at: https://developers.google.com/maps/documentation/places/



Figure 192. MapComponent with Google Location API

7.35. Chart Component

The `charts` package enables Codename One developers to add charts and visualizations to their apps without having to include external libraries or embedding web views. You also wanted to harness the new features in the graphics pipeline to maximize performance.

7.35.1. Device support

Since the charts package makes use of 2D transformations and shapes, it requires some graphics features that aren't yet available on all platforms. The following platforms are supported:

1. Simulator
2. Android
3. iOS

7.35.2. Features

1. **Built-in support for many common types of charts** including bar charts, line charts, stacked charts, scatter charts, pie charts and more.
2. **Pinch Zoom** - The ChartComponent [https://www.codenameone.com/javadoc/com/codename1/charts/ChartComponent.html] class includes optional pinch zoom support.
3. **Panning Support** - The ChartComponent [https://www.codenameone.com/javadoc/com/codename1/charts/ChartComponent.html] class includes optional support for panning.

7.35.3. Chart types

The `com.codename1.charts` package includes models and renderers for many different types of charts. It's also extensible so that you can add your own chart types if required. The following screenshots show a small sampling of the types of charts that can be created.

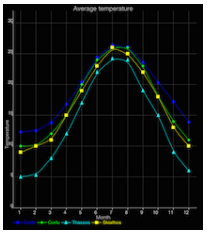


Figure 193. Line Charts

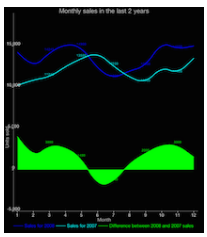


Figure 194. Cubic Line Charts

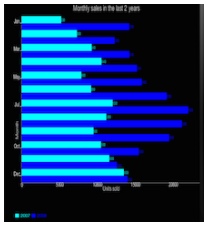


Figure 195. Bar Charts

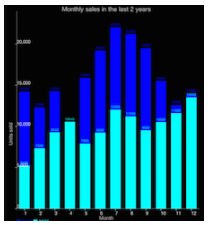


Figure 196. Stacked Bar Charts

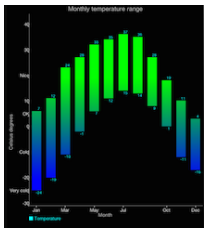


Figure 197. Range Bar Charts

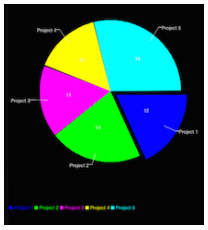


Figure 198. Pie Charts

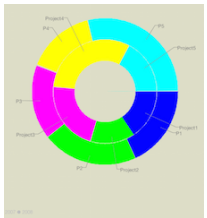


Figure 199. Doughnut Charts

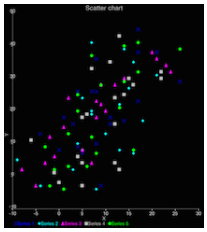


Figure 200. Scatter Charts

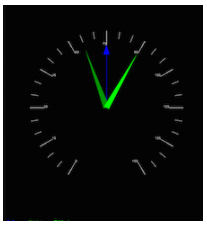


Figure 201. Dial Charts

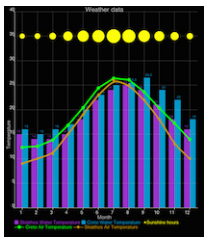


Figure 202. Combined Charts

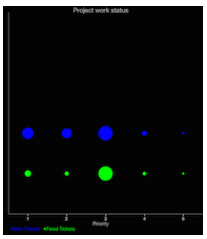


Figure 203. Bubble Charts

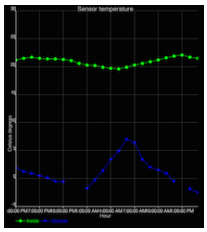


Figure 204. Time Charts



The above screenshots were taken from the ChartsDemo app [https://github.com/codenameone/codenameone-demos/tree/master/ChartsDemo]. You can start playing with this app by checking it out from your git repository.

7.35.4. How to create a chart

Adding a chart to your app involves four steps:

1. **Build the model**. You can construct a model (aka data set) for the chart using one of the existing model classes in the `com.codename1.charts.models` package. Essentially, this is where you add the data that you want to display.
2. **Set up a renderer**. You can create a renderer for your chart using one of the existing renderer classes in the `com.codename1.charts.renderers` package. The renderer allows you to specify how the chart should look. For example: the colors, fonts, styles, to use.
3. **Create the Chart View**. Use one of the existing *view* classes in the `com.codename1.charts.views` package.
4. **Create a ChartComponent [https://www.codenameone.com/javadoc/com/codename1/charts/ChartComponent.html]**. To add your chart to the UI, you need to wrap it in a ChartComponent [https://www.codenameone.com/javadoc/com/codename1/charts/ChartComponent.html] object.

You can check out the ChartsDemo [https://github.com/codenameone/codenameone-demos/tree/master/ChartsDemo] app for specific examples, but here is a high-level view of some code that creates a Pie Chart:

7.36. Calendar

The Calendar [https://www.codenameone.com/javadoc/com/codename1/ui/Calendar.html] class allows you to display a traditional calendar picker and optionally highlight days in various ways.



You recommend developers use the Picker UI rather than use the calendar to pick a date. It looks better on the devices.

Simple usage of the `Calendar` class looks something like this:



Figure 205. The Calendar component

7.37. ToastBar

The ToastBar [https://www.codenameone.com/javadoc/com/codename1/components/ToastBar.html] class allows you to display none-obtrusive status messages to the user at the bottom of the screen. This is useful for such things as informing the user of a long-running task (like downloading a file in the

background), or popping up an error message that doesn't require a response from the user.

Simple usage of the `ToastBar` class looks something like this:

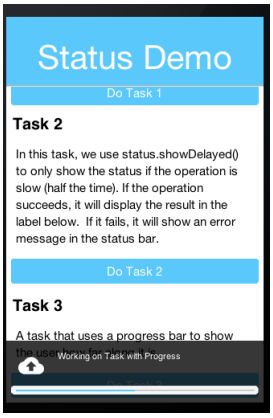


Figure 206. ToastBar with message

You can show a progress indicator in the ToastBar like this:

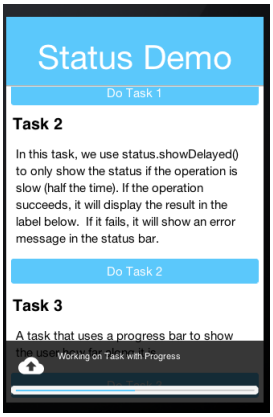


Figure 207. Status Message with Progress Bar

You can automatically clear a status message/progress after a timeout using the `setExpires` method as such:

You can also delay the showing of the status message using `showDelayed` as such:

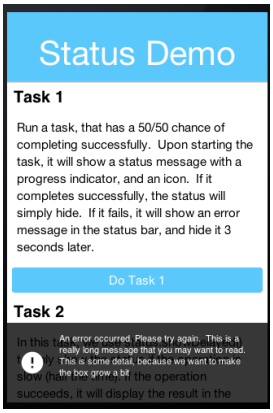


Figure 208. ToastBar with a multiline message

7.37.1. Actions in ToastBar

Probably the best usage example for actions in toast is in the gmail style undo. If you aren't a gmail

user then the gmail app essentially never prompts for confirmation!

It does whatever you ask and pops a "toast message" with an option to undo. If you clicked by mistake you have 3-4 seconds to take that back.

This simple example shows you how you can undo any addition to the UI in a similar way to gmail:

```
Form hi = new Form("Undo", BoxLayout.y());
Button add = new Button("Add");

add.addActionListener(e -> {
    Label l = new Label("Added this");
    hi.add(l);
    hi.revalidate();
    ToastBar.showMessage("Added, click here to undonull", FontImage.MATERIAL_UNDO,
            ee -> {
                l.remove();
                hi.revalidate();
            });
});
hi.add(add);
hi.show();
```

7.38. SignatureComponent

The SignatureComponent [https://www.codenameone.com/javadoc/com/codename1/components/SignatureComponent.html] provides a widget that allows users to draw their signature in the app.

Simple usage of the `SignatureComponent` class looks like:

```
Form hi = new Form("Signature Component");
hi.setLayout(new BoxLayout(BoxLayout.Y_AXIS));
hi.add("Enter Your Name:");
hi.add(new TextField());
hi.add("Signature:");
SignatureComponent sig = new SignatureComponent();
sig.addActionListener((evt)-> {
    System.out.println("The signature was changed");
    Image img = sig.getSignatureImage();
    // Now we can do whatever we want with the image of this signature.
});
hi.addComponent(sig);
hi.show();
```

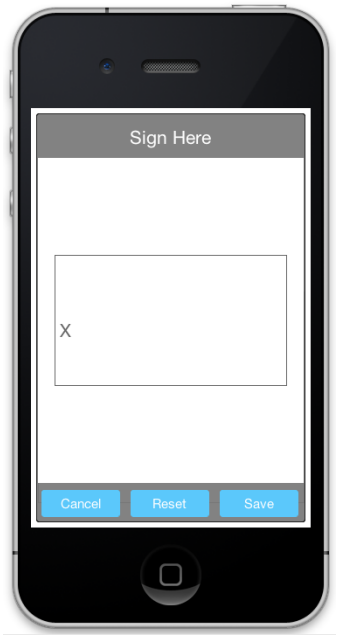


Figure 209. The signature Component

7.39. Accordion

The Accordion [https://www.codenameone.com/javadoc/com/codename1/components/Accordion.html] displays collapsible content panels.

Simple usage of the `Accordion` class looks like:

```
Form f = new Form("Accordion", new BoxLayout(BoxLayout.Y_AXIS));
f.setScrollableY(true);
Accordion accr = new Accordion();
accr.addContent("Item1", new SpanLabel("The quick brown fox jumps over the lazy dog\n"
        + "The quick brown fox jumps over the lazy dog"));
accr.addContent("Item2", new SpanLabel("The quick brown fox jumps over the lazy dog\n"
        + "The quick brown fox jumps over the lazy dog\n "
        + "The quick brown fox jumps over the lazy dog\n "
        + "The quick brown fox jumps over the lazy dog\n "
        + ""));

accr.addContent("Item3", BoxLayout.encloseY(new Label("Label"), new TextField(), new Button("Button"), new CheckBox
("CheckBox")));

f.add(accr);
f.show();
```

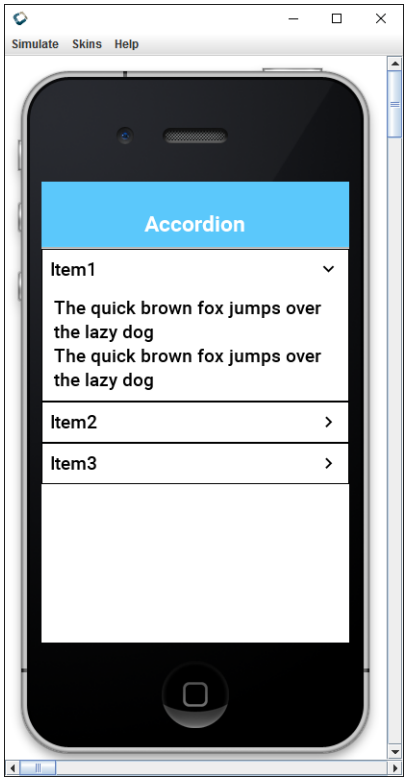


Figure 210. The Accordion Component

7.40. Floating hint

FloatingHint [https://www.codenameone.com/javadoc/com/codename1/components/FloatingHint.html] wraps a text component with a special container that can animate the hint label into a title label when the text component is edited or has content within it:

```
Form hi = new Form("Floating Hint", BoxLayout.y());
TextField first = new TextField("", "First Field");
TextField second = new TextField("", "Second Field");
hi.add(new FloatingHint(first)).
        add(new FloatingHint(second)).
        add(new Button("Go"));
hi.show();
```

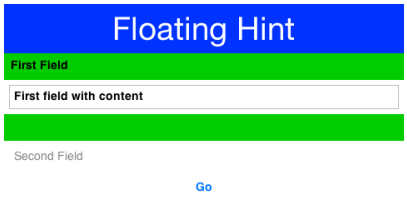


Figure 211. The FloatingHint component with one component that contains text and another that doesn't

7.41. Floating Button

The material design floating action button is a powerful tool for promoting an action within your

application.

FloatingActionButton [https://www.codenameone.com/javadoc/com/codename1/components/FloatingActionButton.html] is a round button that resides on top of the UI typically in the bottom right-hand side.
It has a drop shadow to distinguish it from the UI underneath and it can hide two or more actions under the surface. For example: you can create a simple single click button such as this:

```
FloatingActionButton fab = FloatingActionButton.createFAB(FontImage.MATERIAL_ADD);
fab.addActionListener(e -> ToastBar.showErrorMessage("Not implemented yetnull"));
fab.bindFabToContainer(form.getContentPane());
```

Which will place a + sign button that will perform the action. Or you can create a nested action where a click on the button will produce a submenu for users to pick from for example:

```
FloatingActionButton fab = FloatingActionButton.createFAB(FontImage.MATERIAL_ADD);
fab.createSubFAB(FontImage.MATERIAL_PEOPLE, "");
fab.createSubFAB(FontImage.MATERIAL_IMPORT_CONTACTS, "");
fab.bindFabToContainer(form.getContentPane());
```

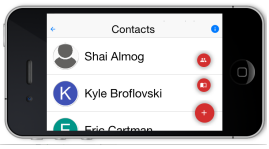


Figure 212. FloatingActionButton with submenu expanded

Those familiar with this widget know that there are many nuances to this UI that may be implemented/exposed in the future. The current API is intentionally simple and minimal for the common use cases, with the plan to refine it based on feedback.

7.41.1. Using floating Button as a badge

Floating buttons can also be used to badge an arbitrary component in the style popularized by iOS/macOS. A badge appears in the top right corner and includes special numeric details such as `unread count`..

The code below adds a simple badge to a chat button:

```
Form hi = new Form("Badge");

Button chat = new Button("Chat");
FontImage.setMaterialIcon(chat, FontImage.MATERIAL_CHAT, 7);

FloatingActionButton badge = FloatingActionButton.createBadge("33");
hi.add(badge.bindFabToContainer(chat, Component.RIGHT, Component.TOP));

TextField changeBadgeValue = new TextField("33");
changeBadgeValue.addDataChangedListener((i, ii) -> {
```

```
    badge.setText(changeBadgeValue.getText());
    badge.getParent().revalidate();
});
hi.add(changeBadgeValue);

hi.show();
```

The code above results in this, notice you can type into the text field to change the badge value:

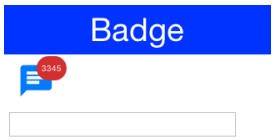


Figure 213. Badge floating button in action

7.42. SplitPane

The split pane component is a bit desktop specific but works reasonably well on devices. To get the image below you changed `SalesDemo.java` in the kitchen sink by changing this:

To:

```
private Container encloseInMaximizableGrid(Component cmp1, Component cmp2) {
    return new SplitPane(SplitPane.VERTICAL_SPLIT, cmp1, cmp2, "25%", "50%", "75%");
}
```



Figure 214. Split Pane in the Kitchen Sink Demo

This is self-explanatory but "mostly." you have 5 arguments the first 3 make sense:

- Split orientation
- Components to split

The last 3 arguments seem weird but they also make sense once you understand them, they're:

- The least position of the split - 1/4 of available space
- The default position of the split - middle of the screen
- The most position of the split - 3/4 of available space

The units don't have to be percentages they can be mm (millimeters) or px (pixels).

[1] String width is the real expensive part here, the complexity of font kerning and the recursion required to reflow text is a big performance hurdle

[2] Image by RegisFrey - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=10298177

[3] Fisheye is an effect where the selection stays in place as the list moves around it

[4] Image was fetched from http://awoiaf.westeros.org/index.php/Portal:Books

[5] The text below is from A Wiki of Ice & Fire: http://awoiaf.westeros.org/index.php/A_Game_of_Thrones

Chapter 8. Rich Text and Code Editing

Codename One includes two full-document editors. Use `RichTextArea` [https://www.codenameone.com/javadoc/com/codename1/ui/RichTextArea.html] when users compose formatted prose and the application stores HTML. Use `CodeEditor` [https://www.codenameone.com/javadoc/com/codename1/ui/CodeEditor.html] for source code, configuration files, queries, or any text that benefits from syntax highlighting, line numbers, completion, and diagnostics. Both are lightweight Codename One components and share the same editing, selection, scrolling, undo, and platform-input infrastructure. When you only need to *display* formatted text — no editing — use `RichTextComponent` [https://www.codenameone.com/javadoc/com/codename1/ui/RichTextComponent.html], the read-only, word-wrapping renderer described later in this chapter; it reuses the same document model and styling engine.

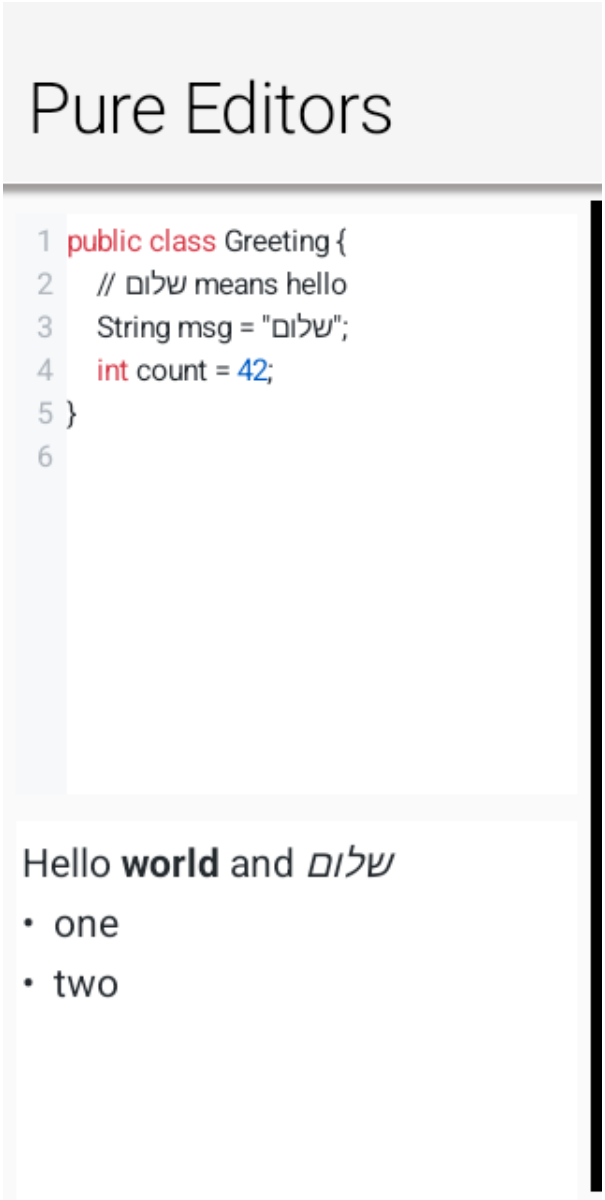


Figure 215. The rich-text and code editors rendered by the same lightweight engine

8.1. Editor architecture and backend selection

Both components extend `AbstractEditorComponent` [https://www.codenameone.com/javadoc/com/codename1/ui/AbstractEditorComponent.html]. The public components send semantic commands and queries to one

of two backends:

- A port can provide a native peer through `CodenameOneImplementation.createNativeEditorPeer(…)`.
- Otherwise the components use the pure Codename One engine in `com.codename1.ui.editor`. It paints text, selections, carets, gutters, diagnostics, and completion UI with `Graphics` and `Font`, and it drives keyboard and IME input through the low-level text-input contract.

Backend selection is transparent to application code. `isNativeEditor()` reports the native-peer case. The pure engine is the default renderer on every platform; there is no browser fallback.

Editor initialization can be asynchronous for a native peer. Commands issued immediately after construction are queued. Use `onReady(Runnable)` or a ready listener when work must run only after the backend exists. Reads use callbacks (`getHtml(…)`, `getText(…)`, and `getCursorPosition(…)`) so the same application code works with either backend.

8.1.1. The editor package

The `com.codename1.ui.editor` package separates editing mechanics from feature-specific rendering. `EditorDocument` owns normalized UTF-16 text and line offsets. `EditorView` owns the caret, selection, composition range, scrolling, hit testing, clipboard commands, and undo transactions. `CodeView` and `RichView` extend that surface instead of creating independent input implementations.

The code layer adds incremental tokenization, palettes, completion, diagnostics, indentation, and the line-number gutter. The rich layer keeps `InlineStyles`, `RichBlocks`, hyperlink targets, and image runs parallel to the text. Every mutation, including undo and redo, updates those parallel models in the same transaction. `HtmlImporter` and `HtmlSerializer` are model adapters; they don't implement a second HTML tokenizer. Fragment parsing and entity decoding go through the framework's tolerant `HTMLParser`, which extends `XMLParser`. The parser's fragment mode retains modern attributes such as `data-*` while the legacy `HTMLComponent` path keeps its existing validation behavior.

`EditorHost` is the narrow bridge between a view and its owner. It starts and stops platform text input, pushes updated editing state to the port, and reports semantic editor events. This split lets the same document and interaction engine run inside the full editors and inside the lightweight `EditField` component.

8.2. Focus, change events, and editability

`focusEditor()` moves focus into the editing surface and opens the virtual keyboard when appropriate. `blurEditor()` relinquishes focus and dismisses the active platform input session. `setEditable(false)` keeps the document visible while preventing edits; `CodeEditor.setReadOnly(true)` is the code-oriented equivalent.

Register an `addChangeListener(…)` listener to observe user edits and programmatic mutations. Avoid saving on every keystroke when serialization or storage is expensive; debounce the listener, then read the document through the asynchronous callback.

8.3. Platform text input and input methods

The pure editor doesn't treat a key release as text. Modern input can come from a virtual keyboard, autocorrect, dictation, handwriting, a hardware keyboard, or an input method editor (IME). The platform therefore drives a `TextInputClient` [https://www.codenameone.com/javadoc/com/codename1/ui/TextInputClient.html] with semantic operations:

- `commitText(…)` inserts final text, including accepted autocorrect and dictation results.
- `setComposingText(…)` replaces the current marked-text range while an IME is still composing. `finishComposing()` finalizes that range.
- `deleteSurroundingText(…)` implements soft-keyboard deletion without manufacturing key codes.
- `onKeyCommand(…)` carries navigation, selection, clipboard, undo, and redo commands.
- Geometry methods map UTF-16 offsets to caret and selection rectangles so native candidate windows, selection handles, and accessibility UI align with the lightweight rendering.

`TextInputConfig` [https://www.codenameone.com/javadoc/com/codename1/ui/TextInputConfig.html] tells the port which keyboard constraint, return-key action, autocorrection, capitalization, and multiline behavior the client needs. Rich prose enables normal language assistance; source editing disables autocorrection and automatic capitalization. The immutable `TextInputState` [https://www.codenameone.com/javadoc/com/codename1/ui/TextInputState.html] returns the authoritative text, selection, and composing range to the port after each edit.

All offsets in this contract are UTF-16 indices, matching Java `String`, Android `Editable`, and Apple string indices. The document canonicalizes CRLF and bare CR line endings to LF before it updates the caret, undo history, formatting runs, or platform state. Port authors must deliver composition updates on the EDT and must stop the input session when the client loses focus.



`TextInputClient`, `TextInputConfig`, and `TextInputState` are general port-level APIs. Custom lightweight editing components can implement the same protocol, but ordinary applications normally use `RichTextArea` or `CodeEditor` and don't bind an input client directly.

8.4. Lightweight fields with EditField

`TextArea` and `TextField` keep their established native editing behavior. When an application wants Codename One to retain control of painting, selection, scrolling, focus, and clipboard handling for a plain field, use `EditField` [https://www.codenameone.com/javadoc/com/codename1/ui/EditField.html]. It offers an API familiar from `TextField` — hint text, constraint, single-line or multiline mode, rows and columns, data-change listeners, and action events — but renders and edits entirely through the pure `com.codename1.ui.editor` engine, with no native peer. Adoption is opt-in per component; existing `TextField` and `TextArea` usages are unaffected.

```
EditField title = new EditField("", "Title", TextArea.ANY);
title.setSingleLineTextArea(true);
title.setColumns(80);
```

```
EditField notes = new EditField();
notes.setSingleLineTextArea(false);
notes.setRows(5);
notes.setColumns(30);
```

`EditField` extends `EditorView` directly, so selection, caret navigation, arrow-key movement, undo, and clipboard commands are the same mechanics the full editors use. A single-line `EditField` fires its action event on the return key and strips embedded newlines; a multiline `EditField` grows to its configured row count. Because there is no native overlay, the field behaves identically on every platform that implements the low-level text-input contract.

8.5. Selection, bidirectional text, and scrolling

On touch devices, a tap places the caret, a long press selects a word, and draggable handles adjust the selection. The loupe follows a handle or long-press placement gesture. A normal one-finger drag scrolls the document and doesn't turn the swipe into a selection. On desktop, mouse drag selects text, double-click selects a word, triple-click selects a line, and the mouse wheel or trackpad scrolls the document. Standard command/control shortcuts handle copy, cut, paste, select all, undo, and redo.

Hebrew, Arabic, and mixed left-to-right/right-to-left lines use Unicode bidirectional layout. Glyph painting, hit testing, caret movement, and selection rectangles share the same visual runs. Set the component's RTL flag to choose the paragraph base direction.

8.6. RichTextArea

`RichTextArea` accepts and returns an HTML fragment. The importer maps common paragraph, heading, blockquote, preformatted, list, link, image, bold, italic, underline, strike-through, color, highlight, and relative-size markup into a document model. `getHtml(…)` emits canonical HTML from that model; it preserves meaning rather than the input's exact whitespace, tag aliases, or attribute order.

```
Form hi = new Form("Compose", new BorderLayout());
RichTextArea editor = new RichTextArea();
editor.setPlaceholder("Write something...");
editor.setHtml("<h2>Trip itinerary</h2><p>Meet at the <b>main lobby</b>.</p>");

Toolbar tb = hi.getToolbar();
tb.addCommandToRightBar("B", null, e -> editor.bold());
tb.addCommandToRightBar("I", null, e -> editor.italic());
tb.addCommandToRightBar("List", null, e -> editor.insertUnorderedList());
tb.addCommandToRightBar("Save", null, e ->
        editor.getHtml(html -> Log.p("User wrote: " + html)));

hi.add(BorderLayout.CENTER, editor);
hi.show();
```

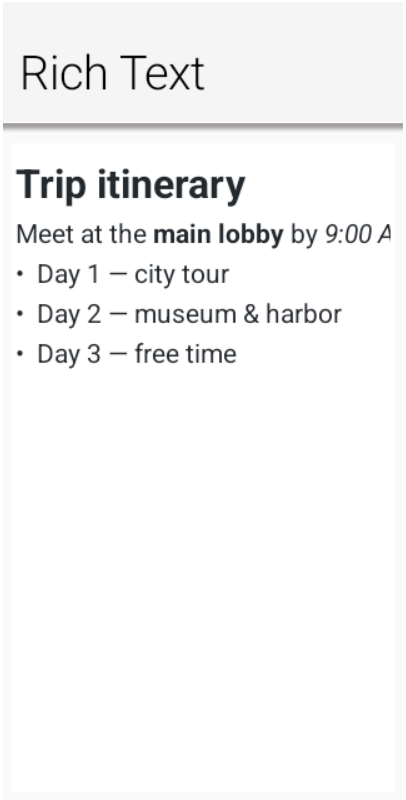


Figure 216. RichTextArea rendering headings, inline formatting, and a list

The formatting methods act on the current selection: `bold()`, `italic()`, `underline()`, `strikeThrough()`, ordered and unordered lists, indentation, alignment, foreground and highlight colors, block formats, and font sizes. `queryCommandState(…)` keeps toolbar toggles synchronized with the selection. `removeFormat()` clears inline formatting; `undo()` and `redo()` restore text together with inline styles, paragraph attributes, links, and image runs.

Use `setHtml(…)` to replace the document and `insertHtml(…)` to insert a fragment at the selection. Fragment insertion retains its formatting and block structure. `createLink(url)` stores the target in the model, and `removeLink()` removes it. `insertImage(url)` accepts normal URLs, data URIs, and supported application resources; the source is retained so `getHtml(…)` can emit an `img` element when the document is saved and reloaded.

8.6.1. HTML, RTF, Markdown, and AsciiDoc

`setContent(…)` and `insertContent(…)` accept a `RichTextFormat`. HTML is parsed directly with the shared `HTMLParser`. Markdown and AsciiDoc use their own bidirectional model adapters; they never pass through HTML. Headings, emphasis, inline code, links, images, lists, quotes, and literal blocks are mapped directly to the rich model and serialized directly back with `getMarkdown(…)` or `getAsciiDoc(…)`. The RTF importer likewise maps paragraph breaks, Unicode escapes, bold, italic, underline, strike-through, and skips metadata destinations such as font and color tables.

Editing and reading the same format returns canonical output for that format. Requesting a different output format performs an explicit model conversion, just as `getHtml(…)` canonicalizes a visual rich-text document.

```
editor.setContent("# Release notes\n\nThis is **ready**.", RichTextFormat.MARKDOWN);
editor.insertContent("{\\rtf1\\ansi {\\i pasted notes}}", RichTextFormat.RTF);
```

```
editor.insertContent("== Details\n\n* Portable\n* Lightweight", RichTextFormat.ASCIIDOC);
```

Rich paste uses `ClipboardContent`, a MIME-keyed container exposed by `Display.copyToClipboard(…)` and `Display.getClipboardContent()`. `RichTextClipboardData` is its convenience subtype for `text/plain`, `text/html`, `text/rtf`, `text/markdown`, and `text/asciidoc`. A writer publishes all representations in one operation and a reader selects the richest format it understands; plain `EditorView` consumers use `text/plain`. Copying a rich selection publishes canonical HTML, RTF, Markdown, AsciiDoc, and the required plain fallback. Java SE publishes native `DataFlavor` values, Android publishes native plain and HTML clip data, the JavaScript port uses `ClipboardItem` when the browser allows it, and iOS publishes pasteboard types. A port that only has a text clipboard still publishes plain text externally and retains the complete `ClipboardContent` for in-application use.

The negotiation API belongs at the implementation boundary, not inside `RichTextArea`. Port authors override clipboard handling in `CodenameOneImplementation` and return the formats actually present on the system clipboard. Applications can use the same API for domain-specific formats without changing the editor.

RTF is also recognized when a clipboard provider returns it as a string beginning with an RTF header. Format import is intentionally a practical interchange subset, not a layout-compatible word processor: unsupported RTF destinations and advanced Markdown or AsciiDoc extensions degrade to the supported text and structure rather than controlling document layout.



Treat imported HTML as document data, not as a browser page. The pure importer doesn't execute scripts. Keep image URLs and link destinations subject to the same trust and validation rules used by the rest of the application.

8.7. Read-only rich text with RichTextComponent

`RichTextComponent` (in `com.codename1.ui`) is the read-only counterpart to `RichTextArea`. It renders multi-styled, word-wrapped text — headings, bold, italic, underline, strike-through, inline code, colored and highlighted spans, per-paragraph alignment and indentation, ordered and unordered lists, block quotes, preformatted blocks, inline images, and tappable hyperlinks — without any of the editing machinery. It sits between a single-style `SpanLabel` (one style for the whole component) and a full `BrowserComponent` (a native web view): a lightweight native component that keeps a distinct style per character, measures and paints text directly, and reports an accurate height-for-width preferred size, so it embeds cleanly inside ordinary layouts. It's the modern replacement for the deprecated `HTMLComponent`.

It shares the editor's document model and the `RichRunPainter` styling primitive, so styled text rendered read-only matches what the editor produces for the same content.

Content is supplied in any of the interchange formats the rich model understands, or built up programmatically from styled runs:

```
RichTextComponent view = new RichTextComponent();

// From Markdown, HTML, or any RichTextFormat:
view.setMarkdown("# Trip summary\n\n"
```

```
        + "Departs **09:40**, arrives *11:15*. See the [itinerary](app://itinerary) for details.\n\n"
        + "- Window seat\n"
        + "- Carry-on only");
// view.setHtml("<h1>Trip summary</h1><p>Departs <b>09:40</b> ...</p>");
// view.setContent(source, RichTextFormat.ASCIIDOC);

// Or assembled run by run with editor TextStyle values:
RichTextComponent built = new RichTextComponent();
built.append("Status: ", TextStyle.DEFAULT)
        .append("confirmed", TextStyle.DEFAULT.withBold(true).withForeColor(0x1a7f37));

form.add(view);
```

Hyperlinks in the content fire an `ActionListener` whose event source is the link target string, so the application decides what a link means:

```
view.addLinkListener(e -> Display.getInstance().execute((String) e.getSource()));
```

The component resolves inline images through an `ImageResolver`, keeping image loading and caching policy in the application rather than the component. Return `null` to render a placeholder:

```
view.setImageResolver(src -> myImageCache.get(src));
```

By default (`SizeMode.SHRINK`) the component is exactly as tall as its wrapped content at the width it is given, which is ideal inside a scrollable `Form`. `SizeMode.SCROLL` instead keeps the size the parent assigns and scrolls the content vertically. `setTextAlign(…)` overrides the alignment of every paragraph, and `preferredSizeForWidth(width)` is a stateless height-for-width query for custom layout managers.



The same trust guidance applies as for `RichTextArea` — imported HTML is document data, not a browser page; validate link targets and image sources with the application's own rules.

8.8. CodeEditor

`CodeEditor` provides incremental highlighting for `java`, `kotlin`, `javascript`, `python`, `css`, `xml`, `json`, and `c`. It includes a line-number gutter, light and dark palettes, automatic bracket and quote closing, tab expansion, automatic indentation, horizontal and vertical scrolling, and read-only viewing.

```
Form hi = new Form("Editor", new BorderLayout());
CodeEditor editor = new CodeEditor();
editor.setLanguage("java");
editor.setTheme("light");          // or "dark"
editor.setShowLineNumbers(true);
editor.setText("public class Main {\n\n}");
hi.add(BorderLayout.CENTER, editor);
```

```
hi.show();
```

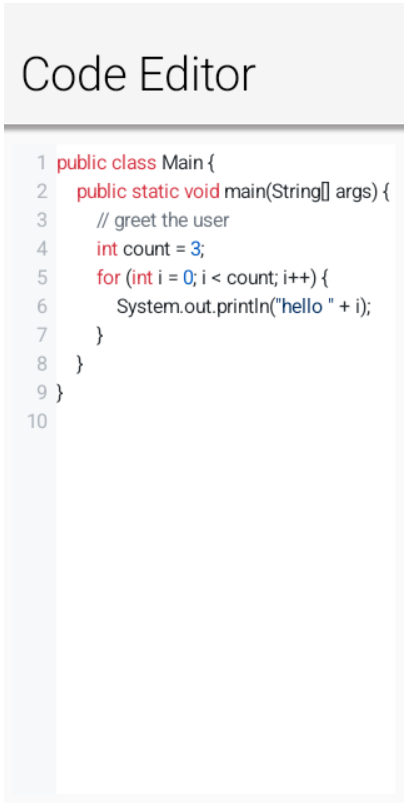


Figure 217. CodeEditor with Java syntax highlighting and line numbers

Choose the language with `setLanguage(…)`, select `"light"` or `"dark"` with `setTheme(…)`, toggle the gutter with `setShowLineNumbers(…)`, and configure indentation with `setTabSize(…)`. `insertAtCursor(…)` replaces the active selection, while `getCursorPosition(…)` reports the current UTF-16 caret offset asynchronously.

The Codename One Playground uses `CodeEditor` directly for its Java and CSS source panes. This uses the same language highlighting, line-number gutter, keyboard navigation, diagnostics, and diagnostic tooltips available to applications. Compiler diagnostics are passed to `setDiagnostics(…)` with their source ranges; status messages use lightweight Codename One components beneath the editor. The Playground doesn't install or bundle a separate editor engine. Source editing disables platform spelling and autocorrection; the only wavy underlines the editor draws are diagnostics supplied through `setDiagnostics(…)`.

8.8.1. Custom syntax highlighters

Built-in languages use grammar-specific incremental lexical analyzers. JSON doesn't accept single-quoted strings, CSS doesn't interpret `//` as a comment, XML tracks tags and attributes, Python carries triple-quoted strings across lines, and C-family numeric prefixes are validated by radix. These rules are lexical by design; a compiler or language server remains the authority for diagnostics.

Third-party libraries can register a `SyntaxHighlighter` under any language ID. The highlighter receives one line and the state returned for the preceding line, then returns ordered token spans and the next state. Tokens can use the built-in semantic kinds or provide explicit light and dark RGB colors. Registration is global and the pure renderer invokes the Java highlighter directly, so a

custom language needs no platform-specific grammar.

```
SyntaxHighlighter properties = (line, state) -> {
    List<SyntaxToken> tokens = new ArrayList<>();
    int equals = line.indexOf('=');
    if (equals > 0) {
        tokens.add(new SyntaxToken(0, equals, SyntaxToken.PROPERTY,
                0x005cc5, 0x9cdcfe));
    }
    return new SyntaxHighlightResult(tokens, 0);
};
CodeEditor.registerSyntaxHighlighter("properties", properties);

CodeEditor editor = new CodeEditor("properties", "accent=#0a66c2");
```

8.8.2. Asynchronous code completion

A `CodeCompletionProvider` receives the complete source and caret offset, then delivers proposals through its callback. The callback can run after a local analysis or network/language-server request; the editor doesn't block the EDT while proposals are produced.

```
editor.setCompletionProvider((sourceEditor, code, cursor, results) -> {
    List<CodeCompletion> proposals = new ArrayList<>();
    proposals.add(new CodeCompletion("println(String value)", "println()")
            .setType("method")
            .setDetail("void"));
    proposals.add(new CodeCompletion("private").setType("keyword"));
    results.onSucess(proposals);
});
```

Return an empty list or `null` to dismiss the popup. A proposal can use different display and insertion text, plus a type badge and detail string.

8.8.3. Diagnostics

`setDiagnostics(…)` accepts 1-based source ranges. Errors, warnings, and informational entries appear under only the supplied character range, with a gutter marker for the affected line. Hover the range or marker on desktop, or tap it on touch devices, to display its message.

```
editor.setDiagnostics(Arrays.asList(
        new CodeDiagnostic(6, 9, 6, 16, "Cannot resolve method prinln")
                .setSeverity(CodeDiagnostic.ERROR),
        new CodeDiagnostic(3, 5, "Value is never read")
                .setSeverity(CodeDiagnostic.WARNING)
));
```

Replace the list whenever a compiler, linter, or language server publishes a newer analysis. Passing

an empty list clears the current diagnostics.

8.9. Port integration checklist

The pure engine drives the editors on every platform, so a port must implement the low-level text-input contract: `isTextInputSupported()`, `startTextInput(…)`, `updateTextInputState(…)`, and `stopTextInput(…)` in `CodenameOneImplementation`. The binding must distinguish committed from composing text, preserve UTF-16 offsets, map native navigation and clipboard actions to `TextInputClient.KEY_*`, update native selection state after programmatic edits, and anchor candidate UI with the supplied caret geometry.

A port with a better platform editor can instead supply a native peer and implement the existing semantic command/query channel. In both cases application code, change listeners, ready handling, and asynchronous queries remain the same.

Accessibility Semantics

Codename One components are lightweight: they’re painted by Codename One instead of being represented by a native widget on every platform. The accessibility semantics API builds an explicit, immutable virtual tree from those components and presents it to VoiceOver, TalkBack, Windows UI Automation, AT-SPI, Java Access Bridge, and web accessibility APIs.

The semantic tree is separate from the visual tree. This is important for renderer-backed controls, merged cards, decorative content, custom controls, and applications that need a reading order different from paint order.

Chapter 9. Built-in semantics

Standard components work without extra configuration. `Button`, `CheckBox`, `RadioButton`, `Slider`, text fields, lists, tables, tabs, labels, dialogs, and containers infer appropriate roles, values, states, collection metadata, and standard actions. Renderer-backed `List` rows are exposed as stable virtual children even though no `Component` exists for each painted row.

`Component.setAccessibilityText()` remains supported. It's a compatibility alias for the semantic label. New code should use `getSemantics()` when it needs more than a label.

```
Button save = new Button("Save");
save.getSemantics()
        .setHint("Saves the edited profile")
        .setIdentifier("profile-save");
```

Changes to text, selection, focus, enabled state, bounds, scrolling, ranges, tabs, and the current `Form` automatically invalidate the native semantic tree. Custom components should call `accessibilityChanged()` after a semantic value changes outside a standard setter.

Chapter 10. Roles, names, values, and state

`AccessibilityRole` contains portable control roles including buttons, toggle buttons, switches, checkboxes, radio buttons, headings, links, images, text and search fields, sliders, progress indicators, lists, grids, rows, cells, headers, tabs, dialogs, alerts, menus, toolbars, combo boxes, trees, and separators. A port maps an unsupported role to the closest native role without discarding its label or state.

```
wifiSwitch.getSemantics()
        .setRole(AccessibilityRole.SWITCH)
        .setLabel("Wi-Fi")
        .setChecked(AccessibilityCheckedState.CHECKED)
        .setEnabled(Boolean.TRUE)
        .setHint("Double tap to turn Wi-Fi off");
```

title.getSemantics() .setHeadingLevel(2) .setIdentifier("network-heading");

Tri-state controls use `UNCHECKED`, `CHECKED`, or `MIXED`. Nullable Boolean properties distinguish an explicit state from an unspecified state. Available properties include selected, expanded, enabled, invalid, busy, read-only, required, multiline, obscured, pressed, and current. `setValidationError()` supplies both the error message and invalid state. `setPaneTitle()` describes a newly displayed pane or screen, and `setModal(true)` identifies a modal scope.

Use `setRoleDescription()` only when the platform role isn’t sufficiently specific. Localize labels, hints, descriptions, errors, pane titles, role descriptions, and custom action labels.

Chapter 11. Ranges and editable values

`AccessibilityRange` describes minimum, maximum, current value, increment, and an optional spoken value. The spoken value is useful for ranges such as ratings and durations where a raw number is ambiguous.

```
rating.getSemantics()
        .setRole(AccessibilityRole.SLIDER)
        .setLabel("Rating")
        .setRange(new AccessibilityRange(0, 5, 3, 1, "3 out of 5"));
```

Editable controls can provide `SET_VALUE` or `SET_TEXT` actions. Sliders and other adjustable controls should expose `INCREMENT` and `DECREMENT`. Standard components infer these actions.

Chapter 12. Custom accessibility actions

An `AccessibilityAction` has a stable ID, a localized label, an enabled state, and a handler. Handlers always execute on the Codename One EDT, including when the action originates on a native accessibility thread.

```
card.getSemantics().addAction(new AccessibilityAction(
        "archive",
        "Archive message",
        new AccessibilityAction.Handler() {
            public boolean perform(Component component, Object argument) {
                archiveMessage();
                return true;
            }
        }
));
```

Use the standard IDs in `AccessibilityAction` for activation, long press, increment, decrement, set value, set text, selection, character or word cursor movement, focus, show-on-screen, dismiss, expand, collapse, scrolling, and clipboard operations. A custom ID must be stable and its label must not be null.

Chapter 13. Grouping and virtual descendants

`AccessibilityGrouping` controls how a component participates in the semantic tree:

- `AUTO` uses normal component inference.
- `LEAF` exposes the component but none of its descendants.
- `GROUP` exposes the component as a semantic container and retains its descendants.
- `MERGE_DESCENDANTS` combines descendant labels, descriptions, values, state, and actions into one node.
- `EXCLUDE` removes the component but promotes its descendants to the semantic parent.
- `EXCLUDE_SUBTREE` removes the component and all descendants.

```
summaryCard.getSemantics()
        .setRole(AccessibilityRole.GENERIC)
        .setGrouping(AccessibilityGrouping.MERGE_DESCENDANTS);
```

For content that's painted without child components, attach an `AccessibilityChildProvider`, or add standalone `AccessibilityNode` children. Every virtual node must use a stable `virtualKey`. Stable keys preserve native focus while rows are updated or recycled.

```
chart.getSemantics().setChildProvider(new AccessibilityChildProvider() {
    public List<AccessibilityNode> getAccessibilityChildren(Component owner) {
        List<AccessibilityNode> result = new ArrayList<AccessibilityNode>();
        for (ChartPoint point : points) {
            AccessibilityNode node = new AccessibilityNode("point-" + point.getId());
            node.setRole(AccessibilityRole.IMAGE)
                    .setLabel(point.getLabel())
                    .setValue(point.getFormattedValue())
                    .setBounds(point.getBounds());
            result.add(node);
        }
        return result;
    }
});
```

Virtual bounds are relative to the owning component. Component nodes use their clipped absolute bounds.

Chapter 14. Traversal order

Visual order is the default reading order. `setSortKey()` assigns an ordered semantic key among siblings. `setTraversalBefore()` and `setTraversalAfter()` express a direct relationship when a numeric order would be fragile.

```
cancel.getSemantics().setSortKey(1);
continueButton.getSemantics().setSortKey(2);
help.getSemantics().setTraversalAfter(continueButton);
```

Keep traversal relationships within the same semantic parent. The tree builder resolves these constraints deterministically and preserves stable node IDs.

Chapter 15. Live regions and screen changes

Set `AccessibilityLiveRegion.POLITE` for non-urgent updates and `ASSERTIVE` for errors or urgent status. `OFF` is the default. Live changes produce the corresponding platform notification or ARIA live update.

```
status.getSemantics()
        .setRole(AccessibilityRole.ALERT)
        .setLiveRegion(AccessibilityLiveRegion.POLITE);
status.setText("Upload complete");
```

Form changes and nodes with a pane title produce screen or pane transition notifications. `Display.announceForAccessibility()` remains available for an exceptional announcement that isn't represented by persistent UI. Prefer a live region for state that's visible on screen.

Chapter 16. User accessibility preferences

Semantics describe what a control means. Accessibility preferences describe how the user wants the interface presented. Read these preferences through `Display` or the matching static shortcuts in `CN`:

API	Meaning
`isHighContrastEnabled()`	The user requests stronger foreground and background contrast.
`isDifferentiateWithoutColorEnabled()`	The user requests labels, shapes, or patterns in addition to color.
`getColorVisionDeficiency()`	Returns `NONE`, `PROTANOPIA`, `DEUTERANOPIA`, `TRITANOPIA`, `MONOCHROMACY`, or `UNKNOWN`.
`isReduceMotionEnabled()`	Nonessential motion should be removed or replaced with an instant or cross-fade change.
`isReduceTransparencyEnabled()`	Blur and translucent surfaces should use an opaque alternative.
`isBoldTextEnabled()`	The system requests a heavier text weight.
`isInvertColorsEnabled()` and `isGrayscaleEnabled()`	The operating system is transforming the displayed colors.
`isOnOffSwitchLabelsEnabled()`	Switches should include a visible on/off distinction.
`isScreenReaderEnabled()`	A screen reader or touch-exploration service is active.

An unavailable preference returns `false`; an unavailable color-vision mode returns `UNKNOWN`. Don't hide accessibility support behind `isScreenReaderEnabled()`. A screen reader can start after the application launches, and other assistive technologies consume the same semantic tree.

```
if (CN.isHighContrastEnabled()) {
    animatedChart.setUIID("HighContrastChart");
}

// Important state always uses text or an icon as well as color. The
// preference is an extra signal, not permission to use color alone.
if (CN.isDifferentiateWithoutColorEnabled()
        || CN.getColorVisionDeficiency() != AccessibilityColorVisionDeficiency.NONE) {
    connectionStatus.setText("Disconnected: action required");
}

if (CN.isReduceMotionEnabled()) {
    animatedChart.putClientProperty("animate", Boolean.FALSE);
}
if (CN.isReduceTransparencyEnabled()) {
    animatedChart.setUIID("OpaqueChart");
}
```



Never use `getColorVisionDeficiency() == NONE` as permission to convey information only through color. The setting can be unavailable, a person might not enable a system filter, and contrast can change with lighting and display hardware. Design the default interface with redundant cues, then use preferences for further adaptation.

Preference detection uses each platform's public signals. iOS and Mac Catalyst expose contrast, differentiate-without-color, motion, transparency, bold text, inversion, grayscale, switch labels, and VoiceOver; Apple doesn't expose the selected color-filter type, so the deficiency value is `UNKNOWN`. Android reads high-text contrast, display inversion and color-correction mode, animation scale, font-weight adjustment, and touch exploration. Windows reads High Contrast, client-area animation, and screen-reader settings. Linux detects the GTK HighContrast theme, disabled GTK animation, and ATK bridge activation. JavaScript uses `forced-colors`, `prefers-contrast`, `prefers-reduced-motion`, and `prefers-reduced-transparency`. The Java SE simulator provides deterministic controls for every portable signal.

Chapter 17. Collections

`AccessibilityCollectionInfo` describes row and column counts, hierarchy, and selection mode. `AccessibilityCollectionItemInfo` describes row and column index, spans, position and size in a set, nesting level, and header status.

```
grid.getSemantics().setCollectionInfo(
        new AccessibilityCollectionInfo(20, 3, false,
                AccessibilityCollectionInfo.SELECTION_MULTIPLE));
```

cell.getSemantics().setCollectionItemInfo(new AccessibilityCollectionItemInfo(4, 1, 2, 1, 5, 20, 1, false));

Row and column indexes are zero-based. Position and set size are one-based; use -1 when unknown. Lists, tables, tabs, and their items infer this metadata where possible.

Renderer-backed lists keep the full model size in their collection metadata but materialize only the visible rows, a small navigation buffer, and the selected row as virtual children. Forward and backward semantic scroll actions move the window. This keeps accessibility-tree work proportional to the viewport instead of invoking a cell renderer for every row whenever a large list scrolls.

Chapter 18. Inspecting and testing semantics

`AccessibilityInspector` returns an immutable snapshot. A snapshot is safe to inspect without racing native accessibility callbacks and includes stable IDs, hierarchy, bounds, resolved state, actions, range and collection data. `toJson()` is suitable for diagnostic reports and tooling.

Semantic setters always mark the cached tree dirty. Pull-based native ports rebuild it on the Codename One EDT only while assistive technology is active or when a tool explicitly requests a snapshot; the browser keeps its off-screen ARIA projection synchronized continuously. Applications should still publish complete semantics unconditionally—this runtime optimization is transparent to application code.

```
AccessibilityTreeSnapshot tree =
        AccessibilityInspector.snapshot(myForm);
```

AccessibilityAssertions.assertNoErrors(tree);
AccessibilityAssertions.assertNoUnlabeledInteractiveNodes(tree);

String diagnosticJson = tree.toJson(); AccessibilityNodeSnapshot hit = tree.getNodeAt(screenX, screenY);

`AccessibilityAssertions.audit()` reports machine-readable `AccessibilityIssue` values for unlabeled controls, invalid ranges, duplicate identifiers or actions, invalid collection positions, missing parents, cycles, unnamed dialogs, and unlabeled custom actions. Warnings also identify suspicious role/state combinations and empty bounds.

The Java SE Component Inspector provides **Copy Accessibility Tree JSON** and **Audit Accessibility Tree** actions. Tests should assert resolved semantic properties instead of platform-specific spoken sentences. The `scripts/hellocodenameone` suite contains the cross-port conformance fixture used by native CI.

Chapter 19. Diagnosing and correcting an inaccessible screen

Accessibility review works best as a repeatable loop instead of a final checklist:

1. Use the keyboard to reach every operation. Verify visible focus, logical order, escape from modal UI, and operation without pointer gestures.
2. Open **Tools → Component Inspector**. Select the form or a suspicious component, then choose **Audit Accessibility Tree**. Start with errors, such as an unlabeled interactive node, invalid range, or unnamed dialog.
3. Choose **Copy Accessibility Tree JSON** and inspect the resolved role, label, value, state, actions, bounds, and child order. This catches cases where the visual component hierarchy doesn't match the virtual accessibility hierarchy.
4. Use **Simulate → Accessibility Preferences** to test high contrast, reduced motion, reduced transparency, inversion, grayscale, and each color-vision filter. Also test **Simulate → Larger Text** at Accessibility 5. Look for clipped text, color-only state, invisible focus, unreadable overlays, and motion that remains essential to understanding the result.
5. Correct the component API or theme. Run the audit again, add a focused assertion, and verify the native screen reader before considering the issue closed.

The simulator menu changes the values returned by `Display` and `CN`. Color-vision, inversion, and grayscale choices also filter the simulator canvas, so screenshots show approximately what remains distinguishable. These filters are diagnostic approximations; they don't replace testing with platform display filters or users.

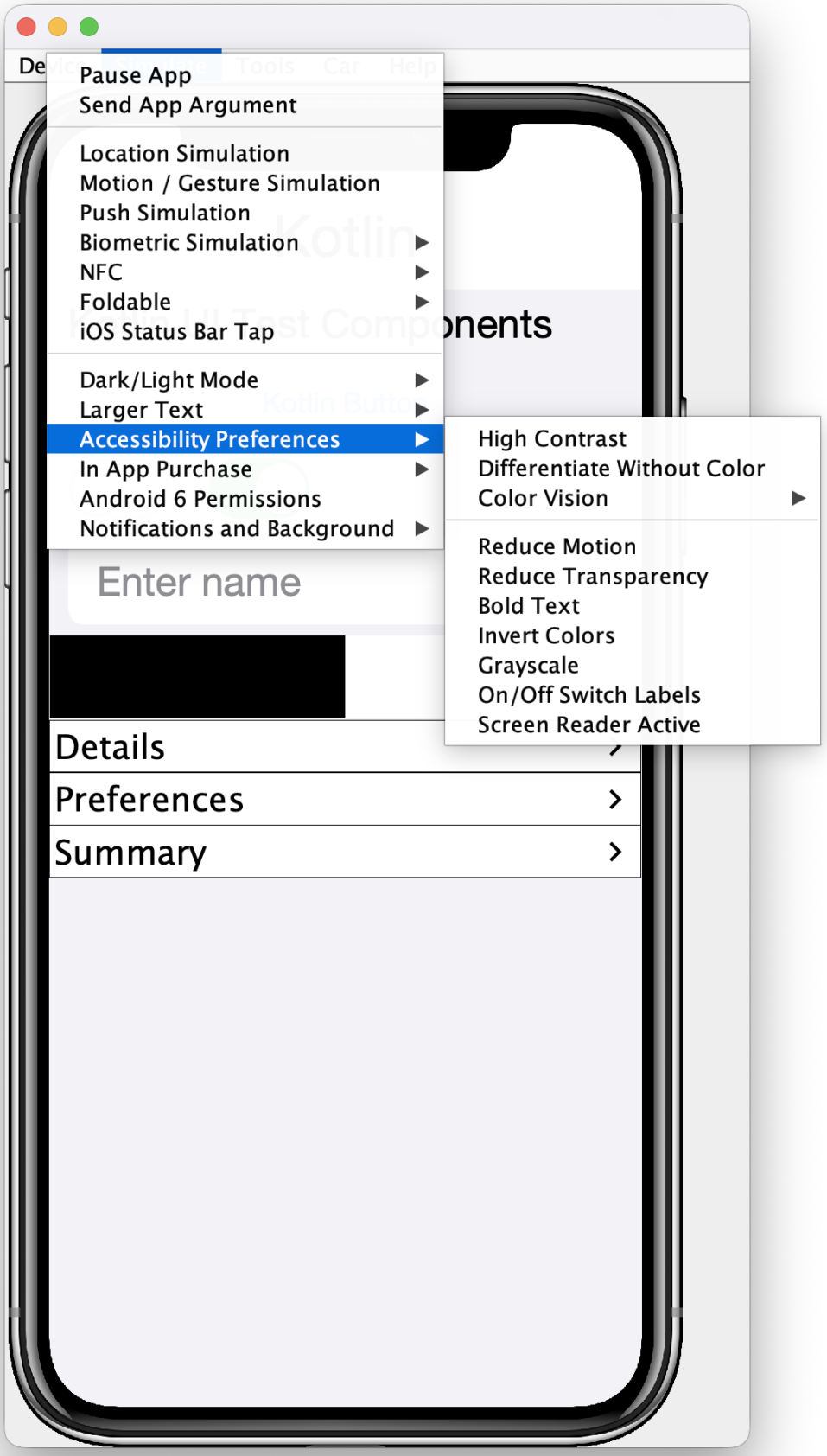


Figure 218. Accessibility preferences in the Java SE simulator

The following icon-only custom control is painted and responds to a pointer, but initially has no

usable name, role, or keyboard-equivalent action. The audit reports it as an unlabeled interactive node. Repair the semantic contract at the custom-control boundary:

```
iconOnlyButton.getSemantics()
        .setRole(AccessibilityRole.BUTTON)
        .setLabel("Delete message")
        .setHint("Removes this message from the conversation")
        .setIdentifier("message-delete")
        .addAction(new AccessibilityAction(
                AccessibilityAction.ACTIVATE,
                null,
                new AccessibilityAction.Handler() {
                    public boolean perform(Component component, Object argument) {
                        deleteMessage();
                        return true;
                    }
                }));
```

Don’t fix the warning by adding a label alone if the role, current state, range, or available actions are still wrong. For example, a custom switch needs `SWITCH`, checked state, and activation; a chart needs virtual descendants for its meaningful data points; a renderer-backed grid needs collection and item metadata.

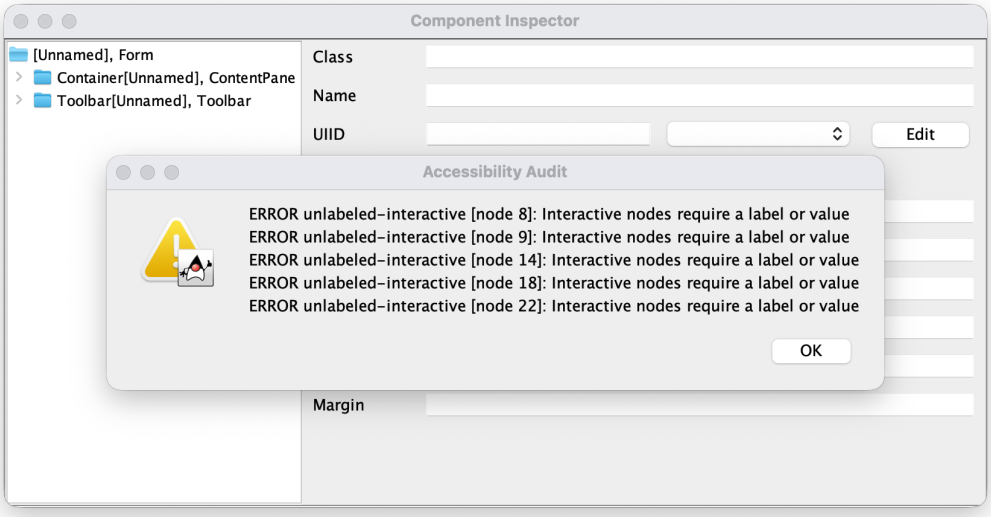


Figure 219. Auditing the semantic tree in the Component Inspector

19.1. Assertions that prevent regressions

Snapshot the finished form and assert behavior that matters to a user. Prefer stable identifiers and resolved semantics over implementation class names or platform-specific speech:

```
AccessibilityTreeSnapshot tree = AccessibilityInspector.snapshot(form);
AccessibilityAssertions.assertNoErrors(tree);
AccessibilityAssertions.assertNoUnlabeledInteractiveNodes(tree);
```

```
AccessibilityNodeSnapshot save = tree.getNodeByIdentifier("profile-save");
assert save.getRole() == AccessibilityRole.BUTTON;
assert save.getAction(AccessibilityAction.ACTIVATE) != null;
```

Keep targeted assertions for traversal order, validation errors, live-region behavior, ranges, and stable virtual keys. The `scripts/hellocodenameone` accessibility fixture invokes every portable preference API and exercises the resolved semantic tree on each native CI platform.

19.2. Native verification

The simulator finds portable problems, but the final pass must use VoiceOver, TalkBack, Narrator, Orca, or the browser accessibility tree. Navigate in both directions, activate every action, change adjustable values, enter and leave collections, trigger errors, and confirm that focus moves sensibly after a dialog, navigation, deletion, or live update. Record the platform, assistive-technology version, input method, and failed semantic node in the bug report; attach the inspector JSON when the native result differs from the portable snapshot.

Chapter 20. Platform mappings

Target	Native representation	Notable behavior
iOS, iPadOS, and Mac Catalyst	`UIAccessibilityElement` instances owned by the rendering view	Traits, values, frames, custom actions, adjustable actions, live announcements, and screen/layout transitions
Android	`AccessibilityNodeProvider` virtual descendants	Class/role mapping, checked and mixed state, headings, pane titles, errors, ranges, collections, focus, hit testing, and custom actions
Windows native	UI Automation fragment-root provider	Virtual fragments, control types, names, values, state, bounds, navigation, focus lookup, hit testing, and action dispatch
Linux native	Transparent GTK semantic widgets exposed through ATK/AT-SPI	Roles, names, descriptions, state, hierarchy, bounds, and action dispatch
Java SE simulator and desktop	Swing `AccessibleContext` virtual children	Roles, states, values, geometry, accessible actions, and Component Inspector integration
JavaScript	A synchronized off-screen DOM tree	ARIA roles, names, state, ranges, live regions, collection metadata, traversal order, and action routing; the painted canvas is hidden from the accessibility tree

The API intentionally goes beyond a label-only model: it includes stable virtual children, explicit traversal constraints, collection spans and set metadata, pane transitions, machine-readable inspection, and built-in semantic audits. Platform accessibility tools remain the final verification step because screen readers can apply platform-specific presentation rules.

For underlying platform guidance, see the Apple UIAccessibility protocol [https://developer.apple.com/documentation/uikit/uiaccessibility-protocol], Android accessibility principles [https://developer.android.com/guide/topics/ui/accessibility/principles], Microsoft UI Automation provider overview [https://learn.microsoft.com/windows/win32/winauto/uiauto-providersoverview], and W3C WAI-ARIA [https://www.w3.org/TR/wai-aria-1.2/].

MCP Headless API

The Model Context Protocol (MCP) headless API lets an application expose itself to a large language model agent. An agent such as Claude Desktop, Claude Code, Codex, or opencode can read the current screen, drive the user interface, and call tools the application publishes.

The API reuses the accessibility semantics tree. The same immutable tree that describes the screen to VoiceOver and TalkBack also describes it to the agent, so any application is drivable without extra code. Application specific data and actions are exposed through the existing `com.codename1.ai.Tool` contract.

The socket transport runs anywhere the port can bind the loopback interface, which includes an application running on a device, so an agent can attach to a build on a phone as well as to the simulator. It's blocked on a release build; see Development builds only. The stdio transport needs process standard input and so is supported by the JavaSE port, which powers the simulator and the desktop tooling.

Chapter 21. The MCP menu

Every desktop Codename One tool, including the simulator and Codename One Settings, gets a native `MCP` pull-down menu. The menu is added by the JavaSE port, so a tool gets it without any code of its own. It's the main way a user turns a tool into something an agent can drive.

The menu has these items:

- **Expose This Tool To Agents** starts and stops the loopback MCP server for the running tool.
- **Install in MCP Hosts** registers this tool with the MCP hosts detected on the machine, such as Claude Desktop, Claude Code, Codex, and opencode. After a host restarts, the tool appears as an MCP server the agent can use.
- **Remove From MCP Hosts** removes that registration.
- **Detect MCP Hosts** lists the hosts found on the machine and where their configuration lives.
- **Debug Logging** controls how much of the MCP conversation is echoed to the log.

Install writes a command that launches `MCPStdioLauncher`, a small bridge that relays the host's standard input and output to the running tool's socket. Hosts speak stdio and the tool serves a socket, so the bridge lets a stdio host such as Claude Desktop drive the already-running, human-visible tool.

Chapter 22. Starting the server

Invoking the API is the switch. There is no build hint or property to set. The socket server lets an agent attach to a session a person is watching, which is the usual choice in the simulator:

```
MCP.startSocketServer(8765);
```

A tool that a host launches as a subprocess can serve over standard input and output instead:

```
MCP.startStdioServer();
```

The stdio transport is the standard MCP local transport, exchanging newline delimited JSON-RPC messages. While it runs, application logging is redirected away from standard output so it can’t corrupt the protocol stream. The stdio transport lives in the JavaSE port because it needs process standard input, which isn’t available on every target.

Chapter 23. Development builds only

The socket server refuses to bind on a release build and throws an `IllegalStateException` instead.

The reason is what an attached agent can do. It reads the screen and drives the user interface, and the loopback interface is shared by everything on the device rather than being private to one application. On a phone that means any other installed application can connect to the port and drive yours. That's a reasonable trade while you're developing, because it's how an agent attaches to a running build, and a poor one in an application a user installs.

A development build is a debuggable Android package or a development provisioned iOS build. Any other target counts as a release build, so a port that can't tell the difference withholds the server rather than exposing it.

The JavaSE port is the exception, and it's worth understanding before you rely on this. That port runs the simulator, the designer and the desktop tooling, and it can't distinguish those from a desktop application packaged for distribution: there's no debuggable flag and no provisioning profile to read. It reports a development build in every case, because reporting otherwise would take the MCP menu away from every desktop tool. A packaged desktop application is therefore not gated by this, and one that needs to withhold the server from its own release build has to supply that signal itself.

Nothing binds a port unless the application calls a starter, and an application that never calls one has no server at all. The gate matters for the case where a starter is left in the code by accident, which is the way a debugging facility usually reaches production.

The same distinction is available to application code, so a starter can be left in place and simply not fire in a shipped build:

```
if (Display.getInstance().isDebuggableBuild()) {
    MCP.startSocketServer(8765);
}
```

A build that ships to devices you control, such as a kiosk fleet, a test lab, or a managed enterprise deployment, can lift the block:

```
MCP.setAllowOnReleaseBuilds(true);
MCP.startSocketServer(8765);
```

Treat that call the way you'd treat any other decision to expose a control surface, because the device itself becomes the boundary.

Chapter 24. Built-in user interface tools

Every server registers a small set of tools that read and drive the screen through the accessibility tree. An agent calls `ui_snapshot` to get the semantics tree of the current form as JSON, including the identifier, role, label, value, and state of each node, and the identifiers of the actions each node supports.

The agent drives the screen with `ui_perform_action`, which performs an action such as `activate` or `setText` on a node from the snapshot. The convenience tools `ui_activate` and `ui_set_text` wrap the common cases, and `ui_find` locates nodes by identifier, by label, or by screen coordinate.

Actions run on the Codename One EDT, so an agent never touches the live component tree directly. Each action returns whether it succeeded together with a fresh snapshot.

Chapter 25. Publishing application tools

An application publishes its own data and actions as MCP tools. A `Tool` has a name, a description, a JSON schema for its parameters, and a handler:

```
MCP.addTool(new Tool(
        "current_user",
        "Returns the signed in user",
        "{\"type\":\"object\",\"properties\":{}}",
        new ToolHandler() {
            @Override
            public String invoke(String argumentsJson) {
                return "{\"name\":\"" + signedInUser + "\"}";
            }
        }));
```

The server merges these tools with the built-in tools when a host lists the available tools, and routes each call to the matching handler. The same `Tool` type is used by the Codename One AI client, so a tool defined once serves both a hosted agent and an in-application model.

Chapter 26. Screenshots and the simulator

The server exposes a screenshot of the current form as an MCP image resource, so a vision capable model can see the screen alongside the semantic tree. The resource is enabled by default and can be disabled per server.

The JavaSE simulator drives the same MCP menu described above, so an application can be exposed and driven from the running simulator during development.

Chapter 27. Debug logging

Because an agent drives the tool on its own, it helps to watch what it does. The server echoes the MCP conversation to the Codename One log at a level chosen through the menu's **Debug Logging** item or in code:

```
MCP.setVerbosity(MCPVerbosity.FULL);
```

The levels of `MCPVerbosity` are, from quietest to loudest: `OFF`, `ERRORS` (only failed calls), `SUMMARY` (one line per call), and `FULL` (every request and response). Each level includes the one below it.

Chapter 28. Registering with installed hosts

The Install and Remove items in the MCP menu call `MCPClientRegistrar`, which detects the MCP hosts installed on the machine and writes a server entry into each host configuration, so an end user doesn't edit configuration by hand. Detection and registration are available to any Codename One tool, and they run inside the runtime because they use the portable `FileSystemStorage`.

A caller describes the server with an `MCPClientDescriptor` (a name and the command that launches it) and registers it with the detected hosts. Hosts whose configuration format isn't yet supported are reported so the user can add the entry manually.

Media and Audio

Chapter 29. MediaManager and MediaPlayer



`MediaPlayer` is a **peer component**, understanding this is crucial if your application depends on such a component. You can learn about peer components and their issues here [https://www.codenameone.com/manual/advanced-topics.html#native-peer-components].

The MediaPlayer [https://www.codenameone.com/javadoc/com/codename1/components/MediaPlayer.html] allows you to control video playback. To use the `MediaPlayer` you need to first load the `Media` object from the MediaManager [https://www.codenameone.com/javadoc/com/codename1/media/MediaManager.html].

The `MediaManager` is the core class responsible for media interaction in Codename One.



You should also check out the Capture [https://www.codenameone.com/javadoc/com/codename1/capture/Capture.html] class for things that aren't covered by the `MediaManager`.

In the demo code below you use the gallery functionality to pick a video from the device's video gallery:

```
final Form hi = new Form("MediaPlayer", new BorderLayout());
hi.setToolbar(new Toolbar());
Style s = UIManager.getInstance().getComponentStyle("Title");
FontImage icon = FontImage.createMaterial(FontImage.MATERIAL_VIDEO_LIBRARY, s);
hi.getToolbar().addCommandToRightBar("", icon, (evt) -> {
    Display.getInstance().openGallery((e) -> {
        if(e != null && e.getSource() != null) {
            String file = (String)e.getSource();
            try {
                Media video = MediaManager.createMedia(file, true);
                hi.removeAll();
                hi.add(BorderLayout.CENTER, new MediaPlayer(video));
                hi.revalidate();
            } catch(IOException err) {
                Log.e(err);
            }
        }
    }, Display.GALLERY_VIDEO);
});
hi.show();
```



Figure 220. Video playback running on the simulator

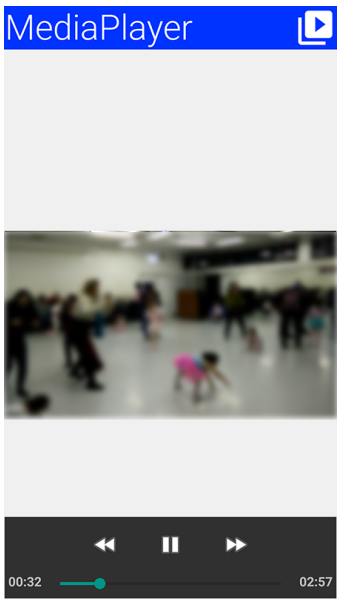


Figure 221. Video playback running on an Android device. Notice the native playback controls that appear when the video is tapped



Video playback in the simulator will work with JavaFX enabled. This is the default for Java 8 or newer so recommend using that.

Chapter 30. PCM Audio Buffers, Mixing, and Effects

`AudioBuffer` stores raw interleaved float PCM samples in the `[-1, 1]` range, and `WAVWriter` writes those samples to a WAV file.

The portable live PCM source in Codename One is microphone capture: create an `AudioBuffer` with `MediaManager.getAudioBuffer(…)`, then create a recorder with `MediaRecorderBuilder.redirectToAudioBuffer(true)`. The builder's `path` value is the buffer key in this mode; it isn't a file path.

```
String bufferKey = "voice-buffer";
AudioBuffer livePcm = MediaManager.getAudioBuffer(bufferKey, true, 8192);
livePcm.addCallback(buffer -> {
    float[] frame = new float[buffer.getSize()];
    buffer.copyTo(frame);
    // Process or copy this frame before the next callback arrives.
});

Media recorder = MediaManager.createMediaRecorder(
        new MediaRecorderBuilder()
                .path(bufferKey)
                .samplingRate(44100)
                .audioChannels(1)
                .redirectToAudioBuffer(true));
recorder.play();

// Later:
recorder.cleanup();
MediaManager.releaseAudioBuffer(bufferKey);
```

When the `VideoIO` API is available, it can also produce PCM from a prerecorded video file. `VideoIO.getVideoIO().openReader(path).readAudio()` returns the decoded audio track as an `AudioBuffer`, so video audio can be mixed or processed with the same APIs shown below. `VideoIO` support is platform-gated with `VideoIO.isSupported()`, and JavaScript currently reports no decoded audio track even though frame decoding and encoding are supported.

Codename One still doesn't expose a standalone portable decoder that converts an arbitrary prerecorded audio-only file, such as MP3 or AAC, into an `AudioBuffer` without going through a video container API. Use the normal `Media` APIs to play those files. If you need offline PCM editing for audio-only formats, decode them before passing samples to Codename One, or add a platform decoder API for that format.

When you need to combine PCM clips before writing or playing them, use `AudioMixer`. It creates a sample-accurate timeline on one sample clock. Each track must use the same sample rate and channel count as the mixer. Track offsets are supplied in milliseconds and are rounded to the nearest sample frame on the mixer clock; if you already know the exact frame position, use

`addTrackAtFrame(…)`.

The mixed buffer is clipped to the normalized PCM range `[-1, 1]`, so it can be passed directly to `WAVWriter`. The mixer doesn't resample or convert channel layouts; downsample or otherwise convert source buffers before adding them if they're not already on the target clock.

`AudioEffects` provides small platform-neutral PCM transforms that compose with the mixer:

`equalize(…)` is a simple three-band tone shaper for lightweight cleanup and demos. `removeCenter(…)`, `isolateCenter(…)`, and `midSide(…)` use classic stereo mid/side processing. They can reduce, isolate, or enhance centered dialog only when the source is stereo and the voice is actually centered. These methods aren't source-separation, speech-isolation, or noise-reduction APIs.

`AudioMixer` and the simple buffer-wide `AudioEffects` operations use the `com.codename1.util.Simd` float-array API internally when the current platform supports it. Stateful filters, such as the equalizer, stay scalar because each sample depends on the previous filter state. You can force scalar behavior for diagnostics with `AudioMixer.setSimdOptimizationsEnabled(false)` or `AudioEffects.setSimdOptimizationsEnabled(false)`.

Chapter 31. Animations

There are many ways to animate and liven the data within a Codename One application, layout animations are probably chief among them. But first you need to understand some basics such as layout reflows.

31.1. Layout reflow

Layout in tools such as HTML is implicit, when you add something into the UI it's automatically placed. Other tools such as Codename One use explicit layout, that means you've to explicitly request the UI to layout itself after making changes!



Like many such rules exceptions occur. For example, if the device is rotated or window size changes a layout will occur automatically.

When adding a component to a UI that's already visible, the component won't show by default.



When adding a component to a form which isn't shown on the screen, there is no need to tell the UI to repaint or reflow. This happens implicitly.

The chief advantage of explicit layout is performance.

For example, imagine adding 100 components to a form. If the form was laid out automatically, layout would have happened 100 times instead of once when adding was finished. In fact layout reflows are often considered the #1 performance issue for HTML/JavaScript applications.



Smart layout reflow logic can alleviate some pains of the automatic layout reflows but since the process is implicit it's almost impossible to optimize complex usages across browsers/devices. A major JavaScript performance tip is to use absolute positioning which is akin to not using layouts at all!

That's why, when you add components to a form that's already showing, you should invoke `revalidate()` or animate the layout appropriately. This also enables the layout animation behavior explained below.

31.2. Layout animations

To understand animations you need to understand a couple of things about Codename One components. When you add a component to a container, it's added but not positioned anywhere. A novice might notice the `setX`/`setY`/`setWidth`/`setHeight` methods on a component and try to position it.

This won't work since these methods are meant for the layout manager, which is implicitly invoked when a form is shown (internally in Codename One). The layout manager uses these methods to position/size the components based on the hints given to it.

If you add components to a form that's showing, it's your responsibility to invoke `revalidate` /`layoutContainer` to arrange the newly added components (see Layout Reflows).

`animateLayout()` method is a fancy form of revalidate that animates the components into their laid out position. After changing the layout & invoking this method the components move to their new sizes/positions seamlessly. `Form` exposes convenience wrappers such as `animateLayout*()` that forward to the underlying content pane, so you can call the methods directly on the form unless you specifically need to animate a nested container.

This sort of behavior creates a special case where setting the size/position makes sense. When you set the size/position in the demo code here you're positioning the components at the animation start position above the frame:

```
Form hi = new Form("Layout Animations", new BoxLayout(BoxLayout.Y_AXIS));
Button fall = new Button("Fall"); ①
fall.addActionListener((e) -> {
    for (int iter = 0; iter < 10; iter++) {
        Label b = new Label("Label " + iter);
        b.setWidth(fall.getWidth());
        b.setHeight(fall.getHeight());
        b.setY(-fall.getHeight());
        hi.add(b);
    }
    hi.animateLayout(20000); ②
});
hi.add(fall);
```

Notice these things about this example:

① You used a button to do the animation rather than doing it on show. Since `show()` implicitly lays out the components it wouldn't have worked.

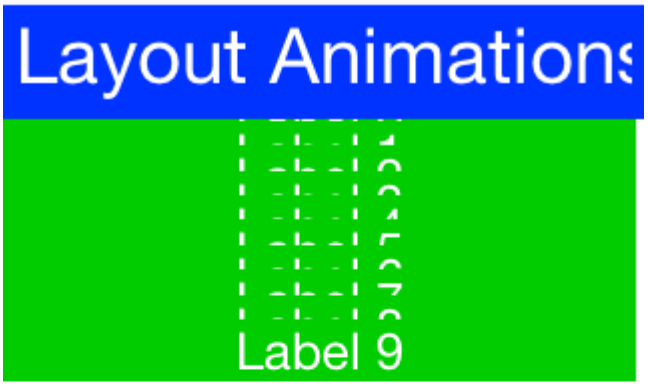
② You used `hi.animateLayout(20000);`, which delegates to the `Form` content pane. If you need to animate a specific container (for example, a nested layout), call `animateLayout()` on that container instead.

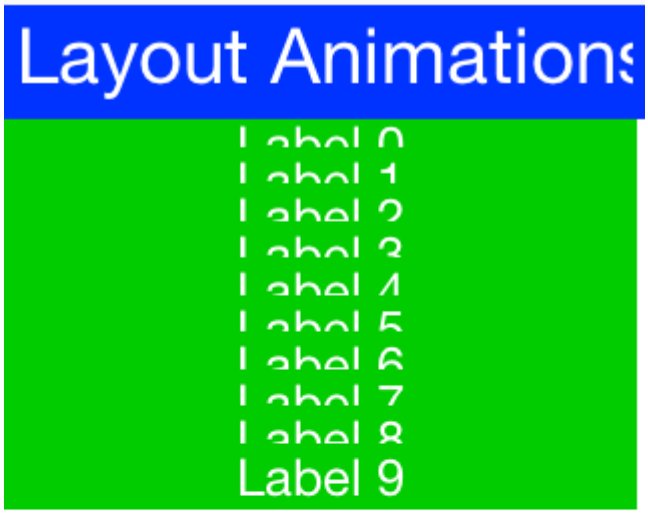
This results in:

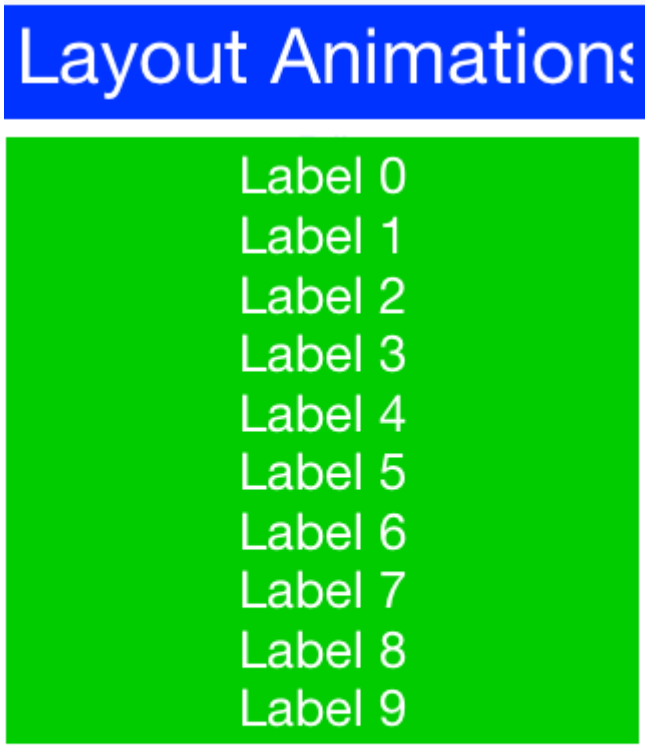

Layout Animations

Fall


Layout Animations
Label 9


Layout Animations
Label 9


Layout Animations
Label 0
Label 1
Label 2
Label 3
Label 4
Label 5
Label 6
Label 7
Label 8
Label 9


Layout Animations
Label 0
Label 1
Label 2
Label 3
Label 4
Label 5
Label 6
Label 7
Label 8
Label 9


Layout Animations
Fall
Label 0
Label 1
Label 2
Label 3
Label 4
Label 5
Label 6
Label 7
Label 8
Label 9


Layout Animations
Fall
Label 0
Label 1
Label 2
Label 3
Label 4
Label 5
Label 6
Label 7

31.2.1. Unlayout animations

While layout animations are powerful effects for adding elements into the UI and drawing attention to them. The inverse of removing an element from the UI is often more important. For example, when you delete or remove an element you want to animate it out.

Layout animations don't do that since they will try to bring the animated item into place. The exact opposite of a layout animation is the "unlayout animation." `Container.animateUnlayout(int, int, Runnable)` and the `Form.animateUnlayout*()` helpers trigger this transition either asynchronously (with a completion callback) or synchronously via the `AndWait` variants.

The "unlayout animation" takes a valid laid out state and shifts the components to an invalid state that you defined in advance. For example, you can fix the example above to flip the "fall" button into a "rise" button when the buttons come into place and this will allow the buttons to float back up to where they came from in the exact reverse order.



An unlayout animation always leaves the container in an invalidated state.

```
Form hi = new Form("Layout Animations", new BoxLayout(BoxLayout.Y_AXIS));
Button fall = new Button("Fall");
fall.addActionListener((e) -> {
    if (hi.getContentPane().getComponentCount() == 1) {
        fall.setText("Rise");
        for (int iter = 0; iter < 10; iter++) {
            Label b = new Label("Label " + iter);
            b.setWidth(fall.getWidth());
            b.setHeight(fall.getHeight());
            b.setY(-fall.getHeight());
            hi.add(b);
        }
        hi.animateLayout(20000);
    } else {
        fall.setText("Fall");
        for (int iter = 1; iter < hi.getContentPane().getComponentCount(); iter++) { ①
            Component c = hi.getContentPane().getComponentAt(iter);
            c.setY(-fall.getHeight()); ②
        }
        hi.animateUnlayoutAndWait(20000, 255); ③
        hi.removeAll(); ④
        hi.add(fall);
        hi.revalidate();
    }
});
hi.add(fall);
```

You will notice some similarities with the unlayout animation but the differences represent the exact opposite of the layout animation:

① You loop over existing components (not newly created ones)

② You set the desired end position not the desired starting position

③ You used the `animateUnlayoutAndWait(…)` variant to block until completion; `animateUnlayout(duration, opacity, callback)` provides the non-blocking alternative when you want to continue and run code from a callback instead.

④ After the animation completes you need to actually remove the elements since the UI is now in an invalid position with elements outside of the screen but still physically there!

31.2.2. Hiding & visibility

A common trick for animating Components in Codename One is to set their preferred size to `0` and then invoke `animateLayout()` thus triggering an animation to hide said `Component`. several issues with this trick but one of the biggest ones is the fact that `setPreferredSize` has been deprecated for quite a while.

Instead of using that trick you can use `setHidden/isHidden` who effectively encapsulate this functionality and a bit more.

One of the issues `setHidden` tries to solve is the fact that preferred size doesn't include the margin in the total and thus a component might still occupy space despite being hidden. When you request the margin change the current margins are cached, the component is given zero margins while hidden, and those cached values are restored when it's shown again—without resetting the UIID or other style state.

This functionality might be undesirable which is why there is a version of the `setHidden` method that accepts a boolean flag indicating whether the margin cache should be manipulated. You can effectively `hide/show` a component without deprecated code using something like this:

```
Button toHide = new Button("Will Be Hidden");
Button hide = new Button("Hide It");
hide.addActionListener((e) -> {
    hide.setEnabled(false);
    boolean t = !toHide.isHidden();
    toHide.setHidden(t);
    toHide.getParent().animateLayoutAndWait(200);
    toHide.setVisible(!t);
    hide.setEnabled(true);
});
```



Notice that the code above uses `setVisible()`, which shouldn't be confused with `setHidden`. `setVisible()` toggles the visibility of the component it would still occupy the same amount of space

31.2.3. Synchronicity in animations

Most animations have two or three variants:

- Standard animation for example, `animateLayout(int)` or the non-blocking `animateUnlayout(int,`

`int, Runnable)`

- And wait variant for example, `animateLayoutAndWait(int)` / `animateUnlayoutAndWait(int, int)`
- Callback variant for example, `animateLayoutFade(int, int, Runnable)`

The standard animation is invoked when you don't care about the completion of the animation. You can do this for a standard animation.



Unlayout animations always leave the container in an invalid state, so even the standard helper expects you to tidy up either in the callback or after the animation finishes.

The `AndWait` variant blocks the calling thread until the animation completes. This is useful for sequencing animations one after the other e.g this code from the kitchen sink demo:

```
arrangeForInterlace(effects);
effects.animateUnlayoutAndWait(800, 20);
effects.animateLayoutFade(800, 20);
```

First the UI goes through an "unlayout" animation, once that completes the layout itself is performed.



The `AndWait` calls needs to be invoked on the Event Dispatch Thread despite being "blocking." This is a common convention in Codename One powered by a unique capability of Codename One: `invokeAndBlock`.
You can learn more about `invokeAndBlock` in the EDT section [https://www.codenameone.com/manual/edt.html].

The callback variant is like the `invokeAndBlock` variant but uses a more conventional callback semantic which is more familiar to some developers. It accepts a `Runnable` callback that will be invoked after the fact. For example, you can change the unlayout call from before to use the callback semantics as such:

```
hi.animateUnlayout(20000, 255, () -> {
    hi.removeAll();
    hi.add(fall);
    hi.revalidate();
});
```

Animation fade and hierarchy

There are several more variations on the standard animate methods. Several methods accept a numeric `fade` argument. This is useful to fade out an element in an "unlayout" operation or fade in a regular animation.

The value for the fade argument is a number between 0 and 255 where 0 represents full transparency and 255 represents full opacity.

Some animate layout methods are hierarchy based. They work like the regular `animateLayout` methods but recurse into the entire Container [https://www.codenameone.com/javadoc/com/codename1/ui/Container.html] hierarchy. These methods work well when you have components in a nested hierarchy that need to animate into place. This is demonstrated in the opening sequence of the kitchen sink demo:

```
for(int iter = 0 ; iter < demoComponents.size() ; iter++) {
    Component cmp = (Component)demoComponents.elementAt(iter);
    if(iter < componentsPerRow) {
        cmp.setX(-cmp.getWidth());
    } else {
        if(iter < componentsPerRow * 2) {
            cmp.setX(dw);
        } else {
            cmp.setX(-cmp.getWidth());
        }
    }
}
boxContainer.setShouldCalcPreferredSize(true);
boxContainer.animateHierarchyFade(3000, 30);
```

The `demoComponents Vector` contains components from separate containers and this code would not work with a simple animate layout.



Avoid the hierarchy version where possible. Its slower but more importantly, it's flaky. Since the size/position of the `Container` might be affected by the layout the animation could get clipped and skip. These are hard issues to debug.

31.2.4. Sequencing animations via AnimationManager

All the animations go through a per-form queue: the AnimationManager [https://www.codenameone.com/javadoc/com/codename1/ui/AnimationManager.html]. This effectively prevents two animations from mutating the UI in parallel so you won't have collisions between two conflicting sides. Things get more interesting when you try to do something like this:

```
cnt.add(myButton);
int componentCount = cnt.getComponentCount();
cnt.animateLayout(300);
cnt.removeComponent(myButton);
if(componentCount == cnt.getComponentCount()) {
    // this will happen...
}
```

The reason this happens is that the second remove gets postponed to the end of the animation so it won't break the animation. This works for remove and add operations on a Container [https://www.codenameone.com/javadoc/com/codename1/ui/Container.html] as well as other animations.

The simple **yet problematic** fix would be:

```
cnt.add(myButton);
int componentCount = cnt.getComponentCount();
cnt.animateLayoutAndWait(300);
cnt.removeComponent(myButton);
if(componentCount == cnt.getComponentCount()) {
    // this probably won't happen...
}
```

Why might that still fail?

Events come in constantly during the run of the EDT [1], so an event might come in that might trigger an animation in your code. Even if you're on the EDT keep in mind that you don't actually block it and an event might come in.

In those cases an animation might start and you might be unaware of that animation and it might still be in action when you expect remove to work.

Animation manager to the rescue

AnimationManager [https://www.codenameone.com/javadoc/com/codename1/ui/AnimationManager.html] has built-in support to fix this exact issue.

You can flush the animation queue and run synchronously after all the animations finished and before new ones come in by using something like this:

```
cnt.add(myButton);
int componentCount = cnt.getComponentCount();
cnt.animateLayout(300);
cnt.getAnimationManager().flushAnimation(() -> {
    cnt.removeComponent(myButton);
    if(componentCount == cnt.getComponentCount()) {
        // this shouldn't happen...
    }
});
```

31.3. Low-level animations

The Codename One event dispatch thread has a special animation "pulse" allowing an animation to update its state and draw itself. Code can make use of this pulse to implement repetitive polling tasks that have little to do with drawing.

This is helpful since the callback will always occur on the event dispatch thread.

Every component in Codename One contains an `animate()` method that returns a boolean value, you can also implement the Animation [https://www.codenameone.com/javadoc/com/codename1/ui/animations/Animation.html] interface in an arbitrary component to implement your own animation. To

receive animation events you need to register yourself within the parent form, it's the responsibility of the parent for to call `animate()`.

If the `animate` method returns true then the animation will be painted (the `paint` method of the `Animation` interface would be invoked).



It's important to deregister animations when they aren't needed to conserve battery life.

If you derive from a component, which has its own animation logic you might damage its animation behavior by deregistering it, so use care with the low-level API's.

For example, you can add more animation logic using code like this:

```
myForm.registerAnimated(this);
```

```
private int spinValue;
@Override
public boolean animate() {
   if(userStatusPending) {
       spinValue++;
       super.animate();
       return true;
   }
   return super.animate();
}
```

31.3.1. Why not just write code in paint

Animations comprise two parts, the logic (deciding the position, etc.) and the painting. The paint method should be dedicated to painting, not to the actual moving of the components.

The separation of concerns allows you to avoid redundant painting for example, if animate didn't trigger a change return `false` to avoid the overhead related to animations.

You discuss low-level animations in more details within the animation section of the clock demo [https://www.codenameone.com/manual/graphics.html#clock-animation-section].

31.4. Transitions

Transitions allow you to replace one component with another, most typically forms or dialogs are replaced with a transition but a transition can be applied to replace any arbitrary component.

Developers can implement their own custom transition and install it to components by deriving the Transition [https://www.codenameone.com/javadoc/com/codename1/ui/animations/Transition.html] class, although most commonly the built in CommonTransitions [https://www.codenameone.com/javadoc/com/codename1/ui/animations/CommonTransitions.html] class is used for almost everything.

You can define transitions for forms/dialogs/menus globally either via the theme constants or via the LookAndFeel [https://www.codenameone.com/javadoc/com/codename1/ui/plaf/LookAndFeel.html] class. Or you can install a transition on top-level components via setter methods.

In/Out Transitions

When defining a transition you define the entering transition and the exiting transition. For most cases only one of those is necessary and you default to the exiting (out transition) as a convention.

For most cases the method `setFormTransitonIn` should go unused. That API exists for some elaborate custom transitions that might need to have a special effect both when transitioning in and out of a specific form. But, most of these effects are easier to achieve with layout animations (for example, components dropping into place etc.).

For `Dialog` the transition in shows its appearance and the transition out shows its disposal. In that case both transitions make a lot of sense.

Back/Forward Transitions

Transitions have a direction and can all be played either in incoming or outgoing direction. A transition can be flipped (played in reverse) when use an RTL language [2] or when you traverse backwards in the form navigation hierarchy.

Normally `Form.show()` displays the next `Form` with an incoming transition based on the current RTL mode. If you use `Form.showBack()` it will play the transition in reverse.



When working with high-level animations you can select Slow Motion option in the simulator to slow down animations and inspect their details

Themes define the default transitions used when showing a form, these differ based on the OS. In most platforms the default is `Slide` whereas in iOS the default is `SlideFade` which slides the content pane and title while fading in/out the content of the title area.



`SlideFade` is problematic without a title area. If you have a `Form` that lacks a title area you would recommend to disable `SlideFade` at least for that `Form`.

Check out the full set of theme constants in the Theme Constants Section [https://www.codenameone.com/manual/advanced-theming.html#theme-constants-section].

31.4.1. Replace

To apply a transition to a component you can use the `Container.replace()` method as such:

```
Form hi = new Form("Replace", new BoxLayout(BoxLayout.Y_AXIS));
Button replace = new Button("Replace Pending");
Label replaceDestiny = new Label("Destination Replace");
```

```
hi.add(replace);
replace.addActionListener((e) -> {
    replace.getParent().replaceAndWait(replace, replaceDestiny, CommonTransitions.createCover(CommonTransitions
.SLIDE_VERTICAL, true, 800));
    replaceDestiny.getParent().replaceAndWait(replaceDestiny, replace, CommonTransitions.createUncover(CommonTransitions
.SLIDE_VERTICAL, true, 800));
});
```



Replace even works when you have a layout constraint in place for example, replacing a component in a border layout will do the "right thing." But, some layouts such as `TableLayout` might be tricky in such cases so recommend wrapping a replaceable `Component` in a border layout and replacing the content.

`Container.replace()` can also be used with a null transition at which point it replaces instantly with no transition.

31.4.2. Slide transitions

The slide transitions are used to move the `Form`/`Component` in a sliding motion to the side or up/down. Four basic types of slide transitions exist:

1. Slide - the most commonly used transition
2. Fast Slide - historically this provided better performance for old device types. It's no longer recommended for newer devices
3. Slide Fade - the iOS default where the title area features a fade transition
4. Cover/Uncover - a kind of slide transition where the source or destination form slides while the other remains static in place

The code below demonstrates the usage of all the main transitions:

```
    Toolbar.setGlobalToolbar(true);
    Form hi = new Form("Transitions", new BoxLayout(BoxLayout.Y_AXIS));
    Style bg = hi.getContentPane().getUnselectedStyle();
    bg.setBgTransparency(255);
    bg.setBgColor(0xff0000);
    Button showTransition = new Button("Show");
    Picker pick = new Picker();
    pick.setStrings("Slide", "SlideFade", "Cover", "Uncover", "Fade", "Flip");
    pick.setSelectedString("Slide");
    TextField duration = new TextField("10000", "Duration", 6, TextArea.NUMERIC);
    CheckBox horizontal = CheckBox.createToggle("Horizontal");
    pick.addActionListener((e) -> {
        String s = pick.getSelectedString().toLowerCase();
        horizontal.setEnabled(s.equals("slide") || s.indexOf("cover") > -1);
    });
    horizontal.setSelected(true);
    hi.add(showTransition).
        add(pick).
        add(duration).
        add(horizontal);

    Form dest = new Form("Destination");
    bg = dest.getContentPane().getUnselectedStyle();
    bg.setBgTransparency(255);
```

```
    bg.setBgColor(0xff);
    dest.setBackCommand(
            dest.getToolbar().addCommandToLeftBar("Back", null, (e) -> hi.showBack()));

    showTransition.addActionListener((e) -> {
        int h = CommonTransitions.SLIDE_HORIZONTAL;
        if(!horizontal.isSelected()) {
            h = CommonTransitions.SLIDE_VERTICAL;
        }
        switch(pick.getSelectedString()) {
            case "Slide":
                hi.setTransitionOutAnimator(CommonTransitions.createSlide(h, true, duration.getAsInt(3000)));
                dest.setTransitionOutAnimator(CommonTransitions.createSlide(h, true, duration.getAsInt(3000)));
                break;
            case "SlideFade":
                hi.setTransitionOutAnimator(CommonTransitions.createSlideFadeTitle(true, duration.getAsInt(3000)));
                dest.setTransitionOutAnimator(CommonTransitions.createSlideFadeTitle(true, duration.getAsInt(3000)));
                break;
            case "Cover":
                hi.setTransitionOutAnimator(CommonTransitions.createCover(h, true, duration.getAsInt(3000)));
                dest.setTransitionOutAnimator(CommonTransitions.createCover(h, true, duration.getAsInt(3000)));
                break;
            case "Uncover":
                hi.setTransitionOutAnimator(CommonTransitions.createUncover(h, true, duration.getAsInt(3000)));
                dest.setTransitionOutAnimator(CommonTransitions.createUncover(h, true, duration.getAsInt(3000)));
                break;
            case "Fade":
                hi.setTransitionOutAnimator(CommonTransitions.createFade(duration.getAsInt(3000)));
                dest.setTransitionOutAnimator(CommonTransitions.createFade(duration.getAsInt(3000)));
                break;
            case "Flip":
                hi.setTransitionOutAnimator(new FlipTransition(-1, duration.getAsInt(3000)));
                dest.setTransitionOutAnimator(new FlipTransition(-1, duration.getAsInt(3000)));
                break;
        }
        dest.show();
    });
    hi.show();
```

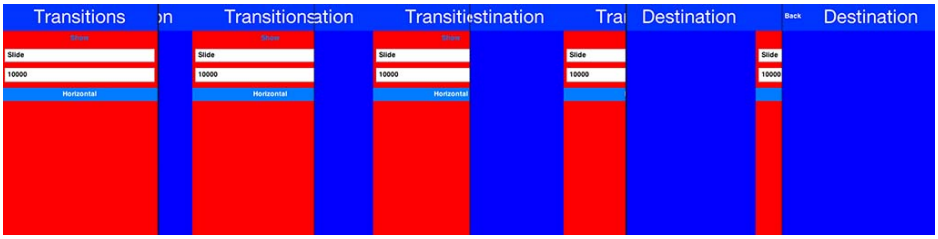


Figure 222. The slide transition moves both incoming and outgoing forms together

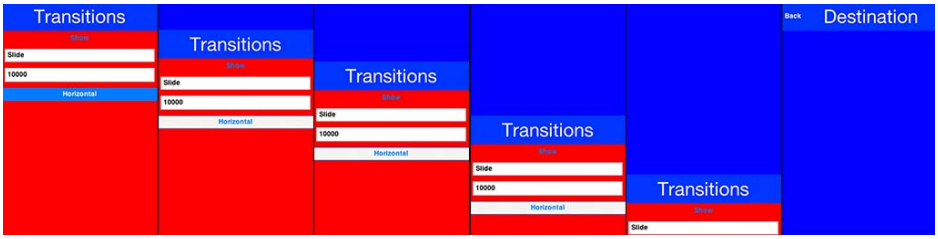


Figure 223. The slide transition can be applied vertically as well



Figure 224. Slide fade fades in the destination title while sliding the content pane it's the default on iOS



`SlideFade` is problematic without a title area. If you have a `Form` that lacks a title area you would recommend to disable `SlideFade` at least for that `Form`.

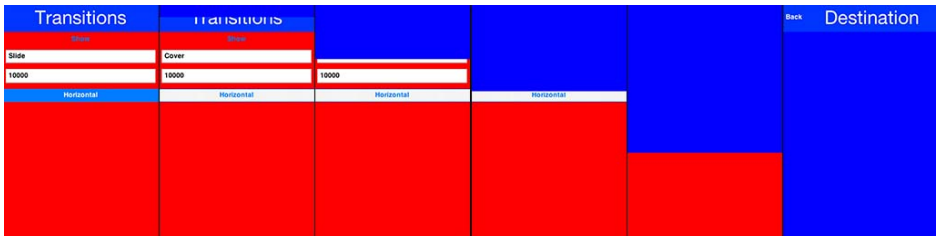


Figure 225. With cover transitions the source form stays in place as it's covered by the destination. This transition can be played both horizontally and vertically

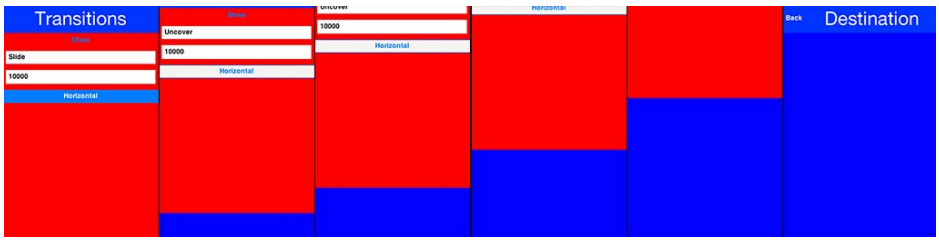


Figure 226. Uncover is the inverse of cover. The destination form stays in place while the departing form moves away

31.4.3. Fade and flip transitions

The fade transition is pretty trivial and accepts a time value since it has no directional context.

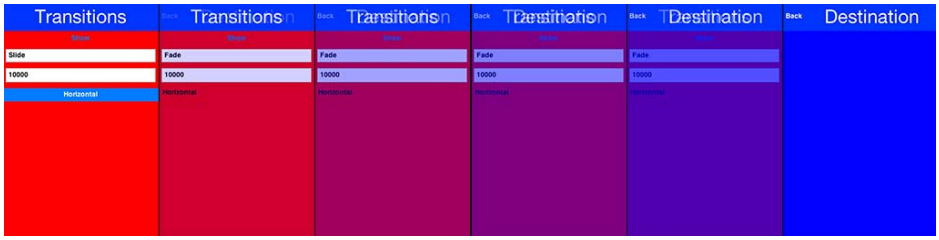


Figure 227. Fade transition is probably the simplest one around

The FlipTransition [https://www.codenameone.com/javadoc/com/codename1/ui/animations/FlipTransition.html] is also pretty simple but unlike the others it isn't a part of the `CommonTransitions`. It has its own `FlipTransition` class.



This transition looks different on devices as it uses native perspective transforms available there

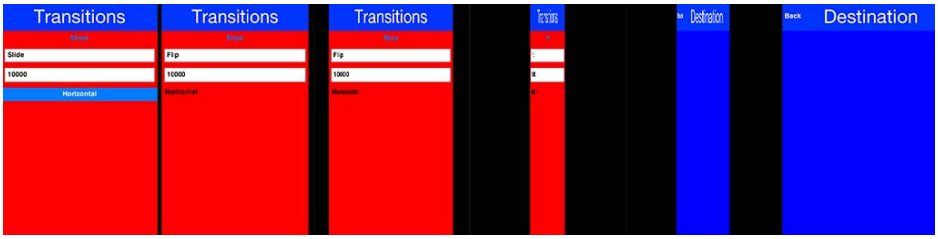


Figure 228. Fade transition is probably the simplest one around

31.4.4. Bubble transition

BubbleTransiton [https://www.codenameone.com/javadoc/com/codename1/ui/animations/BubbleTransition.html] morphs a component into another component using a circular growth motion.

The `BubbleTransition` accepts the component that will grow into the bubble effect as one of its arguments. It's primarily designed for `Dialog` transitions although it could work for more creative use cases:



The code below manipulates styles and look. This is done to make the code more "self contained." Real world code should probably use the theme

```
Form hi = new Form("Bubble");
Button showBubble = new Button("+");
showBubble.setName("BubbleButton");
Style buttonStyle = showBubble.getAllStyles();
buttonStyle.setBorder(Border.createEmpty());
buttonStyle.setFgColor(0xffffff);
buttonStyle.setBgPainter((g, rect) -> {
    g.setColor(0xff);
    int actualWidth = rect.getWidth();
    int actualHeight = rect.getHeight();
    int xPos, yPos;
    int size;
    if(actualWidth > actualHeight) {
        yPos = rect.getY();
        xPos = rect.getX() + (actualWidth - actualHeight) / 2;
        size = actualHeight;
    } else {
        yPos = rect.getY() + (actualHeight - actualWidth) / 2;
        xPos = rect.getX();
        size = actualWidth;
    }
    g.setAntiAliased(true);
    g.fillArc(xPos, yPos, size, size, 0, 360);
});
hi.add(showBubble);
hi.setTintColor(0);
showBubble.addActionListener((e) -> {
    Dialog dlg = new Dialog("Bubbled");
    dlg.setLayout(new BorderLayout());
    SpanLabel sl = new SpanLabel("This dialog should appear with a bubble transition from the button", "DialogBody");
    sl.getTextUnselectedStyle().setFgColor(0xffffff);
    dlg.add(BorderLayout.CENTER, sl);
    dlg.setTransitionInAnimator(new BubbleTransition(500, "BubbleButton"));
    dlg.setTransitionOutAnimator(new BubbleTransition(500, "BubbleButton"));
    dlg.setDisposeWhenPointerOutOfBounds(true);
    dlg.getTitleStyle().setFgColor(0xffffff);

    Style dlgStyle = dlg.getDialogStyle();
    dlgStyle.setBorder(Border.createEmpty());
```

```
        dlgStyle.setBgColor(0xff);
        dlgStyle.setBgTransparency(0xff);
        dlg.showPacked(BorderLayout.NORTH, true);
    });

    hi.show();
```



Figure 229. Bubble transition converting a circular button to a Dialog

31.4.5. Morph transitions

Android's material design has a morphing effect where an element from the previous form (activity) animates into a different component on a new activity. Codename One has a morph effect in the Container [https://www.codenameone.com/javadoc/com/codename1/ui/Container.html] class but it doesn't work as a transition between forms and doesn't allow for multiple separate components to transition at once.



Figure 230. Morph Transition

To support this behavior you have the MorphTransition [https://www.codenameone.com/javadoc/com/codename1/ui/animations/MorphTransition.html] class that provides this same effect coupled with a fade to the rest of the UI (see Figure 230, "Morph Transition").

Since the transition is created before the form exists you can't reference explicit components within the form when creating the morph transition (to show which component becomes which) so you need to refer to them by name. This means you need to use `setName(String)` on the components in the source/destination forms so the transition will be able to find them:

```
Form demoForm = new Form(currentDemo.getDisplayName());
demoForm.setScrollable(false);
demoForm.setLayout(new BorderLayout());
Label demoLabel = new Label(currentDemo.getDisplayName());
demoLabel.setIcon(currentDemo.getDemoIcon());
demoLabel.setName("DemoLabel");
demoForm.addComponent(BorderLayout.NORTH, demoLabel);
demoForm.addComponent(BorderLayout.CENTER, wrapInShelves(n));
```

```
// ...
demoForm.setBackCommand(backCommand);
demoForm.setTransitionOutAnimator(
    MorphTransition.create(3000).morph(
        currentDemo.getDisplayName(),
        "DemoLabel"));
f.setTransitionOutAnimator(
    MorphTransition.create(3000).
        morph(currentDemo.getDisplayName(),
        "DemoLabel"));
demoForm.show();
```

Snapshot mode

By default `MorphTransition` paints both endpoints by re-rendering the live source / destination components every frame at the interpolated bounds. That works well when the source is fully visible — a card in the body of one form morphing to a card in the next.

It runs into edge cases when the source lives inside a scrolling container that has children extending past the source's bounds, or when the source has dynamic content (a video frame, a `BrowserComponent`, a custom-painted background) that should be visually frozen for the duration of the animation. The legacy live-paint path can leak off-viewport pixels into the morph because the layered pane that holds the source during the animation doesn't carry the original parent's clip.

To opt into the image-snapshot path call `snapshotMode(true)` on the builder:

`snapshotMode(true)` captures each `(source, dest)` pair as a clipped `Image` at `initTransition()`, then the tween draws those images at the interpolated bounds rather than re-painting the live components. Off-viewport children of the source are clipped at capture time (the image's own bounds are the clip), so they can't leak into the morph.

The default `MorphTransition` behavior (live paint, no snapshots) is unchanged for back-compat. Use snapshot mode opportunistically when the live-paint output exhibits the off-viewport leak, or when the source's children produce frame-by-frame visual change you want to freeze.

31.4.6. SwipeBackSupport

iOS7+ allows swiping back one form to the previous form, Codename One has an API to enable back swipe transition:

```
SwipeBackSupport.bindBack(currentForm, destination);
```

That one command will enable swiping back from `currentForm`. LazyValue [https://www.codenameone.com/javadoc/com/codename1/util/LazyValue.html] allows you to pass a value lazily:

This effectively allows you to pass a form and create it as necessary (for example, for a GUI builder app you don't have the actual previous form instance), notice that the arguments aren't used for

this case but will be used in other cases.

The code below should work for the transition sample above. Notice that this API was designed to work with "Slide Fade" transition and might have issues with other transition types:

```
SwipeBackSupport.bindBack(dest, (args) -> hi);
```

[1] Event Dispatch Thread

[2] Right to left/bidi language such as Hebrew or Arabic

Chapter 32. The EDT - Event Dispatch Thread

32.1. What's The EDT

Codename One allows developers to create as many threads as they want; however to interact with the Codename One user interface components a developer must use the EDT. The EDT stands for "Event Dispatch Thread" but it handles a lot more than "events."

The EDT is the main thread of Codename One, by using one thread Codename One can avoid complex synchronization code and focus on simple functionality that assumes one thread.



This has huge advantages for your code. You can normally assume that all code will occur on a single thread and avoid complex synchronization logic.

You can visualize the EDT as a loop such as this:

Normally, every call you receive from Codename One will occur on the EDT. For example, every event, calls to paint(), and lifecycle calls (start etc.) should all occur on the EDT.

This is pretty powerful, however it means that as long as your code is processing nothing else can happen in Codename One!



If your code takes too long to execute then no painting or event processing will occur during that time, so a call to `Thread.sleep()` will actually stop everything!

The solution is pretty simple, if you need to perform something that requires intensive CPU you can spawn a thread.

Codename One's networking code automatically spawns its own network thread (see the NetworkManager [https://www.codenameone.com/javadoc/com/codename1/io/NetworkManager.html]). However, this also poses a problem...

Codename One assumes all modifications to the UI are performed on the EDT. If you spawn a separate thread, how do you force its modifications back into the EDT?

Codename One includes helper methods in the Display [https://www.codenameone.com/javadoc/com/codename1/ui/Display.html] class to help in these situations: `isEDT()`, `callSerially(Runnable)`, `callSeriallyOnIdle(Runnable)`, and `callSeriallyAndWait(Runnable)` (with an optional timeout overload).

`isEDT()` is useful for generic code that needs to test whether the current code is executing on the EDT.

32.2. Call Serially (And Wait)

`callSerially(Runnable)` should normally be called off the EDT (in a separate thread), the run method within the submitted runnable will be invoked on the EDT.



The Runnable passed to the `callSerially` and `callSeriallyAndWait` methods isn't a `Thread`. Use the `Runnable` interface as a convenient callback interface:



You can write this code more concisely using Java 8 lambda code as such:

This allows code to leave the EDT and then later on return to it to perform things within the EDT.

The `callSeriallyAndWait(Runnable)` method blocks the current thread until the method completes, this is useful for cases such as user notification e.g.:



If you're unsure use `callSerially`. The use cases for `callSeriallyAndWait` are rare. When you do need to wait, the overload that accepts a timeout lets background threads marshal results back to the EDT without risking an indefinite stall if the EDT is busy.

If the work you're posting back to the EDT is expensive and can wait until the UI finishes its current pass, use `callSeriallyOnIdle()`. This queues the runnable for execution after the EDT completes painting, animations, and any other queued tasks. Deferring longer tasks to the idle queue helps avoid starving animations and transitions when bursts of background callbacks arrive together.

32.2.1. callSerially On The EDT

One of the misunderstood topics is why you'd ever want to invoke `callSerially` when you're still on the EDT. This is best explained by example. Say you have a button that has quite a bit of functionality tied to its events e.g.:

1. A user added an action listener to show a Dialog.
2. A framework the user installed added some logging to the button.
3. The button repaints a release animation as its being released.

However, this might cause a problem if the first event being handled (the dialog) causes an issue with the following events. For example, a dialog will block the EDT (using `invokeAndBlock`), events will keep happening but since the event you're in "already happened" the button repaint and the framework logging won't occur. The same problem also occurs if you show a form which might trigger logic that relies on the current form still being present.

One of the solutions to this problem is to wrap the action listeners body with a `callSerially`. In this case the `callSerially` will postpone the event to the next cycle (loop) of the EDT and let the other events in the chain complete. Notice that you shouldn't use this normally since it includes an overhead and complicates application flow, however when you run into issues in event processing this is worth trying to see if its the cause.



You should never invoke callSeriallyAndWait on the EDT since this would effectively mean sleeping on the EDT. That method throws an exception if its invoked from the EDT.

If you need to run logic on the EDT but must ensure nothing inside it blocks, the `Display` class also

provides `invokeWithoutBlocking()` and `invokeWithoutBlockingWithResultSync()`. These helpers temporarily disable `invokeAndBlock()` during the runnable. Any nested call to `invokeAndBlock()` while blocking is disabled results in a `BlockingDisallowedException`. Framework code that wraps user callbacks can use these methods to guard against accidental nested blocking that would otherwise deadlock or stall the UI, yet still obtain a return value when needed via `invokeWithoutBlockingWithResultSync()`.

32.3. Debugging EDT Violations

Two types of EDT violations exist:

1. Blocking the EDT thread so the UI performance is considerably slower.
2. Invoking UI code on a separate thread

Codename One provides a tool to help you detect some of these violations some caveats may apply though...

It's an imperfect tool. It might fire "false positives" meaning it might detect a violation for legal code and it might miss some illegal calls. However, it's a valuable tool in the process of detecting hard to track bugs that are sometimes reproducible on the devices (due to race condition behavior).

To activate this tool select the Debug EDT menu option in the simulator and pick the level of output you wish to receive:

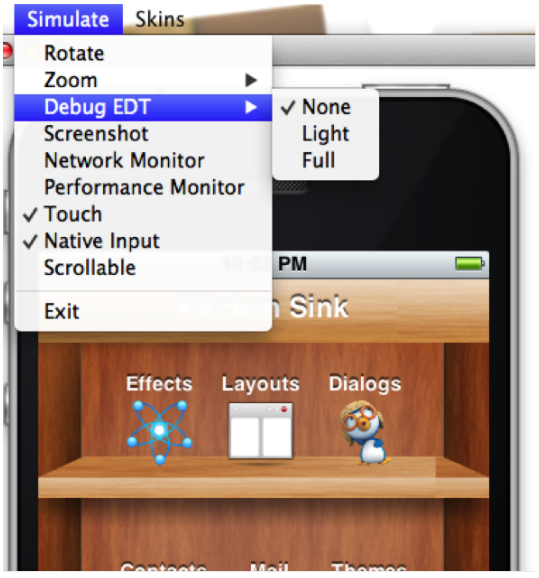


Figure 231. Debug EDT

Full output will include stack traces to the area in the code that's suspected in the violation.

32.4. Invoke And Block

Invoke and block is the exact opposite of `callSeriallyAndWait()`, it blocks the EDT and opens a separate thread for the runnable call. This functionality is inspired by the Foxtrot [http://foxtrot.sourceforge.net/] API, which is a remarkably powerful tool most Swing developers don't know about.

This is best explained by an example. Typical Java code reads as a single sequence as such:

This works well normally but on the EDT it might be a problem, if one of the operations is slow it might slow the whole EDT (painting, event processing etc.). Normally you can move operations into a separate thread e.g.:

This means that operation C will happen in parallel to operation B which might be a problem...
For example, instead of using operation names lets use a more "real world" example:

Notice that the first and last operations must be conducted on the EDT but the middle operation might be slow! Since `updateUIWithContentOfFile` needs `readAndParseFile` to occur before it starts doing the new thread won't be enough.

A simplistic approach is to do something like this:

However, `updateUIWithContentOfFile` should be executed on the EDT and not on a random thread. The right way to do this would therefore be something like this:

This is legal and would work reasonably well, however it gets complicated as you add more and more features that need to be chained serially after all these are 3 methods!

Invoke and block solves this in a unique way you can get almost the exact same behavior by using this:

Or this with Java 8 syntax:

Invoke and block effectively blocks the current EDT in a legal way. It spawns a separate thread that runs the `run()` method and when that run method completes it goes back to the EDT.

All events and EDT behavior still work while `invokeAndBlock` is running, this is because `invokeAndBlock()` keeps calling the main thread loop internally.



Notice that `invokeAndBlock` comes at a slight performance penalty. Also notice that nesting `invokeAndBlock` calls (or over using them) isn't recommended.
However, they're convenient when working with multiple threads/UI.

Even if you never call `invokeAndBlock` directly you're probably using it indirectly in APIs such as Dialog [https://www.codenameone.com/javadoc/com/codename1/ui/Dialog.html] that show a dialog while blocking the current thread e.g.:

```
public void actionPerformed(ActionEvent ev) {
  // will return true if the user clicks "OK"
  if(!Dialog.show("Question", "How Are You", "OK", "Not OK")) {
  // ask what went wrongnull
  }
}
```

Notice that the dialog show method will block the calling thread until the user clicks OK or Not OK...



Other APIs such as `NetworkManager.addToQueueAndWait()` also make use of this feature. Pretty much every "AndWait" method or blocking method uses this API internally!

To explain how invokeAndBlock works you can return to the sample above of how the EDT works:

`invokeAndBlock()` works in a similar way to this pseudo-code:

The EDT is effectively "blocked" but `invokeAndBlock` "redoes" it from within...

As you can see this is a simple approach for thread programming in UI, you don't need to block your flow and track the UI thread. You can program in a way that seems sequential (top to bottom) but uses multi-threading correctly without blocking the EDT.

Chapter 33. Graphics, drawing, images, and fonts



Drawing is considered a low-level API that might introduce some platform fragmentation.

33.1. Basics - where and how to draw manually

The Graphics [https://www.codenameone.com/javadoc/com/codename1/ui/Graphics.html] class handles drawing basics, shapes, images, and text. Developers never instantiate it directly; the Codename One API passes it in.

You can gain access to a `Graphics` object using one of the following methods:

- **Derive Component [https://www.codenameone.com/javadoc/com/codename1/ui/Component.html] or a subclass of `Component`** - `Component` includes many methods that let developers change drawing behavior. Override them to change how the component is drawn:
  - `paint(Graphics)` - invoked to draw the component, this can be overridden to draw the component from scratch.
  - `paintBackground(Graphics)` / `paintBackgrounds(Graphics)` - these allow overriding the way the component background is painted although you would probably be better off implementing a painter (see below).
  - `paintBorder(Graphics)` - allows overriding the process of drawing a border, notice that border drawing might differ based on the style of the component.
  - `paintComponent(Graphics)` - allows painting the components contents while leaving the default paint behavior to the style.
  - `paintScrollbars(Graphics)`, `paintScrollbarX(Graphics)`, `paintScrollbarY(Graphics)` allows overriding the behavior of scrollbar painting.
- **Implement the painter interface**; this interface can be used as a `GlassPane` or a background painter.

  The painter interface is simple and includes one paint method. It lets developers perform custom painting without subclassing `Component`. Painters can be chained together to create elaborate paint behavior with the PainterChain [https://www.codenameone.com/javadoc/com/codename1/ui/painter/PainterChain.html] class.

  - **Glass pane** - a glass pane allows developers to paint on top of the form painting. This allows an overlay effect on top of a form.

    At first, a glass pane may seem like overriding the Form's paint method and drawing after `super.paint(g)` completes. It isn't the same. When a component repaints by invoking `repaint()`, that component is drawn, and Form [https://www.codenameone.com/javadoc/com/codename1/ui/Form.html]'s `paint()` method isn't invoked. The glass pane painter is invoked in those cases and behaves as expected.

Container [https://www.codenameone.com/javadoc/com/codename1/ui/Container.html] has a glass-pane method called `paintGlass(Graphics)`, which you can override to provide a similar effect at the `Container` level. This is useful for complex containers such as Table [https://www.codenameone.com/javadoc/com/codename1/ui/table/Table.html], which draws its lines using this approach.

- **Background painter** - the background painter is installed through the style. By default, Codename One installs its own custom background painter. Installing a custom painter lets a developer define how the component background is drawn.
  Many background-style behaviors can also be achieved using styles alone.



A common mistake is overriding the `paint(Graphics)` method of `Form`. The problem is that a form has child components that may request a repaint. To avoid that, place a paintable component in the center of the `Form`, or override the `glass pane` or background painter.

A paint method can be implemented as such:

```
// hide the title
Form hi = new Form("", new BorderLayout());
hi.add(BorderLayout.CENTER, new Component() {
    @Override
    public void paint(Graphics g) {
        // red color
        g.setColor(0xff0000);

        // paint the screen in red
        g.fillRect(getX(), getY(), getWidth(), getHeight());

        // draw hi world in white text in the top left corner of the screen
        g.setColor(0xffffff);
        g.drawString("Hi World", getX(), getY());
    }
});
hi.show();
```

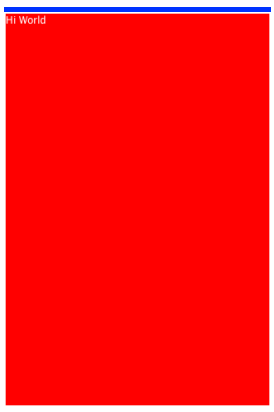


Figure 232. Hi world demo code, notice that the blue bar on top is the iOS7+ status bar

33.1.1. Painting with gradients

Solid colors are the starting point. `Graphics` also understands "paint" objects that describe gradients and other patterns. You can pass an instance of `Paint` [https://www.codenameone.com/javadoc/com/codename1/ui/Paint.html] to `setColor(Paint)` in place of an integer color value to activate a gradient for later fill and draw operations. The `LinearGradientPaint` [https://www.codenameone.com/javadoc/com/codename1/ui/LinearGradientPaint.html] class is the most common option and accepts a list of color stops along a line:

The `fillLinearGradient()` convenience methods (with optional `repeat` flag) provide a shorthand when you need a two-color gradient without constructing your own `Paint` object.

Radial gradients are equally straightforward using `fillRadialGradient()` or `fillRectRadialGradient()`, which can render circular and rectangular radial transitions respectively. These APIs accept the inner/outer colors, focal point, and spread so you can combine them with linear gradients to build sophisticated backgrounds and lighting effects.

For the full CSS gradient range, `Graphics` exposes a single `fillGradient` method that consumes a `Gradient` value object. `Gradient` is a `Paint` subclass with three concrete forms - `LinearGradient`, `RadialGradient`, `ConicGradient` - following the same pattern as the `Shape` hierarchy:

`Gradient` carries the cycle method (`CYCLE_NONE` / `CYCLE_REPEAT` / `CYCLE_REFLECT`) for repeating / reflected fills. Each port implements `fillGradient` via the fastest native shader available - Java2D `LinearGradientPaint` / `RadialGradientPaint` on the simulator, `LinearGradient` / `RadialGradient` / `SweepGradient` shaders on Android - and falls back to a software rasterizer that calls `Gradient#sampleArgb` per pixel where no native shader exists.

33.1.2. Image blur

`Graphics.gaussianBlur(Image, float radius)` returns a blurred copy of an image using the platform's fastest path (Core Image on iOS, RenderScript / RenderEffect on Android, JHLabs `GaussianFilter` in the simulator). The same blur is reachable from CSS via the `filter: blur(<length>)` and `backdrop-filter: blur(<length>)` properties, which are stored on the component's `Style` as `filterBlurRadius` and `backdropFilterBlurRadius`.

33.2. Glass pane

The `GlassPane` `in Codename One is inspired by the Swing `GlassPane` & `LayeredPane` with a few twists. You tried to imagine how Swing developers would have implemented the glass pane knowing what they do now about painters and Swings learning curve. But first: what's the glass pane?

A typical Codename One application is essentially composed of 3 layers (this is a gross simplification though), the background painters are responsible for drawing the background of all components including the main form. The component draws its own content (which might overrule the painter) and the glass pane paints last…

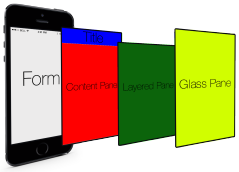


Figure 233. Form layout graphic

Essentially the glass pane is a painter that allows you to draw an overlay on top of the Codename One application.

Overriding the paint method of a form isn't a substitute for `glasspane` as it would appear to work initially, when you enter a `Form`. For example, when modifying an element within the form that element gets repainted not the entire `Form`!

If you have a form with a Button [https://www.codenameone.com/javadoc/com/codename1/ui/Button.html] and text drawn on top using the Form's paint method it would get erased whenever the button gets focus.

The glass pane is called whenever a component gets painted, it paints within the clipping region of the component hence it won't break the rest of the components on the `Form` which weren't modified.

You can set a painter on a form using code like this:

```
hi.setGlassPane(new Painter() {
    @Override
    public void paint(Graphics g, Rectangle rect) {
    }
});
```

Or you can use Java 8 lambdas to tighten the code a bit:

```
hi.setGlassPane((g, rect) -> {
});
```

PainterChain [https://www.codenameone.com/javadoc/com/codename1/ui/painter/PainterChain.html] allows you to chain many painters together to perform different logical tasks such as a validation painter coupled with a fade out painter. The sample below shows a crude validation panel that allows you to draw error icons next to components while exceeding their physical bounds as is common in many user interfaces

```
Form hi = new Form("Glass Pane", new BoxLayout(BoxLayout.Y_AXIS));
Style s = UIManager.getInstance().getComponentStyle("Label");
s.setFgColor(0xff0000);
s.setBgTransparency(0);
Image warningImage = FontImage.createMaterial(FontImage.MATERIAL_WARNING, s).toImage();
TextField tf1 = new TextField("My Field");
tf1.getAllStyles().setMarginUnit(Style.UNIT_TYPE_DIPS);
```

```
tf1.getAllStyles().setMargin(5, 5, 5, 5);
hi.add(tf1);
hi.setGlassPane((g, rect) -> {
    int x = tf1.getAbsoluteX() + tf1.getWidth();
    int y = tf1.getAbsoluteY();
    x -= warningImage.getWidth() / 2;
    y += (tf1.getHeight() / 2 - warningImage.getHeight() / 2);
    g.drawImage(warningImage, x, y);
});
hi.show();
```



Figure 234. The glass pane draws the warning sign on the border of the component peeking out

33.3. Shapes & transforms

The graphics API provides a high performance shape API that allows drawing arbitrary shapes by defining paths and curves and caching the shape drawn in the GPU.

33.4. Device support

Shapes and transforms ship with all Codename One's actively maintained ports (Android, iOS, JavaScript, and desktop/Simulator). Older platforms that have reached end of life may lack these APIs, so keep the guard code shown below if you still target them with legacy builds.

Notice that perspective transform is missing from the desktop/simulator port. No real equivalent to perspective transform in Java SE that you could use.

33.5. A 2D drawing app

You can show shape drawing with a simple example of a drawing app, that allows the user to tap the screen to draw a contour picture.

The app works by keeping a GeneralPath [https://www.codenameone.com/javadoc/com/codename1/ui/geom/GeneralPath.html] in memory, and adding points as Bézier curves. Whenever a point is added, the path is redrawn to the screen.

The center of the app is the `DrawingCanvas` class, which extends Component [https://www.codenameone.com/javadoc/com/codename1/ui/Component.html]:

```
public class DrawingCanvas extends Component {
    GeneralPath p = new GeneralPath();
    int strokeColor = 0x0000ff;
    int strokeWidth = 10;
```

```
    public void addPoint(float x, float y){
        // To be written
    }

    @Override
    protected void paintBackground(Graphics g) {
        super.paintBackground(g);
            Stroke stroke = new Stroke(
                strokeWidth,
                Stroke.CAP_BUTT,
                Stroke.JOIN_ROUND, 1f
            );
            g.setColor(strokeColor);

            // Draw the shape
            g.drawShape(p, stroke);

    }

    @Override
    public void pointerPressed(int x, int y) {
        addPoint(x-getParent().getAbsoluteX(), y-getParent().getAbsoluteY());
    }
}
```

Conceptually this is basic component. You will be overriding the `paintBackground()` [https://www.codenameone.com/javadoc/com/codename1/ui/Component.html#paintBackground(com.codename1.ui.Graphics)] method to draw the path. You keep a reference to a GeneralPath [https://www.codenameone.com/javadoc/com/codename1/ui/geom/GeneralPath.html] object (which is the concrete implementation of the Shape [https://www.codenameone.com/javadoc/com/codename1/ui/geom/Shape.html] interface in Codename One) to store each successive point in the drawing. You also parametrize the stroke width and color.

The implementation of the `paintBackground()` method (shown above) should be straight forward. It creates a stroke of the appropriate width, and sets the color on the graphics context. Then it calls `drawShape()` to render the path of points.

33.5.1. Implementing addpoint()

The addPoint method is designed to allow you to add points to the drawing. A simple implementation that uses straight lines rather than curves might look like this:

You introduced a couple house-keeping member vars (`lastX` and `lastY`) to store the last point that was added that you know whether this is the first tap or a later tap. The first tap triggers a `moveTo()` call, whereas later taps trigger `lineTo()` calls, which draw lines from the last point to the current point.

A drawing might look like this:

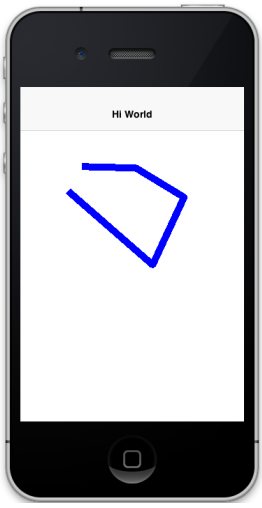


Figure 235. lineTo example

33.5.2. Using Bézier curves

Your previous implementation of addPoint() used lines for each segment of the drawing. Now make a change to allow for smoother edges by using quadratic curves instead of lines.

Codename One's `GeneralPath` class includes two methods for drawing curves:

1. `quadTo()` [https://www.codenameone.com/javadoc/com/codename1/ui/geom/GeneralPath.html#quadTo(float,%20float,%20float,%20float)] : Appends a quadratic Bézier curve. It takes 2 points: a control point, and an end point.
2. `curveTo()` [https://www.codenameone.com/javadoc/com/codename1/ui/geom/GeneralPath.html#curveTo(float,%20float,%20float,%20float,%20float,%20float)] : Appends a cubic Bézier curve, taking 3 points: 2 control points, and an end point.

See the General Path javadocs [https://www.codenameone.com/javadoc/com/codename1/ui/geom/GeneralPath.html] for the full API.

You will make use of the `quadTo()` [https://www.codenameone.com/javadoc/com/codename1/ui/geom/GeneralPath.html#quadTo(float,%20float,%20float,%20float)] method to append curves to the drawing as follows:

This change should be straight forward except, perhaps, the business with the `odd` variable. Since quadratic curves require two points (also to the implied starting point), you can't take the last tap point and the current tap point. You need a point between them to act as a control point. This is where you get the curve from. The control point works by exerting a sort of "gravity" on the line segment, to pull the line towards it. This results in the line being curved. The example uses the `odd` marker to alternate the control point between positions above the line and below the line.

A drawing from the resulting app looks like:

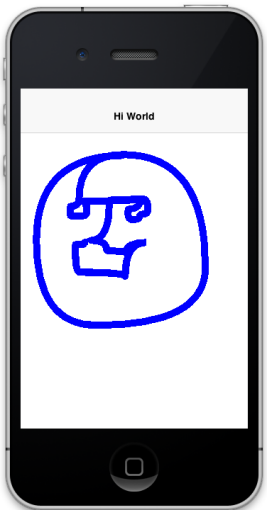


Figure 236. Result of quadTo example

33.5.3. Detecting platform support

The `DrawingCanvas` example is a bit naive in that it assumes that the device supports the shape API. Running this code on a device that doesn't support the Shape [https://www.codenameone.com/javadoc/com/codename1/ui/geom/Shape.html] API would draw a blank canvas where the shape was expected to be drawn. You can fall back gracefully if you make use of the `Graphics.isShapeSupported()` [https://www.codenameone.com/javadoc/com/codename1/ui/Graphics.html#isShapeSupported()] method. For example:

33.6. Transforms

The Graphics [https://www.codenameone.com/javadoc/com/codename1/ui/Graphics.html] class has included limited support for 2D transformations for some time now including scaling, rotation, and translation:

- `scale(x,y)` : Scales drawing operations by a factor in each direction.
- `translate(x,y)` : Translates drawing operations by an offset in each direction.
- `rotate(angle)` : Rotates about the origin.
- `rotate(angle, px, py)` : Rotates about a pivot point.



`scale()` and `rotate()` methods are available on platforms that support Affine transforms. See table X for a compatibility list.

33.6.1. Device support

All current Codename One ports expose affine transforms (that's: `scale()` and `rotate()`). Use the following table as a quick reference when deciding whether you need a fallback path.

Table 9. Transforms Device Support

Platform	Affine Supported
Simulator/Desktop	Yes

iOS	Yes
Android	Yes
JavaScript	Yes

You can check if a particular Graphics [https://www.codenameone.com/javadoc/com/codename1/ui/Graphics.html] context supports rotation and scaling using the `isAffineSupported()` method.

For example:

```
public void paint(Graphics g){
    if ( g.isAffineSupported() ){
        // Do something that requires rotation and scaling

    } else {
        // Fallback behavior here
    }
}
```

33.7. Example: Drawing an analog clock

The following sections implement an analog clock component. This will show three key concepts in Codename One's graphics:

1. Using the `GeneralPath` class for drawing arbitrary shapes.
2. Using `Graphics.translate()` to translate your drawing position by an offset.
3. Using `Graphics.rotate()` to rotate your drawing position.

There are three separate things that need to be drawn in a clock:

1. **The tick marks**. For example: most clocks will have a tick mark for each second, larger tick marks for each hour, and sometimes even larger tick marks for each quarter-hour.
2. **The numbers**. You will draw the clock numbers (1 through 12) in the appropriate positions.
3. **The hands**. You will draw the clock hands to point at the appropriate points to display the current time.

33.7.1. The AnalogClock Component

Your clock will extend the Component [https://www.codenameone.com/javadoc/com/codename1/ui/Component.html] class, and override the `paintBackground()` method to draw the clock as follows:

```
public class AnalogClock extends Component {
    Date currentTime = new Date();

    @Override
    public void paintBackground(Graphics g) {
```

```
        // Draw the clock in this method
    }
}
```

33.7.2. Setting up the parameters

Before you actually draw anything, take a moment to figure out what values you need to know to draw an effective clock. Minimally, you need two values:

1. The center point of the clock.
2. The radius of the clock.

The clock also exposes the following parameters to help customize how it's rendered:

1. **The padding** (that's: the space between the edge of the component and the edge of the clock circle.
2. **The tick lengths**. The example uses 3 different lengths of tick marks on this clock. The longest ticks will be displayed at quarter points (that's: 12, 3, 6, and 9). Slightly shorter ticks will be displayed at the five-minute marks (that's: where the numbers appear), and the remaining marks (corresponding with seconds) will be short:

33.7.3. Drawing the tick marks

For the tick marks, you will use a single GeneralPath [https://www.codenameone.com/javadoc/com/codename1/ui/geom/GeneralPath.html] object, making use of the `moveTo()` and `lineTo()` methods to draw each individual tick:



This example uses a little bit of trigonometry to calculate the `(x,y)` coordinates of the tick marks based on the angle and the radius. If math isn't your thing, don't worry. This example makes use of the identities: `x=r*cosθ` and `y=r*sinθ`.

At this point your clock should include a series of tick marks orbiting a blank center as shown below:

Figure 237. Drawing tick marks on the watch face

33.7.4. Drawing the numbers

The `Graphics.drawString(str, x, y)` method allows you to draw text at any point of a component. The tricky part here is calculating the correct `x` and `y` values for each string so that the number appears in the correct location.

For the purposes of this tutorial, you will use the following strategy. For each number (1 through 12):

1. Use the `Graphics.translate(x,y)` method to apply a translation from the clock's center point to the point where the number should appear.
2. Draw number (using `drawString()`) at the clock's center. It should be rendered at the correct point due to your translation.
3. Invert the translation performed in step 1:



This example is, admittedly, a little contrived to allow for a demonstration of the `Graphics.translate()` method. You could have as passed the exact location of the number to `drawString()` rather than draw at the clock center and translate to the correct location.

Now, you should have a clock with tick marks *and* numbers as shown below:

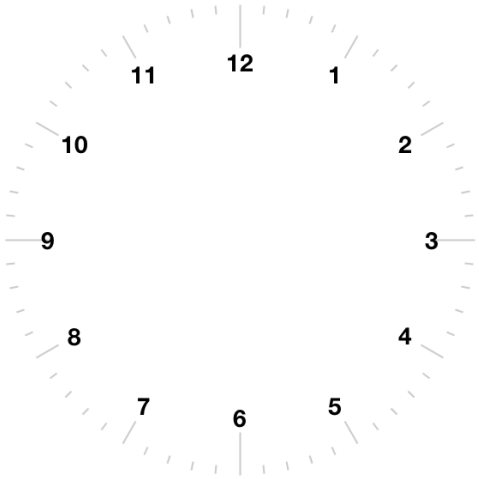


Figure 238. Drawing the numbers on the watch face

33.7.5. Drawing the hands

The clock will include three hands: Hour, Minute, and Second. You will use a separate GeneralPath [https://www.codenameone.com/javadoc/com/codename1/ui/geom/GeneralPath.html] object for each hand. For the positioning/angle of each, use the following strategy:

1. Draw the hand at the clock center pointing toward 12 (straight up).
2. Translate the hand slightly down so that it overlaps the center.
3. Rotate the hand at the appropriate angle for the current time, using the clock center as a pivot point.

Drawing the Second Hand:

For the "second" hand, you will use a simple line from the clock center to the inside edge of the medium tick mark at the 12 o'clock position:

And you will translate it down slightly so that it overlaps the center. This translation will be performed on the `GeneralPath` object directly rather than through the `Graphics` context:

Rotating the Second Hand::

The rotation of the second hand will be performed in the `Graphics` context through the `rotate(angle, px, py)` method. This requires you to calculate the angle. The `px` and `py` arguments constitute the pivot point of the rotation, which, in your case will be the clock center.



The rotation pivot point is expected to be in absolute screen coordinates rather than relative coordinates of the component. You therefore need to get the absolute clock center position to perform the rotation:



Remember to call `resetAffine()` after you're done with the rotation, or you will see some unexpected results on your form.

Drawing the Minute And Hour Hands:

The mechanism for drawing the hour and minute hands is largely the same as for the minute hand, with a couple of added complexities though:

1. You will make these hands trapezoidal, and almost triangular rather than using a simple line, which means the `GeneralPath` construction will be slightly more complex.
2. Calculation of the angles will be slightly more complex because they need to consider many parameters. For example: The hour hand angle is informed by both the hour of the day and the minute of the hour.

The remaining drawing code is as follows:

33.7.6. The final result

At this point, you have a complete clock as shown below:

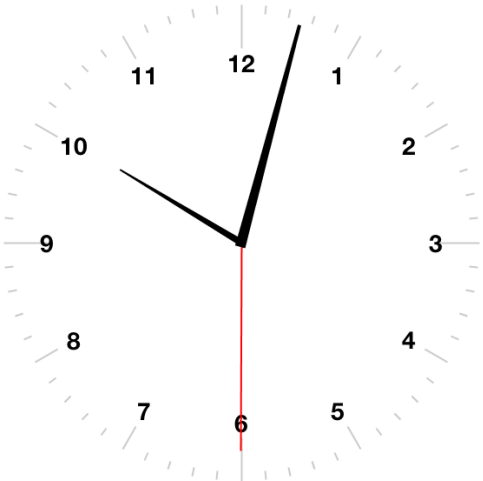


Figure 239. The final result - fully rendered watch face

33.7.7. Animating the clock

The current clock component is cool, but it's static. It displays the time at the point the clock was created. You discussed low-level animations in the animation section of the guide, here you will show a somewhat more elaborate example.

To animate your clock so that it updates once per second, you need to do two things:

1. Implement the `animate()` method to show when the clock needs to be updated/re-drawn.
2. Register the component with the form so that it will receive animation "pulses."

The `animate()` method in the `AnalogClock` class:

This method will be invoked on each "pulse" of the EDT. It checks the last time the clock was rendered and returns `true` if the clock hasn't been rendered in the current "time second" interval. Otherwise it returns false. This ensures that the clock will be redrawn when the time changes.

33.8. Starting and stopping the animation

Animations can be started and stopped through the `Form.registerAnimated(component)` and `Form.deregisterAnimated(component)` methods. You chose to encapsulate these calls in `start()` and `stop()` methods in the component as follows:

The code to instantiate the clock, and start the animation would be something like:

33.9. Shape clipping

Clipping is one of the core tenants of graphics programming, you define the boundaries for drawing and when you exceed said boundaries things aren't drawn. Shape clipping allows you to clip based on any arbitrary `Shape` and not a rectangle, this allows some unique effects generated in runtime.

For example: this code allows you to draw a rather complex image of duke:

```
Image duke = null;
try {
    // duke.png is just the default Codename One icon copied into place
    duke = Image.createImage("/duke.png");
} catch(IOException err) {
    Log.e(err);
}
final Image finalDuke = duke;

Form hi = new Form("Shape Clip");

// We create a 50 x 100 shape, this is arbitrary since we can scale it easily
GeneralPath path = new GeneralPath();
path.moveTo(20,0);
path.lineTo(30, 0);
path.lineTo(30, 100);
path.lineTo(20, 100);
path.lineTo(20, 15);
```

```
path.lineTo(5, 40);
path.lineTo(5, 25);
path.lineTo(20,0);

Stroke stroke = new Stroke(0.5f, Stroke.CAP_ROUND, Stroke.JOIN_ROUND, 4);
hi.getContentPane().getUnselectedStyle().setBgPainter((Graphics g, Rectangle rect) -> {
    g.setColor(0xff);
    float widthRatio = ((float)rect.getWidth()) / 50f;
    float heightRatio = ((float)rect.getHeight()) / 100f;
    g.scale(widthRatio, heightRatio);
    g.translate((int)(((float)rect.getX()) / widthRatio), (int)(((float)rect.getY()) / heightRatio));
    g.setClip(path);
    g.setAntiAliased(true);
    g.drawImage(finalDuke, 0, 0, 50, 100);
    g.setClip(path.getBounds());
    g.drawShape(path, stroke);
    g.translate(-(int)(((float)rect.getX()) / widthRatio), -(int)(((float)rect.getY()) / heightRatio));
    g.resetAffine();
});

hi.show();
```

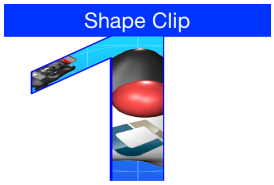


Figure 240. Shape Clipping used to clip the image of duke within the given shape



Notice that this functionality isn't available on all platforms so you need to test if shaped clipping is supported using isShapeClipSupported() [https://www.codenameone.com/javadoc/com/codename1/ui/Graphics.html#isShapeClipSupported--].

33.10. The coordinate system

The Codename One coordinate system follows the example of Swing (and many other - but not all-graphics libraries) and places the origin in the upper left corner of the screen. X-values grow to the right, and Y-values grow downward as illustrated below:

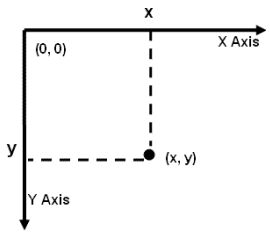


Figure 241. The Codename One graphics coordinate space

The screen origin is in the top left corner of the screen. Given this information, consider the method call on the Graphics [https://www.codenameone.com/javadoc/com/codename1/ui/Graphics.html] context `g`:

```
g.drawRect(10,10, 100, 100);
```

Where would this rectangle be drawn on the screen?

If you answered something like "10 pixels from the top, and 10 pixels from the left of the screen," you *might* be right. It depends on whether the graphics has a translation or transform applied to it. If there is currently a translation of `(20,20)` (that's: 20 pixels to the right, and 20 pixels down), then the rectangle would be rendered at `(30, 30)`.

You can always find out the current translation of the graphics context using the `Graphics.getTranslateX()` and `Graphics.getTranslateY()` methods:

```
// Find out the current translation
int currX = g.getTranslateX();
int currY = g.getTranslateY();

// Reset the translation to zeroes
g.translate(-currX, -currY);

// Now we are working in absolute screen coordinates
g.drawRect(10, 10, 100, 100);

// This rectangle should now be drawn at the exact screen
// coordinates (10,10).

//Restore the translation
g.translate(currX, currY);
```



This example glosses over issues such as clipping and transforms which may cause it to not work as you expect. For example: When painting a component inside its `paint()` method, there is a clip applied to the context so that the content you draw within the bounds of the component will be seen.

If, also, there is a transform applied that rotates the context 45 degrees clockwise, then the rectangle will be drawn at a 45-degree angle with its top left corner somewhere on the left edge of the screen.

You don't have to worry about the exact screen coordinates for the things you paint. Most of the time, you will be concerned with relative coordinates.

33.10.1. Relative coordinates

Usually, when you're drawing onto a `Graphics` context, you're doing so within the context of a Component's `paint()` method (or one of its variants). In this case, you don't care what the exact screen coordinates are of your drawing. You're concerned with their relative location within the coordinate. You can leave the positioning (and even sizing) of the coordinate up to Codename One. Thank you for reading.

To show this, create a simple component called Rectangle [https://www.codenameone.com/javadoc/com/codename1/ui/geom/Rectangle.html] component, that draws a rectangle on the screen. You will use the component's position and size to dictate the size of the rectangle to be drawn. And you will keep a 5 pixel padding between the edge of the component and the edge of your rectangle:

```
class RectangleComponent extends Component {
    public void paint(Graphics g){
        g.setColor(0x0000ff);
        g.drawRect(getX()+5, getY()+5, getWidth()-10, getHeight()-10);
    }
}
```

The result is as follows:



Figure 242. The rectangle component



The `x` and `y` coordinates that are passed to the `drawRect(x,y,w,h)` method are relative to the component's *parent's* origin — **not the component itself.. Its parent.** This is why you the *x* position is `getX()+5` and not *5*.

33.10.2. Transforms and rotations

Unlike the `Graphics drawXXX` primitives, methods for setting transformations, including `scale(x,y)` and `rotate(angle)`, are always applied in screen coordinates. This can be confusing at first, because you may be unsure whether to provide a relative coordinate or an absolute coordinate for a given method.

The general rule is:

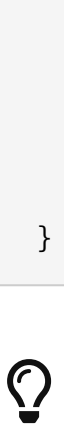
1. **All coordinates passed to the drawXXX() and fillXXX() methods will be subject to the graphics context's transform and translation settings**.
2. **All coordinates passed to the context's transformation settings are considered to be screen coordinates, and aren't subject to current transform and translation settings**.

Take your `RectangleComponent` as an example. Suppose you want to rotate the rectangle by 45 degrees, your first try might look something like:

```
class RectangleComponent extends Component {
```

```
        @Override
        protected Dimension calcPreferredSize() {
            return new Dimension(250,250);
        }

        public void paint(Graphics g) {
            g.setColor(0x0000ff);
            g.rotate((float) (Math.PI / 4.0));
            g.drawRect(getX() + 5, getY() + 5, getWidth() - 10, getHeight() - 10);
            g.rotate(-(float) (Math.PI / 4.0));
        }
    }
```



When performing rotations and transformations inside a `paint()` method, always remember to revert your transformations at the end of the method so that it doesn't pollute the rendering pipeline for later components.

The behavior of this rotation will vary based on where the component is rendered on the screen. To show this, try to place five of these components on a form inside a BorderLayout [https://www.codenameone.com/javadoc/com/codename1/ui/layouts/BorderLayout.html] and see how it looks:

The result is as follows:

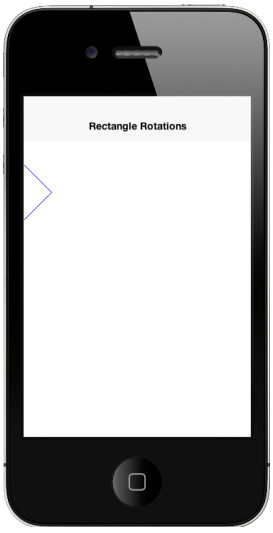


Figure 243. Rotating the rectangle

This may not be an intuitive outcome since you drew 10 rectangle components, but you see a portion of one rectangle. The reason is that the `rotate(angle)` method uses the screen origin as the pivot point for the rotation. Components nearer to this pivot point will experience a less dramatic effect than components farther from it. In your case, the rotation has caused all rectangles except the first one to be rotated outside the bounds of their containing component, and are therefore being clipped. A more sensible solution for your component would be to place the rotation pivot point somewhere inside the component. That way all the components would look the same. Some possibilities would be:

Top Left Corner:



Figure 244. Rotating the rectangle with wrong pivot point

Center:

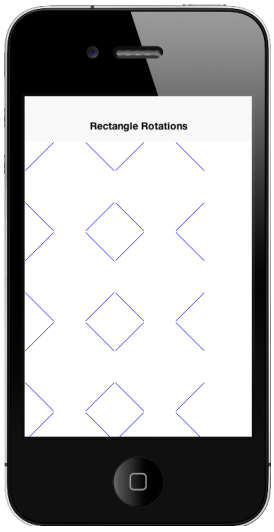


Figure 245. Rotating the rectangle with the center pivot point

You could also use the `Graphics.setTransform()` class to apply rotations and other complex transformations (including 3D perspective transforms), but that belongs to its own topic as it's a little bit more complex.

33.10.3. Global alpha & Anti-Aliasing

Up to this point you've relied on the per-pixel alpha stored in images and gradients. `Graphics` also lets you apply a global alpha multiplier to every draw call by using `setAlpha(int)` or `concatenateAlpha(int)` after checking `isAlphaSupported()`. Both methods accept values from `0` (fully transparent) to `255` (fully opaque) and remain active until you change them again. `concatenateAlpha()` is especially handy when you need to temporarily fade a component because it returns the previous alpha so you can restore it later.

Anti-aliasing can likewise be toggled at runtime. Call `isAntiAliasingSupported()` and `isAntiAliasedTextSupported()` to discover which hints the current port exposes, then use `setAntiAliased(boolean)` and `setAntiAliasedText(boolean)` to opt into smoother edges for shapes and glyphs respectively. These switches make it easy to balance rendering quality versus speed

depending on the type of content you draw.

33.10.4. Event coordinates

The coordinate system and event handling are tied. You can listen for touch events on a component by overriding the `pointerPressed(x,y)` method. The coordinates received in this method will be **absolute screen coordinates**, so you may need to do some conversions on these coordinates before using them in your `drawXXX()` methods.

For example: a `pointerPressed()` callback method can look like this:

In this case you translated these points so that they would be relative to the origin of the parent component. This is because the `drawXXX()` methods for this component take coordinates relative to the parent component.

33.11. Images

Codename One has a few image types: loaded, RGB (built-in), RGB (Codename One), Mutable, EncodedImage, SVG, MultiImage, FontImage & Timeline. URLImage, FileEncodedImage, FileEncodedImageAsync, and `StorageEncodedImage`/Async also exist, and are covered in the IO section.

All image types are seamless to use and will work with `drawImage` and various image related image APIs for the most part with caveats on performance etc.



For animation images the code must invoke the `animate()` method on the image (this is done automatically by Codename One when placing the image as a background or as an icon!
You need to do it if you invoke `drawImage` in code rather than use a built-in component).

Performance and memory wise you should read the section below and be aware of the image types you use. The Codename One designer tries to conserve memory and be "clever" by using `EncodedImage`. While these are great for low memory you need to understand the complexities of image locking and be aware that you might pay a penalty if you don't.

Here are the pros/cons and logic behind every image type. This covers the logic of how it's created:

33.11.1. Loaded Image

This is the basic image you get when loading an image from the jar or network using Image.createImage(String) [https://www.codenameone.com/javadoc/com/codename1/ui/Image.html#createImage-java.lang.String-], Image.createImage(InputStream) [https://www.codenameone.com/javadoc/com/codename1/ui/Image.html#createImage-java.io.InputStream-] & Image.`createImage(byte[` [https://www.codenameone.com/javadoc/com/codename1/ui/Image.html#createImage-byte:A-int-int-], int, int)],...



Some other APIs might return this image type but those APIs do so explicitly!

In some platforms calling `getGraphics()` on an image like this will throw an exception as it's immutable). This is true for most other images as well.

This restriction might not apply for all platforms.

The image is stored in RAM based on device logic and should be reasonably efficient in drawing speed. For example, it takes up a lot of RAM.

To calculate the amount of RAM taken by a loaded image you use the following formula:

```
Image Width * Image Height * 4 = Size In RAM in Bytes
```

For example: a 50×100 image will take up 20,000 bytes of RAM.

The logic behind this is simple, every pixel contains 3 color channels and an alpha component hence 3 bytes for color and one for alpha.



This isn't the case for all images but it's common and you prefer calculating for the worst case scenario. Even with JPEGs that don't include an alpha channel some OSes might require that more byte.

33.11.2. The RGB image's

There are two types of RGB constructed images that are different from one another but since they're both technically "RGB image's" you're bundling them under the same subsection.

Internal

This is a close cousin of the loaded image. This image is created using the method Image.createImage(int array, int, int) [https://www.codenameone.com/javadoc/com/codename1/ui/Image.html#createImage-int:A-int-int-] and receives the AARRGGBB data to form the image. It's more efficient than the Codename One RGB image but can't be modified, at least not on the pixel level.

The goal of this image type is to provide an easy way to render RGB data that isn't modified efficiently at platform native speeds. It's technically a standard "Loaded Image" internally.

RGBImage class

RGBImage [https://www.codenameone.com/javadoc/com/codename1/ui/RGBImage.html] is effectively an AARRGGBB array that can be drawn by Codename One.

On most platforms this is inefficient but for some pixel level manipulations there is no other way.

An `RGBImage` is constructed with an `int` array (`int[]`) that includes `width*height` elements. You can then change the colors and alpha channel directly within the array and draw the image to any source using standard image drawing APIs.



This is inefficient in rendering speed and memory overhead. Only use this technique if there is absolutely no other way!

33.11.3. EncodedImage

EncodedImage [https://www.codenameone.com/javadoc/com/codename1/ui/EncodedImage.html] is the workhorse of Codename One. Images returned from resource files are `EncodedImage` and many APIs expect it.

The `EncodedImage` is effectively a loaded image that's "hidden" and extracted as needed to remove the memory overhead associated with loaded image. When creating an `EncodedImage` the PNG (or JPEG etc.) is loaded to an array in RAM. Such images are small (relatively) so they can be kept in memory without much overhead.

When image information is needed (pixels) the image is decoded into RAM and kept in a weak/sort reference. This allows the image to be cached for performance and allows the garbage collector to reclaim it when the memory becomes scarce.

Since the fully decoded image can be pretty big (`width X height X 4`) the ability to store the encoded image can be pretty stark. For example: taking your example above a 50×100 image will take up 20,000 bytes of RAM for a loaded image but an `EncodedImage` can reduce that to 1kb-2kb of RAM.



An `EncodedImage` might be more expensive than a loaded image as it will take up both the encoded size and the loaded size. The cost might be slightly bigger sometimes. It's main value is its ability to shrink.

When drawing an `EncodedImage` it checks the weak reference cache and if the image is cached then it's shown otherwise the image is loaded the encoded image cache it then drawn.

`EncodedImage` isn't final and can be derived to produce complex image fetching strategies for example: the URLImage [https://www.codenameone.com/javadoc/com/codename1/ui/URLImage.html] class that can dynamically download its content from the web.

`EncodedImage` can be instantiated through the create methods in the `EncodedImage` class. Pretty much any image can be converted into an `EncodedImage` through the createFromImage(Image, boolean) [https://www.codenameone.com/javadoc/com/codename1/ui/EncodedImage.html#createFromImage-com.codename1.ui.Image-boolean-] method.

EncodedImage Locking

Loading the image is more expensive so you want the images that are on the current form to remain in cache (otherwise GC will thrash a lot). That's where `lock()` kicks in, when `lock()` is active you keep a hard reference to the actual native image so it won't get GC'd. This significantly improves performance!

Internally this is invoked automatically for background images, icons etc. which results in a huge performance boost. This makes sense since these images are showing and they will be in RAM anyway. But, if you use a complex renderer or custom drawing UI you should `lock()` your images where possible!

To verify that locking might be a problem you can launch the performance monitor tool (accessible from the simulator menu), if you get log messages that show that an unlocked

image was drawn you might have a problem.

33.11.4. MultiImage

Multi images don't physically exist as a concept within the Codename One API so there is no way to actually create them and they're in no way distinguishable from `EnclodedImage`.

The built-in support for multi images is in the resource file loading logic where a MultiImage is decoded and the version that matches the current DPI is physically loaded. From that point on user code can treat it like any other `EnclodedImage`.

The 9-image borders use multi images by default to keep their appearance more refined on the different DPI's.

33.11.5. FontImage & Material design icons

FontImage [https://www.codenameone.com/javadoc/com/codename1/ui/FontImage.html] allows using an icon font as if it was an image. You can specify the character, color and size and then treat the `FontImage` as if its a regular image. The huge benefits are that the font image can adapt to platform conventions in color and scale to adapt to DPI.

You can generate icon fonts using free tools on the internet such as this [http://fontello.com/]. Icon fonts are a simple and powerful technique to create a small, modern applications.

Icon fonts can be created in 2 basic ways the first is explicitly by defining all the elements within the font:

```
Form hi = new Form("Icon Font");
Font materialFont = FontImage.getMaterialDesignFont();
int w = Display.getInstance().getDisplayWidth();
FontImage fntImage = FontImage.createFixed("\uE161", materialFont, 0xff0000, w, w);
hi.add(fntImage);
hi.show();
```

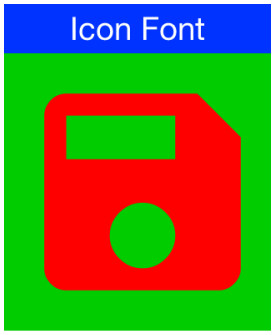


Figure 246. Icon font from material design icons created with the fixed size of display width



The samples use the built-in material design icon font. This is for convenience so the sample will work out of the box, for everyone. For example you should be able

> to do this with any arbitrary icon font off the internet as long as its a valid TTF file.

A more common and arguably "correct" way to construct such an icon would be through the Style [https://www.codenameone.com/javadoc/com/codename1/ui/plaf/Style.html] object. The `Style` object can provide the color, size and background information needed by `FontImage`.

There are two versions of this method: the first one expects the `Style` object to have the correct icon font set to its font attribute. The second accepts a `Font` object as an argument. The latter is useful for a case where you want to reuse the same `Style` object that you defined for a general UI element for example: you can set an icon for a `Button` like this and it will take up the style of the `Button`:

```
Form hi = new Form("Icon Font");
Font materialFont = FontImage.getMaterialDesignFont();
int size = Display.getInstance().convertToPixels(6, true);
materialFont = materialFont.derive(size, Font.STYLE_PLAIN);
Button myButton = new Button("Save");
myButton.setIcon(FontImage.create("\uE161", myButton.getUnselectedStyle(), materialFont));
hi.add(myButton);
hi.show();
```



Figure 247. An image created from the Style object



Notice that for this specific version of the method the size of the font is used to determine the icon size. In the other methods for `FontImage` creation the size of the font is ignored!

Material design icons

There are many icon fonts on the web, but the field is rather volatile and constantly changing. For example, you wanted to have built-in icons that would allow you to create better looking demos and built-in components.

That's why you picked the material design icon font for inclusion in the Codename One distribution. It features a stable core set of icons, that aren't IP encumbered.

You can use the built-in font directly as demonstrated above but there are far better ways to create a material design icon. To find the icon you want you can check out the material design icon gallery [https://design.google.com/icons/]. For example: you used the save icon in the samples above.

To recreate the save icon from above you can do something like:

```
Form hi = new Form("Icon Font");
Button myButton = new Button("Save");
myButton.setIcon(FontImage.createMaterial(FontImage.MATERIAL_SAVE, myButton.getUnselectedStyle()));
hi.add(myButton);
```



Figure 248. Material save icon



Notice that the icon is smaller now as it's calculated based on the font size of the `Button` UIID.

You can even write the code in a more terse style using:

```
Form hi = new Form("Icon Font");
Button myButton = new Button("Save");
FontImage.setMaterialIcon(myButton, FontImage.MATERIAL_SAVE);
hi.add(myButton);
```

This will produce the same result for slightly shorter syntax.



`FontImage` can conflict with some complex APIs that expect a "real" image underneath. Some odd issues can often be resolved by using the `toImage()` or `toEncodedImage()` methods to convert the scaled `FontImage` to a loaded image.

33.11.6. Timeline

Timelines allow rudimentary animation and enable GIF importing using the Codename One Designer. Effectively a timeline is a set of images that can be moved rotated, scaled & blended to provide interesting animation effects. It can be created manually using the Timeline [https://www.codenameone.com/javadoc/com/codename1/ui/animations/Timeline.html] class.

33.11.7. Image masking

Image masking allows you to manipulate images by changing the opacity of an image according to a mask image. The mask image can be hardcoded or generated dynamically, it's then converted to a Mask object that can be applied to any image. Notice that the masking process is computationally intensive, it should be done once and cached/saved.

The code below can convert an image to a rounded image:

```
Toolbar.setGlobalToolbar(true);
Form hi = new Form("Rounder", new BorderLayout());
Label picture = new Label("", "Container");
hi.add(BorderLayout.CENTER, picture);
hi.getUnselectedStyle().setBgColor(0xff0000);
hi.getUnselectedStyle().setBgTransparency(255);
Style s = UIManager.getInstance().getComponentStyle("TitleCommand");
Image camera = FontImage.createMaterial(FontImage.MATERIAL_CAMERA, s);
hi.getToolbar().addCommandToRightBar("", camera, (ev) -> {
    try {
        int width = Display.getInstance().getDisplayWidth();
        Image capturedImage = Image.createImage(Capture.capturePhoto(width, -1));
```

```
        Image roundMask = Image.createImage(width, capturedImage.getHeight(), 0xff000000);
        Graphics gr = roundMask.getGraphics();
        gr.setColor(0xffffff);
        gr.fillArc(0, 0, width, width, 0, 360);
        Object mask = roundMask.createMask();
        capturedImage = capturedImage.applyMask(mask);
        picture.setIcon(capturedImage);
        hi.revalidate();
    } catch(IOException err) {
        Log.e(err);
    }
});
```



Figure 249. Picture after the capture was complete and the resulted image was rounded. The background was set to red so the rounding effect will be more noticeable

Notice that this example is simplistic to be self contained. You often recommend that developers ship "ready made" mask images with their application which can allow complex effects on the images.

33.11.8. URLImage

URLImage [https://www.codenameone.com/javadoc/com/codename1/ui/URLImage.html] is an image created with a URL, it implicitly downloads and adapts the image in the given URL while caching it locally. The typical adapt process scales the image or crops it to fit into the same size which is a hard restriction because of the way `URLImage` is implemented.

How Does URLImage Work?

The reason for the size restriction lies in the implementation of `URLImage`. `URLImage` is physically an animated image and so the UI thread tries to invoke its `animate()` method to refresh. The `URLImage` uses that call to check if the image was fetched and if not fetches it asynchronously.

Once the image was fetched the `animate()` method returns true to refresh the UI. During the loading process the placeholder is shown, the reason for the restriction in size is that image animations can't "grow" the image. They're assumed to be fixed so the placeholder must match the dimensions of the resulting image.

The simple use case is pretty trivial:

Or you can use the similar `URLImage.createToFileSystem` method instead of the Storage [https://www.codenameone.com/javadoc/com/codename1/io/Storage.html] version.

This image can now be used anywhere a regular image will appear, it will initially show the placeholder image and then seamlessly replace it with the file after it was downloaded and stored. Notice that if you make changes to the image itself (for example: the `scaled` method) it will generate a new image which won't be able to fetch the actual image.



Since ImageIO [https://www.codenameone.com/javadoc/com/codename1/ui/util/ImageIO.html] is used to perform the operations of the adapter interface its required that `ImageIO` will work. it's working in Java SE, Android, iOS & Windows Phone.

If the file in the URL contains an image that's too big it will scale it to match the size of the placeholder precisely!
An option also exists to fail if the sizes don't match. Notice that the image that will be saved is the scaled image, this means you will have little overhead in downloading images that are the wrong size although you will get some artifacts.

The last argument is powerful, its an interface called URLImage.ImageAdapter [https://www.codenameone.com/javadoc/com/codename1/ui/URLImage.ImageAdapter.html] and you can implement it to adapt the downloaded image in any way you like. For example: you can use an image mask to automatically create a rounded version of the downloaded image.

To do this you can override:

In the adapter interface and return the processed encoded image. If you do heavy processing (for example: rounded edge images) you would need to convert the processed image back to an encoded image so it can be saved. You would then also want to show that this operation should run asynchronously through the appropriate method in the class.

If you need to download the file instantly and not wait for the image to appear before download initiates you can explicitly invoke the `fetch()` method which will asynchronously fetch the image from the network. Notice that the downloading will still take time so the placeholder is still required.

Authenticated image URLs — `RequestDecorator`

The standard `createToStorage` path doesn't expose the underlying `ConnectionRequest`, so attaching an `Authorization` header (or any other custom header / cookie / timeout) requires the `RequestDecorator` hook. Two ways to install one:

For per-image overrides (for example one endpoint needs an extra API-version header on top of the global bearer token), use the `createToStorage(placeholder, key, url, adapter, RequestDecorator)` overload. Per-instance decorators run **after** the global default so they can override or augment whatever the default set:

When a decorator is installed (global or per-call), `URLImage` skips the default `Util.downloadImageToStorage` path and builds a `ConnectionRequest` inline so the decorator can inspect

/ mutate it before it's queued. The legacy path remains the default for back-compat when no decorator is set.

Mask adapter

A `URLImage` can be created with a mask adapter to apply an effect to an image. This allows you to round downloaded images or apply any sort of masking for example: you can adapt the round mask code above as such:

URLImage in lists

The biggest problem with image download service is with lists. You decided to attack this issue at the core by integrating URLImage [https://www.codenameone.com/javadoc/com/codename1/ui/URLImage.html] support directly into GenericListCellRenderer [https://www.codenameone.com/javadoc/com/codename1/ui/list/GenericListCellRenderer.html] which means it will work with MultiList [https://www.codenameone.com/javadoc/com/codename1/ui/list/MultiList.html], List [https://www.codenameone.com/javadoc/java/util/List.html] & ContainerList [https://www.codenameone.com/javadoc/com/codename1/ui/list/ContainerList.html]. To use this support define the name of the component (name not UIID) to end with `_URLImage` and give it an icon to use as the placeholder. This is easy to do in the multilist by changing the name of icon to `icon_URLImage` then using this in the data:

Make sure you also set a "real" icon to the entry in the GUI builder or in handcoded applications. This is important since the icon will be implicitly extracted and used as the placeholder value. Everything else should be handled automatically. You can use `setDefaultAdapter` & `setAdapter` on the generic list cell renderer to install adapters for the images. The default is a scale adapter although you might change that to scale fill in the future:

The `createListEntry` method then looks like this:

```
private Map<String, Object> createListEntry(String name, String date, String coverURL) {
    Map<String, Object> entry = new HashMap<>();
    entry.put("Line1", name);
    entry.put("Line2", date);
    entry.put("icon_URLImage", coverURL);
    entry.put("icon_URLImageName", name);
    return entry;
}
```

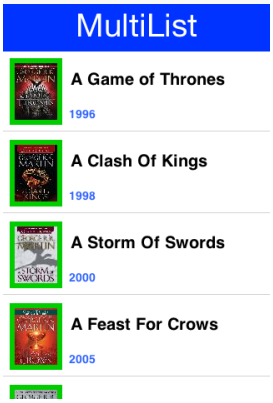


Figure 250. A URL image fetched dynamically into the list model

33.12. Charts

Codename One includes a charting toolkit in the `com.codename1.charts` package that's designed to integrate with regular UI layouts. Charts are drawn by creating an appropriate dataset and renderer pair, instantiating the matching chart view class, and wrapping it in a `ChartComponent` [https://www.codenameone.com/javadoc/com/codename1/charts/ChartComponent.html] it can be added to a form:

The following classes are available for different kinds of visualisations:

Chart class	Dataset & renderer	Notes
`BarChart` [https://www.codenameone.com/javadoc/com/codename1/charts/views/BarChart.html]	`XYMultipleSeriesDataset` / `XYMultipleSeriesRenderer`	Draws categorical data as vertical bars. The `Type` constructor parameter controls default, stacked, or heaped bars.
`BubbleChart` [https://www.codenameone.com/javadoc/com/codename1/charts/views/BubbleChart.html]	`XYMultipleSeriesDataset` containing `XYValueSeries` / `XYMultipleSeriesRenderer`	Represents each data point as a circle whose size is proportional to a third value.
`CombinedXYChart` [https://www.codenameone.com/javadoc/com/codename1/charts/views/CombinedXYChart.html]	`XYMultipleSeriesDataset` / `XYMultipleSeriesRenderer`	Combines several XY chart types in a single plot using `XYCombinedChartDef` to map series to chart implementations.
`CubicLineChart` [https://www.codenameone.com/javadoc/com/codename1/charts/views/CubicLineChart.html]	`XYMultipleSeriesDataset` / `XYMultipleSeriesRenderer`	Smooths line series with cubic interpolation. Pass a smoothness factor to the constructor to control the curve.
`DialChart` [https://www.codenameone.com/javadoc/com/codename1/charts/views/DialChart.html]	`CategorySeries` / `DialRenderer`	Renders one or more gauges on a dial, making it useful for KPI dashboards.
`DoughnutChart` [https://www.codenameone.com/javadoc/com/codename1/charts/views/DoughnutChart.html]	`MultipleCategorySeries` / `DefaultRenderer`	Shows hierarchical proportions as concentric rings around a common center.
`LineChart` [https://www.codenameone.com/javadoc/com/codename1/charts/views/LineChart.html]	`XYMultipleSeriesDataset` / `XYMultipleSeriesRenderer`	Connects series of points using straight line segments. Supports optional fill areas and point markers.

Chart class	Dataset & renderer	Notes
`PieChart` [https://www.codenameone.com/javadoc/com/codename1/charts/views/PieChart.html]	`CategorySeries` / `DefaultRenderer`	Splits a circle into slices that are proportional to each category value.
`RadarChart` [https://www.codenameone.com/javadoc/com/codename1/charts/views/RadarChart.html]	`AreaSeries` / `DefaultRenderer`	Draws a spider/web chart that compares multiple categories across the same set of axes.
`RangeBarChart` [https://www.codenameone.com/javadoc/com/codename1/charts/views/RangeBarChart.html]	`XYMultipleSeriesDataset` / `XYMultipleSeriesRenderer`	Variation of `BarChart` that uses paired min/max values to render ranges.
`RoundChart` [https://www.codenameone.com/javadoc/com/codename1/charts/views/RoundChart.html]	`CategorySeries` / `DefaultRenderer`	Abstract base class for circular charts. Use subclasses such as `PieChart`, `DoughnutChart`, or `DialChart` directly.
`ScatterChart` [https://www.codenameone.com/javadoc/com/codename1/charts/views/ScatterChart.html]	`XYMultipleSeriesDataset` / `XYMultipleSeriesRenderer`	Plots unconnected X/Y points with configurable marker shapes.
`TimeChart` [https://www.codenameone.com/javadoc/com/codename1/charts/views/TimeChart.html]	`XYMultipleSeriesDataset` / `XYMultipleSeriesRenderer`	Extends `LineChart` with date-aware labelling on the X axis for time series data.

Chapter 34. 3D Graphics and Shaders

The `com.codename1.gpu` package provides a portable, hardware accelerated 3D graphics API focused on games but useful anywhere you need GPU rendered content. It runs from one code base on the JavaSE simulator (OpenGL via JOGL), Android (OpenGL ES 2), iOS and Mac Catalyst (Metal), the native Windows port (Direct3D 11), and the JavaScript port (WebGL). The API integrates with the normal Codename One UI: a 3D scene lives inside a regular component that you add to a `Form` like any other.



Your `Renderer` callbacks run on the native platform render thread — NOT the Codename One EDT. This is the single most important thing to understand before you write any 3D code.

`onInit`, `onResize`, `onFrame` and `onDispose` are invoked on the underlying OS/GPU render loop (GL / Metal / Direct3D), not on the event dispatch thread. From inside them you must **not** touch Codename One UI components, show forms, or call other EDT-only APIs. To push data back to the UI (for example to update a HUD label or react to a game event) hop onto the EDT with `CN.callSerially(…)`. Conversely, treat the `GraphicsDevice` and the objects it hands you as owned by the render thread.



3D is a low-level, GPU dependent feature. Always guard usage with `CN.isGpuSupported()` (or `RenderView.isSupported()`); on a platform without a backend the `RenderView` reports unsupported so you can fall back to 2D.

34.1. Concepts

The API is intentionally "hybrid": a low level command layer (buffers, textures, render state, draw calls) for full control, plus high level helpers (meshes, materials, a camera) for common cases. Crucially, **you never write shader source**. Instead you describe a Material [https://www.codenameone.com/javadoc/com/codename1/gpu/Material.html] (a lighting model plus color and texture) and the engine generates the matching platform shader behind the scenes: GLSL ES on OpenGL ES and WebGL, GLSL on the desktop simulator, Metal Shading Language on iOS and Mac, and HLSL on the native Windows (Direct3D) port. This "engine-managed shader" approach is what keeps the same application code rendering identically across a range of different GPUs.

The main types are:

- RenderView [https://www.codenameone.com/javadoc/com/codename1/gpu/RenderView.html] - the `Component` that hosts the GPU surface. You give it a `Renderer`.
- Renderer [https://www.codenameone.com/javadoc/com/codename1/gpu/Renderer.html] - your callback: `onInit`, `onResize`, `onFrame`, `onDispose`. These run on the platform render thread, never the EDT.
- GraphicsDevice [https://www.codenameone.com/javadoc/com/codename1/gpu/GraphicsDevice.html] - the command surface passed to your renderer. It creates buffers and textures, clears, sets the viewport, the camera and the light, and issues `draw` calls.
- Mesh [https://www.codenameone.com/javadoc/com/codename1/gpu/Mesh.html], Material

[https://www.codenameone.com/javadoc/com/codename1/gpu/Material.html], Camera [https://www.codenameone.com/javadoc/com/codename1/gpu/Camera.html], Light [https://www.codenameone.com/javadoc/com/codename1/gpu/Light.html] - the high level scene building blocks.

34.2. A first scene: A spinning cube

The renderer below draws a Phong lit cube. `Primitives.cube` builds the geometry, a `Material` describes the surface, and a `Camera` supplies the view. Notice there is no shader code anywhere.

`setContinuous(true)` runs an animation loop. For a static scene leave it off and call `view.requestRender()` whenever something changes; this conserves battery.

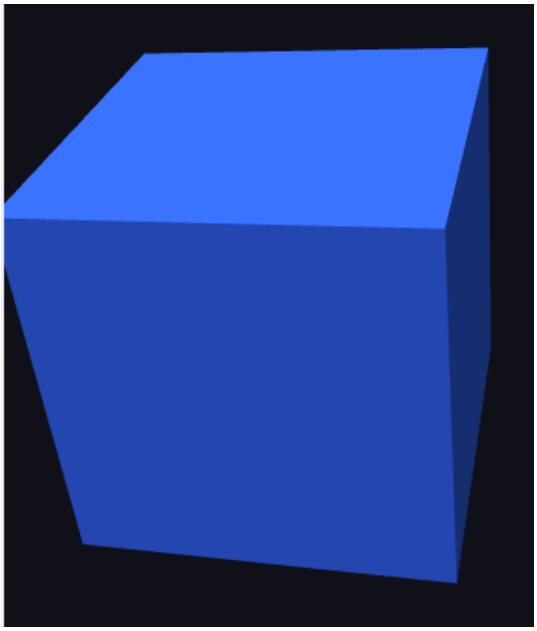


Figure 251. A Phong lit cube rendered by RenderView in the simulator

34.3. Materials

A `Material` is a declarative description of a surface. Its `Type` selects the lighting model:

Type	Description
`UNLIT`	Flat color/texture, no lighting. Ideal for UI, emissive surfaces.
`LAMBERT`	Diffuse (Lambert) lighting from one directional light.
`PHONG`	Diffuse + specular highlight (uses `setShininess`).
`SPRITE`	Unlit, for screen aligned sprites and billboards.
`SKYBOX`	Unlit background, rendered behind the scene.

A material also carries a base color (`setColor`, packed `0xAARRGGBB`), an optional `Texture` (`setTexture`), and a `RenderState` controlling depth testing, alpha blending and face culling. Textures come from a Codename One `Image` or raw ARGB pixels:

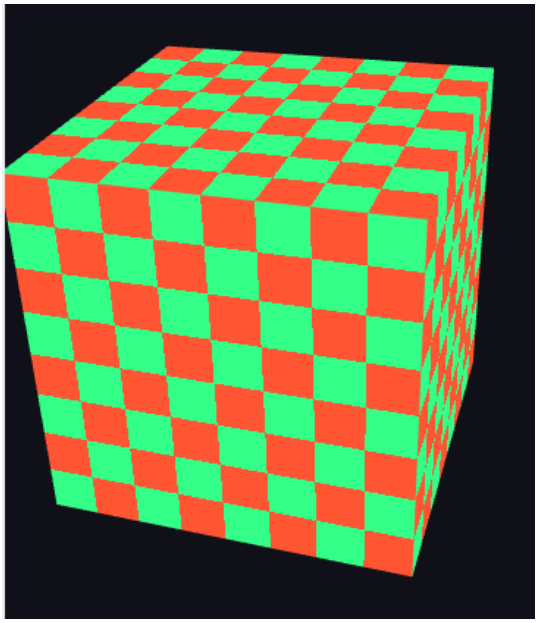


Figure 252. An UNLIT cube with a checkerboard texture (NEAREST filtering)

34.4. Meshes and buffers

`Primitives` builds common shapes (`cube`, `quad`). For custom geometry, allocate a VertexBuffer [https://www.codenameone.com/javadoc/com/codename1/gpu/VertexBuffer.html] with a `VertexFormat`, fill the interleaved float data, and (optionally) an IndexBuffer [https://www.codenameone.com/javadoc/com/codename1/gpu/IndexBuffer.html]:

```
VertexBuffer vb = device.createVertexBuffer(VertexFormat.POSITION_NORMAL_TEXCOORD, 4);
vb.setData(new float[] { /* px,py,pz, nx,ny,nz, u,v  per vertex null */ });
IndexBuffer ib = device.createIndexBuffer(6);
ib.setData(new int[] { 0, 1, 2, 0, 2, 3 });
Mesh mesh = new Mesh(vb, ib, PrimitiveType.TRIANGLES);
```

Vertex buffers are allocated through the platform SIMD allocator, which on iOS (ParparVM) places the data at a fixed, aligned native address so it can be handed to Metal with no intermediate copy. You don't need to do anything special to get this; just write into `getData()` and call `setDirty()` when you mutate it.

34.5. Loading models

Real scenes use authored geometry. `GltfLoader` loads a glTF 2.0 model, both the binary `.glb` container and the JSON `.gltf` form with embedded buffers. `loadModel` returns the mesh together with the base-color texture from the model's own material, so a textured model renders with no extra setup:

The model below is the Khronos "BoomBox" glTF sample (a CC0 model carrying its own base-color texture), loaded this way and drawn Phong lit:

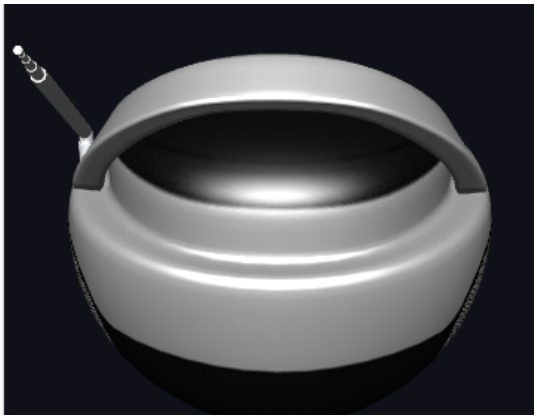


Figure 253. The Khronos BoomBox glTF sample loaded with GltfLoader

34.6. Animation

Drive a value (a rotation angle, a position) over time and redraw. A continuous `RenderView` calls `onFrame` every frame; an on-demand view redraws when you call `requestRender()`. The grid below captures six fixed rotation stages of a spinning cube, each drawn into its own viewport in a single frame:

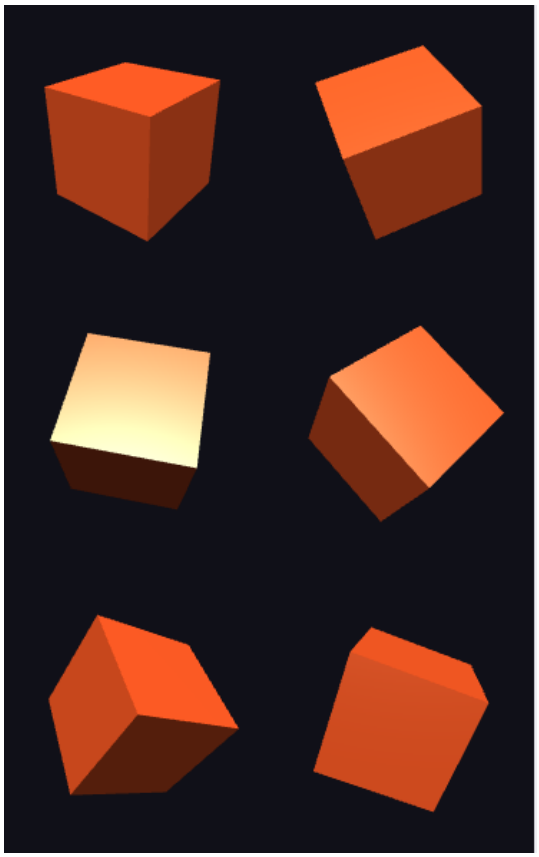


Figure 254. Six rotation stages of an animated cube captured in one frame

34.7. Camera and math

Camera [https://www.codenameone.com/javadoc/com/codename1/gpu/Camera.html] builds the view and projection matrices from an eye position, a look-at target and lens settings (`setPerspective` or `setOrthographic`). All matrix helpers live in Matrix4 [https://www.codenameone.com/javadoc/com/codename1/gpu/Matrix4.html] (column-major `float[16]`): `translation`, `scaling`, `rotation`, `multiply`,

`lookAt`, `perspective`, `ortho`. Pass a model matrix as the third argument to `draw`, or `null` for the identity.

34.8. Platform notes

- **JavaSE simulator** - OpenGL through JOGL, loaded from an isolated class so a missing or failing GL driver degrades to a built in software rasterizer instead of breaking the simulator. Lets you develop and debug 3D without a device.
- **Android** - OpenGL ES 2 via a `GLSurfaceView` hosted as a native peer.
- **iOS and Mac Catalyst** - Metal. Shaders are generated as Metal Shading Language and compiled at runtime; vertex/index data is uploaded to `MTLBuffer`s, zero-copy where the SIMD allocation permits.
- **Native Windows** - Direct3D 11. Shaders are generated as HLSL and compiled with `D3DCompile`; the offscreen render target is read back and composited into the UI.
- **JavaScript** - WebGL on a `<canvas>` peer; the generated GLSL ES runs unmodified.

Querying capabilities at runtime:

34.9. Threading

`Renderer` callbacks run on the platform render thread, not the Codename One EDT. Don't touch UI components from inside them. To move data the other way (for example to update a HUD label from a game loop), use `CN.callSerially(…)`.

Chapter 35. Augmented Reality

The `com.codename1.ar` package composites virtual 3D content into the live camera image: the session tracks the device's position in the real world, detects surfaces, recognizes registered images and faces, and keeps virtual objects locked to real-world positions as the user moves. It's backed by ARKit on iOS and by ARCore (Google Play Services for AR) on Android, behind one portable API.

Referencing any class in the package is enough to wire the build: the class scanner injects the camera permission, the `NSCameraUsageDescription` plist entry (override the text with the `ios.NSCameraUsageDescription` build hint), links ARKit and SceneKit on iOS and adds the ARCore dependency plus the `com.google.ar.core` manifest entry on Android. Apps that never touch the package pay no size, permission or store-visibility cost. By default the Android manifest marks AR `optional` so the app still installs on devices without ARCore; set the `android.ar.required=true` build hint to make the app AR-only.



Always guard AR functionality behind `AR.isSupported()`. Older devices, tvOS, watchOS, desktop builds and browsers report `false`; feature-specific hardware such as face tracking is gated through `AR.getCapabilities()`.

35.1. Concepts

Type	Role
`AR`	Static entry point: support and capability checks, permission request, session opening.
`ARSession`	The running AR experience. One session may be open at a time; it owns the view, the anchors and the detected planes, and delivers all events.
`ARView`	The component that renders the camera image composited with the anchored content. Add it to a form like any other component.
`ARPose`	An immutable position + rotation in world space: meters, right-handed, Y up, -Z forward from the initial camera direction. Converts to a `com.codename1.gpu.Matrix4` compatible matrix.
`ARPlane`	A detected real-world surface (floor, table, wall) that grows and refines as the session learns the environment.
`ARHitResult`	The intersection of a screen-point ray with real-world geometry; the bridge from a tap to a world position.
`ARAnchor`	A tracked fixed point in the real world. The attachment point for content; also delivered for recognized images (`ARImageAnchor`) and faces (`ARFaceAnchor`).
`ARNode` / `ARModel`	The content placed at an anchor: a small scene graph of nodes carrying glTF models or `com.codename1.gpu.Mesh` geometry.
`ARLightEstimate`	The real-world lighting estimate, polled per frame to shade content so it blends in.

35.2. Getting started: Place a model on the floor

The canonical AR loop is: open a session, show its view, wait for a plane, hit test a tap, anchor content at the hit.

```
if (!AR.isSupported()) {
    ToastBar.showInfoMessage("AR is not supported on this device");
    return;
}
final ARSession session = AR.open(new ARSessionOptions());
final ARView view = session.createView();

Form f = new Form("AR", new BorderLayout());
f.add(BorderLayout.CENTER, view);
f.show();

// modelBytes is a .glb asset, e.g. loaded from getResourceAsStream
view.addPointerReleasedListener(e -> {
    float xn = (e.getX() - view.getAbsoluteX()) / (float) view.getWidth();
    float yn = (e.getY() - view.getAbsoluteY()) / (float) view.getHeight();
    session.hitTest(xn, yn).ready(hits -> {
        if (hits.length > 0) {
            ARAnchor anchor = hits[0].createAnchor();
            anchor.setNode(new ARNode(ARModel.fromGltf(modelBytes)));
        }
    });
});
```

`ARSessionOptions` configures the session before it opens: the tracking mode (`WORLD` or `FACE`), which plane orientations to detect, whether to estimate lighting, and the reference images to recognize. The defaults open a world tracking session with horizontal plane detection and light estimation, which suits the place-content-on-a-surface use case above.

35.3. Plane detection

Planes are delivered through `ARPlaneListener` as immutable snapshots: an `ADDED` event when a new surface is found, `UPDATED` snapshots (sharing the same `getId()`) as the surface grows, and `REMOVED` when a plane merges into another. `ARSession.getPlanes()` always reflects the latest snapshots.

```
session.addPlaneListener(ev -> {
    ARPlane p = ev.getPlane();
    System.out.println(ev.getKind() + " " + p.getType()
            + " " + p.getExtentX() + "x" + p.getExtentZ() + "m");
});
```

The plane's local frame has X/Z spanning the surface and Y along the normal; `getCenterPose()` positions that frame in the world. `ARPlane.Type` distinguishes upward-facing horizontal surfaces

(floors, tables), downward-facing ones (ceilings) and vertical surfaces (walls) - request vertical detection with `ARSessionOptions.planeDetection(ARPlaneDetection.HORIZONTAL_AND_VERTICAL)`.

35.4. Hit testing and anchors

`hitTest(xNorm, yNorm)` shoots a ray from a normalized view coordinate into the world and resolves - on the EDT - with the intersections ordered nearest first. Prefer `ARHitResult.createAnchor()` over `session.createAnchor(pose)` when placing on detected geometry: the platform can then anchor to the exact native raycast, which tracks better as the environment refines.

Anchors survive tracking refinements: the session updates `getPose()` over time and reports the change through `ARAnchorListener`. Remove an anchor (and its content) with `detach()`. Content attaches through `setNode(ARNode)`:

```
ARNode root = new ARNode(ARModel.fromMesh(
        Primitives.sphere(0.15f, 24, 32, false), 0xffdd4433));
root.setLocalPosition(0, 0.15f, 0); // rest on the surface, not in it
anchor.setNode(root);
```

Node transforms are relative to the anchor, in meters. Nodes may nest (`addChild`), carry glTF assets (`ARModel.fromGltf`) or portable meshes (`ARModel.fromMesh`), and any mutation after attachment - position, rotation, scale, visibility, children - re-syncs the platform renderer automatically.

35.5. Light estimation

`session.getLightEstimate()` returns the latest estimate of the real-world lighting. It refreshes every frame without firing events, so poll it when rendering-relevant decisions are made. The ambient intensity is normalized so `1.0` means neutral indoor lighting; the color correction is a per-channel scale where `{1, 1, 1}` is neutral. Platform renderers already apply the estimate to anchored content; the API exposes it for custom logic such as prompting the user when the room is too dark.

35.6. Image tracking

Register printed images - posters, game boards, product labels - and the session anchors content to them when the camera sees them:

```
ARSessionOptions opts = new ARSessionOptions().referenceImages(new ARReferenceImage[]{
    new ARReferenceImage("poster", posterPngBytes, 0.42f) // physical width in meters
});
ARSession session = AR.open(opts);
session.addAnchorListener(ev -> {
    if (ev.getKind() == ARAnchorEvent.Kind.ADDED
            && ev.getAnchor() instanceof ARImageAnchor) {
        ARImageAnchor img = (ARImageAnchor) ev.getAnchor();
        System.out.println("Recognized " + img.getReferenceImageName());
        img.setNode(new ARNode(ARModel.fromGltf(overlayModel)));
    }
```

```
});
```

The physical width matters: the platform uses it to estimate the image's distance, so measure the real print. The anchor pose is centered on the physical image with local X/Z spanning its surface.

35.7. Face tracking

Face tracking is a distinct session configuration using the front camera; open it with `ARSessionOptions.trackingMode(ARTrackingMode.FACE)` after checking `AR.getCapabilities().isFaceTrackingSupported()`. Detected faces arrive as `ARFaceAnchor`s through the anchor listener, carrying the head pose plus optional finer geometry:

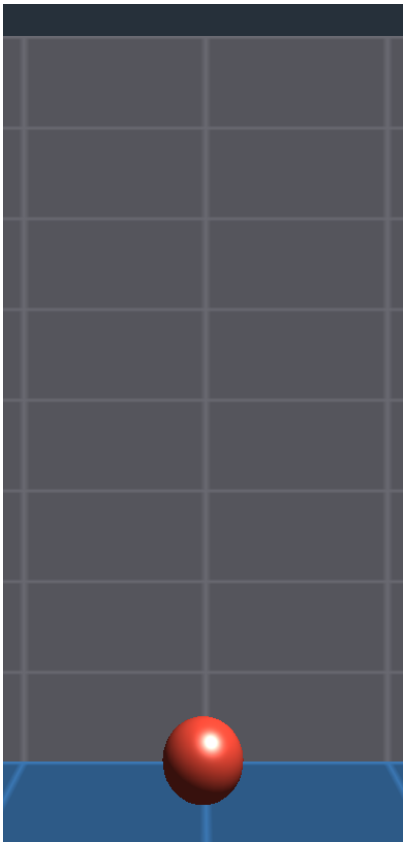
- `getRegionPose(ARFaceRegion)` returns the pose of a named region. Availability differs per platform (ARCore supplies the nose tip and forehead regions, ARKit supplies the eyes), which is why code must always handle a `null` region.
- `getMeshVertices()` / `getMeshTriangles()` expose the tracked face mesh where the platform provides one, or `null` where it doesn't.

35.8. Events and threading

Every listener fires on the EDT and every getter reflects the state as of the events delivered so far, so AR code follows normal Codename One threading rules with no extra synchronization. High-frequency refinements (anchor and plane updates, camera pose, light estimate) are coalesced: a slow EDT sees the latest values rather than a backlog. This contrasts with the `com.codename1.gpu.Renderer` callbacks used by the VR API, which run on the render thread.

35.9. The simulator and debugging

The Codename One simulator ships a simulated AR backend modeled on the Android emulator's virtual scene, so the full AR loop is debuggable with breakpoints without a device. When a session opens, the simulator "detects" the floor (and, when requested, a wall) of a virtual room after a short startup delay, mimicking real AR initialization. The AR view renders the room and every anchored model; drive the virtual device camera with the mouse (drag to look around) and keyboard (`W`/`A`/`S`/`D` or the arrow keys to move, `Q`/`E` for up and down).



The Simulate menu's **AR Simulation** window drives the paths that are hard to reproduce on demand with real hardware: force degraded tracking states and failure reasons, change the light estimate, re-run plane detection, trigger recognition of a registered reference image, and toggle a simulated face anchor for `FACE` sessions.

On devices, use the standard native tools: the generated Xcode project debugs the ARKit session and SceneKit scene directly (including Instruments), and Android Studio attaches to the Gradle project for ARCore logging. Note that Apple's iOS Simulator has no ARKit support - real ARKit behavior needs a physical device - while the Android emulator can run ARCore against its own virtual scene.

35.10. Platform notes

- **Dependencies**: injected automatically as described above. The ARCore dependency adds about 2 MB to Android binaries; iOS links the system ARKit/SceneKit frameworks at no size cost.
- **Capabilities**: face tracking requires a capable front camera (a TrueDepth camera on iOS). Check `ARCapabilities` per feature rather than assuming.
- **Face regions**: ARCore supplies `NOSE_TIP`, `FOREHEAD_LEFT` and `FOREHEAD_RIGHT`; ARKit supplies `LEFT_EYE` and `RIGHT_EYE`. Code that uses regions must handle a `null` region.
- **ARCore installation**: on Android, Play Services for AR may need installing or updating; when `AR.open` triggers that flow it throws with a message saying to retry once installation completes.
- **tvOS / watchOS**: AR compiles out entirely; `AR.isSupported()` reports `false`.

Chapter 36. Virtual Reality and 360 Media

The `com.codename1.vr` package renders application scenes in stereoscopic VR and displays 360-degree panoramas, built entirely on the portable `com.codename1.gpu` pipeline and the `com.codename1.sensors` motion sensors. There is no platform SDK dependency: everything here runs wherever `Display.getInstance().isGpuSupported()` is true, which includes the simulator, and adds nothing to the build for apps that don't use it.



Like `com.codename1.gpu.Renderer`, the `VRRenderer` and `TextureSource` callbacks run on the platform render thread that owns the GPU context - never touch Codename One UI components from them. Everything else in the package is ordinary EDT-friendly component API.

36.1. Concepts

Type	Role
`VRView`	The stereo component: clears the frame, splits the viewport per eye, positions the cameras from head tracking and invokes your renderer once per eye.
`VRRenderer`	Your scene callback: `onInit`, `onEyeFrame(device, eye, camera)` and `onDispose`.
`VREye`	Which eye a frame is for: `LEFT`, `RIGHT`, or `CENTER` in mono mode.
`VRSettings`	Interpupillary distance (default 0.064m), per-eye field of view, clip planes and the stereo default.
`HeadTracker`	Wires the gyroscope, accelerometer and magnetometer into an orientation with thread-safe snapshots for the render thread.
`OrientationFilter`	The deterministic sensor-fusion math behind the tracker, usable on its own.
`VRCameraRig`	Per-eye camera math over the existing `com.codename1.gpu.Camera`, usable without `VRView` for custom pipelines.
`Media360View`	A 360-degree panorama viewer: an equirectangular image on the inside of a sphere with drag or gyroscope look-around.
`TextureSource`	Extension point feeding dynamic textures (procedural animation, future video) to `Media360View`.

36.2. Getting started: A stereo scene

A `VRRenderer` draws the same scene for every eye; the camera differences produce the depth effect. The view configures the viewport and camera before each `onEyeFrame`, so the renderer only issues draw calls:

```
VRView vr = new VRView(new VRRenderer() {
```

```
    Mesh cube;
    Material material;

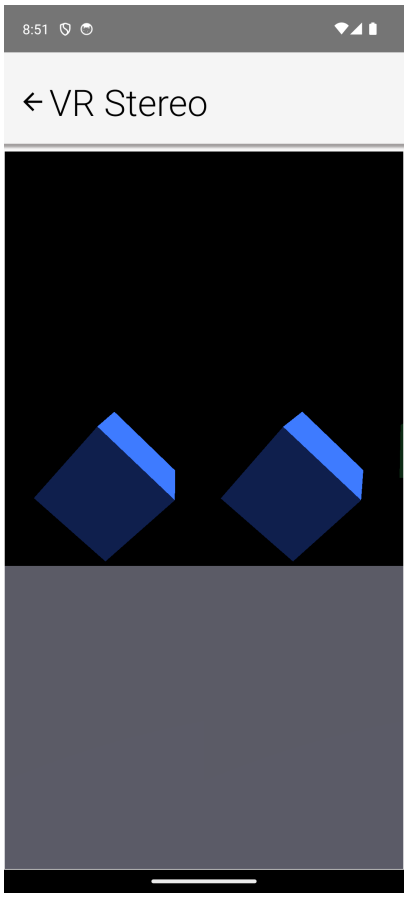
    public void onInit(GraphicsDevice device) {
        cube = Primitives.cube(device, 0.5f);
        material = new Material(Material.Type.PHONG).setColor(0xff3366ff);
    }

    public void onEyeFrame(GraphicsDevice device, VREye eye, Camera camera) {
        device.draw(cube, material, Matrix4.translation(0, 0, -2));
    }

    public void onDispose(GraphicsDevice device) { }
});
vr.setContinuous(true); // head tracked scenes want continuous rendering

Form f = new Form("VR", new BorderLayout());
f.add(BorderLayout.CENTER, vr);
f.show();
```

`setStereo(false)` switches to a single centered viewpoint (`VREye.CENTER`) at runtime, useful for a non-headset preview of the same scene. `setPosition(x, y, z)` moves the viewer through the world; `setClearColor(argb)` sets the background.



36.3. Head tracking

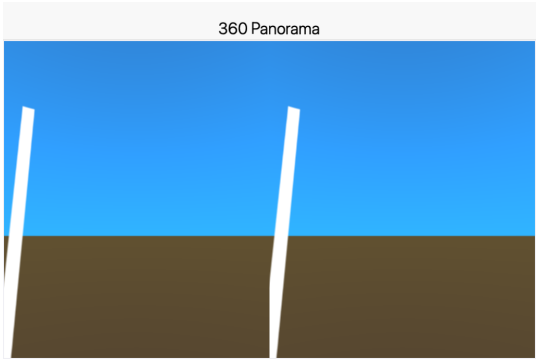
`VRView` starts its `HeadTracker` when the component appears and stops it when it leaves the screen; the sensors draw power only between the two. The tracker fuses the gyroscope with the accelerometer (tilt reference) and, when present, the magnetometer (yaw reference) through a complementary filter - tune the blend via `getHeadTracker().getFilter().setGyroWeight(float)`.

Call `recenter()` to make the current view direction the new "straight ahead," keeping pitch and roll - the standard VR recentering gesture. On devices without a gyroscope the tracker falls back to accelerometer tilt: pitch and roll stay accurate but yaw has no stable reference. `HeadTracker.isSupported()` reports whether any usable motion hardware exists; without it the orientation simply stays put and the scene still renders.

36.4. The 360 photo viewer

`Media360View` displays equirectangular panoramas - the format produced by 360 cameras and phone panorama modes, where longitude maps to X and latitude to Y:

```
Media360View pano = new Media360View();
pano.setImage(EncodedImage.create("/panorama.jpg"));
f.add(BorderLayout.CENTER, pano);
```



Dragging looks around with the grab-the-world gesture; `setHeadTrackingEnabled(true)` adds gyroscope look composed with the drag, and `setStereo(true)` renders side by side for cardboard-style viewers. Photo spheres are captured from a single point, so stereo intentionally uses zero eye separation - there is no parallax information in the image to reproduce. `setYaw`/`setPitch` position the view programmatically and `reset()` returns to straight ahead.

36.5. 360 video and dynamic content

There is no platform path from media frames to GPU textures yet, so 360 video isn't supported directly. `TextureSource` is the extension point for dynamic content: the view calls `createTexture` once on the render thread and `updateTexture` before every frame, so procedurally animated panoramas work today and a future media-to-texture bridge can plug in without API changes.

```
pano.setTextureSource(new TextureSource() {
    public Texture createTexture(GraphicsDevice device) {
```

```
        return device.createTexture(1024, 512, initialPixels);
    }
    public boolean updateTexture(GraphicsDevice device, Texture texture) {
        // mutate pixels and re-upload as needed; return true when changed
        return false;
    }
    public void dispose(GraphicsDevice device) { }
});
```

36.6. The simulator

Both `VRView` and `Media360View` run in the simulator through the desktop 3D backend, and the Simulate menu's **Motion / Gesture Simulation** window feeds synthetic sensor data to the `HeadTracker`, so head-tracked scenes are drivable from the desktop. Drag-look in `Media360View` works with the mouse as it does with a finger.

36.7. Limitations and notes

- **Lens distortion**: the stereo output is undistorted side-by-side rendering. Barrel distortion for lens-based viewers requires an offscreen render pass the portable GPU API doesn't expose yet, so there is no distortion setting to configure.
- **Battery**: continuous rendering plus live sensors is the most power-hungry mode a phone UI can run; stop or pause the view whenever it isn't visible (the component does this automatically on hide).
- **Coordinates**: the world frame matches the AR and 3D graphics APIs (meters, right-handed, Y up, -Z forward), letting content and math move between the three without conversion.

Chapter 37. Game Development

Codename One is a general purpose UI toolkit, but it ships with a package built specifically for games: `com.codename1.gaming` [https://www.codenameone.com/javadoc/com/codename1/gaming/package-summary.html]. It gives you the things games need — a tight update loop, sprite primitives, pollable input, low latency sound effects and rigid body physics — and renders on the GPU through the `com.codename1.gpu` package. Everything written here runs unchanged on every Codename One target, including iOS.



This chapter covers the real time game surface. For the "casual game" approach — building game elements out of regular `Component`s and letting the layout system render them — the techniques in the Chapter 31, Animations chapter are often a better fit. Reach for `com.codename1.gaming` when you want a frame driven loop and GPU sprite rendering.

37.1. Sample games

The samples project ships several runnable starting points. Each is a single file built entirely from art generated at runtime — no asset files — so you can copy one and start changing it immediately. The rest of this chapter draws on them to show how the pieces fit together.



`CasualGameSample`	`ScrollerGameSample`	`Gaming3DDemoSample`
A top-down arena: fly a thruster ship with the on-screen joystick, scoop up spinning gems and dodge drifting asteroids. Many sprites in a `Scene`, per-sprite animation, collision, particle bursts and a twinkling starfield.	A side-scroller: an `AnimatedSprite` runner with a walk cycle that flips to face its direction, a sun/cloud/hill parallax backdrop, hand-rolled jump physics, and a joystick + JUMP button.	A 3D world: a spinning multi-cube monument on a textured grid ground, ringed by colored `Model` buildings and billboarded coins, lit and orbited by a perspective camera.

Two more samples show how far the same 2D primitives stretch — one flat, one faked into 3D:

- `CardGameSample` — a 2D Memory (Concentration) match. Each card is a `Sprite` in a grid; flipping one animates its horizontal scale through zero and swaps the image at the thin point, and taps are hit-tested against sprite bounds through `GameInput`. A compact illustration of flat 2D play with no camera tricks at all. Dissected in Section 37.6, “Case study: The card game (flat 2D)”.
- `BoardGameSample` — **faux-3D** checkers on an isometric board. Every tile and piece is still a flat `Sprite`, but laying them out in an isometric (2:1 diamond) projection and raising the pieces off their tiles with a drop shadow gives a convincing 3D look without any GPU 3D, perspective camera or models. The screen pixel-to-board-cell mapping (and its inverse, for picking) is the same math behind any isometric strategy or tycoon game; it also implements the full move/jump/chain /crown rules and a small AI so you can play it solo. Dissected in Section 37.7, “Case study: Faux-3D checkers (isometric projection)”.

The action samples are driven by the joystick and buttons in Section 37.4, “On-screen touch controls”, so they’re equally playable on a touch device and in the desktop simulator (where the arrow keys and mouse stand in for the stick); the card and board games are pure tap.

37.2. The game loop: `GameView`

`GameView` [https://www.codenameone.com/javadoc/com/codename1/gaming/GameView.html] is the heart of the package. It’s a GPU surface (a `RenderView` [https://www.codenameone.com/javadoc/com/codename1/gpu/RenderView.html]) that runs the frame loop for you — a display link on device, the dependency-free software rasterizer in the simulator — so there’s no EDT busy loop and no frame rate to manage. You build your world by adding `Sprite`s to its `Scene`, advance the game in `update(double)`, and the view draws the scene every frame.

The `dt` passed to `update` is the wall clock time since the previous frame. Multiplying movement by it keeps the game running at the same speed regardless of the actual frame rate ("delta timing"). Notice there’s no draw method — positioning sprites is all you do; the GPU renderer handles the rest (see Section 37.5, “Sprites”).

37.2.1. Lifecycle

`start()` begins continuous rendering, `stop()` halts it, and `pause()`/`resume()` suspend `update` while the view keeps drawing the last frame. `start()` is safe to call before or after the form is shown.

`setClearColor(int argb)` sets the background the view is cleared to each frame.

37.2.2. Fixed timestep and interpolation

Variable time steps are simple but make physics nondeterministic. Call `setFixedTimestep(seconds)` to have `update` invoked at a fixed interval instead; the loop accumulates real time and may call `update` several times in one frame to catch up (capped to avoid a "spiral of death" after a long pause). The leftover fraction is available from `getInterpolationAlpha()` (0 to 1) so you can interpolate rendered positions between physics states:

37.2.3. Threading

`update(double)` runs on the render thread, together with drawing. Keep it non-blocking — offload asset loading, networking or other long work to a background thread and hand the result back with `CN.callSerially` [https://www.codenameone.com/javadoc/com/codename1/ui/CN.html#callSerially-java.lang.Runnable-].

37.3. Input: `GameInput`

Games usually want to ask "is the left key down right now?" rather than handle a stream of events. `GameInput` [https://www.codenameone.com/javadoc/com/codename1/gaming/GameInput.html], obtained from `GameView.getInput()`, captures the view's key and pointer events and exposes them as state. There are two flavors:

- **Level** state, true for as long as the input is held: `isKeyDown(int)`, `isGameKeyDown(int gameAction)`, `isPointerDown()`.
- **Edge** state, true only during the single frame the transition happened: `wasKeyPressed(int)`, `wasKeyReleased(int)`, `wasPointerPressed()`, `wasPointerReleased()`. Edges are cleared at the end of each frame, after `update` has run.

`isGameKeyDown` works with the device independent game actions `Display.GAME_UP`, `GAME_DOWN`, `GAME_LEFT`, `GAME_RIGHT` and `GAME_FIRE`, so directional input works the same across keyboards and devices. Pointer coordinates are reported relative to the `GameView's top left.

37.4. On-screen touch controls

A keyboard is fine in the simulator, but a phone has no arrow keys. Every `GameView` comes with a `TouchControls` [https://www.codenameone.com/javadoc/com/codename1/gaming/TouchControls.html] (from `getControls()`) that draws an on-screen **virtual joystick** and **buttons** and feeds them into the same `GameInput` — so the input code you already wrote for the keyboard works untouched on touch devices.

- The **joystick** reports analog direction through `GameInput#getAxisX()` / `getAxisY()` (each -1 to 1), and at the same time presses the digital game actions `GAME_LEFT` / `GAME_RIGHT` / `GAME_UP` / `GAME_DOWN` once it crosses its dead zone, so `isGameKeyDown(…)` keeps working.
- A **button** holds down a key code while it's pressed, so `isKeyDown(int)` and `wasKeyPressed(int)` report it exactly like a hardware key. Map it to a game action with `Display.getInstance().getKeyCode(Display.GAME_FIRE)`, or to any key code of your own.

Anchor controls to a corner of the view's **safe area** with the alignment constants (`TouchControls.LEFT` / `CENTER` / `RIGHT` and `TOP` / `CENTER` / `BOTTOM`) and a margin. The framework keeps them clear of notches and home indicators and repositions them automatically when the view resizes or the device rotates — you never recompute coordinates:

The joystick (bottom-left) and JUMP button (bottom-right) are visible in the sample screenshots above. There are also absolutely positioned overloads (`addJoystick(centerX, centerY, radius)`), and `setVisible(false)` hides the overlay — handy when a real keyboard is attached. Multi-touch is supported, so the player can steer and press a button at once.

37.5. Sprites

A `Sprite` [https://www.codenameone.com/javadoc/com/codename1/gaming/Sprite.html] is a lightweight data holder: an image plus position, rotation, scale, a normalized anchor and an ARGB tint. It describes **what** and **where**; a `SpriteRenderer` [https://www.codenameone.com/javadoc/com/codename1/gaming/SpriteRenderer.html] turns it into a GPU textured quad each frame using the `com.codename1.gpu` package — an orthographic camera mapping one world unit to one pixel, a `com.codename1.gpu.Material.Type#SPRITE` material and alpha blending. Because a sprite never touches the GPU directly you create one with just an `com.codename1.ui.Image`; the renderer uploads and caches the matching texture on demand.

`getX()`/`getY()` is the location of the anchor point (the image center by default), and rotation and scale pivot around that anchor.

`getBounds()` returns the axis aligned bounding box and `intersects(Sprite)` does a quick box overlap test, handy for broad phase collision before you involve physics.

37.5.1. Scenes

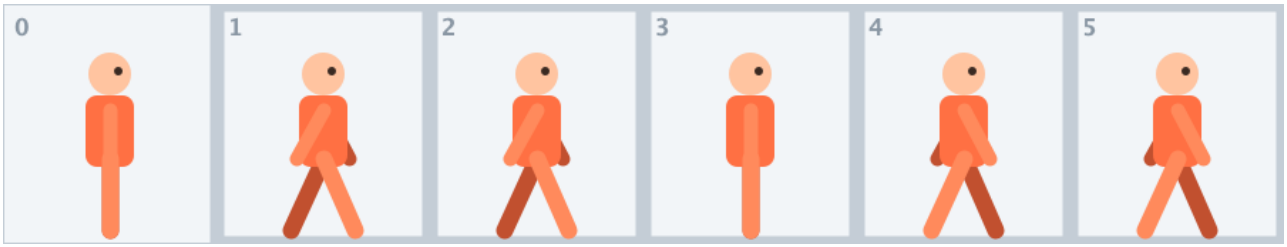
A `Scene` [https://www.codenameone.com/javadoc/com/codename1/gaming/Scene.html] is a z-ordered collection of sprites with an optional camera offset, drawn for you by the renderer. Sprites are drawn from lowest to highest `zOrder`, so higher z-order sprites appear on top; the camera offset (`setCamera(int, int)`) scrolls the whole scene. `GameView.getScene()` is the scene the view draws.

You don't need a `GameView` to render sprites: a `SpriteRenderer` is itself a `com.codename1.gpu.Renderer`, so you can host one in a plain `RenderView`.

37.5.2. Sprite sheets and animation

`SpriteSheet` [https://www.codenameone.com/javadoc/com/codename1/gaming/SpriteSheet.html] slices one atlas image into a grid of equally sized frames, cutting and caching each frame on first use (cutting a sub image copies pixels, so caching matters).

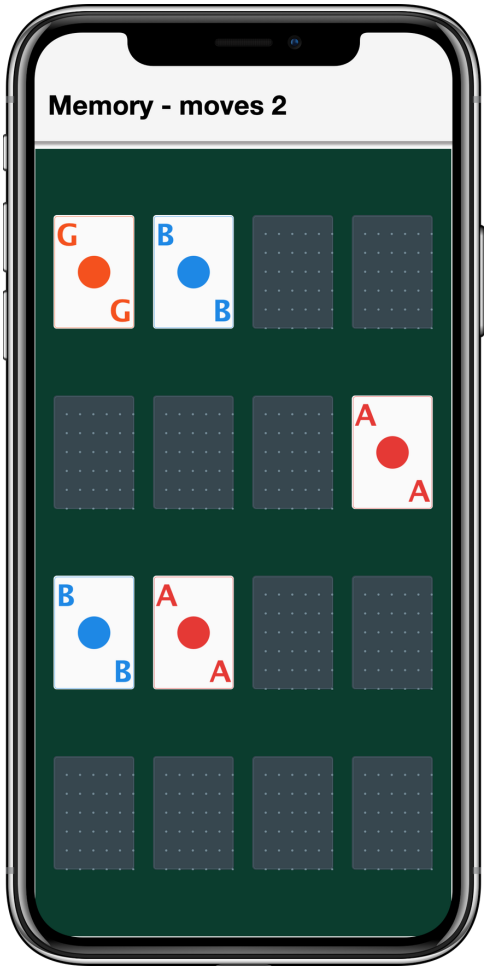
`AnimatedSprite` [https://www.codenameone.com/javadoc/com/codename1/gaming/AnimatedSprite.html] is a `Sprite` that cycles through a sequence of frames over time; the scene advances it every frame, so adding it to the scene is all you need.



A sprite sheet is just a strip (or grid) of frames like the run cycle above: the arms and legs swing a little further each frame, and playing them in sequence reads as motion. `AnimatedSprite` also accepts a plain `com.codename1.ui.Image[]` when you build the frames yourself, which is what `ScrollerGameSample's runner does:

Setting a **negative scale** flips the sprite horizontally, so a single set of frames faces both ways — no mirrored art needed (sprites are drawn double-sided for exactly this reason). `play()`, `pause()`, `stop()`, `setLooping(boolean)` and `setCurrentFrame(int)` control playback.

37.6. Case study: The card game (flat 2D)



`CardGameSample` is Memory (Concentration): sixteen face-down cards in a four-by-four grid, tap two and they stay up when they match, flip back when they don't. It's the smallest complete example of turn-based 2D play with `com.codename1.gaming` — no joystick, no physics, no camera — which makes the three techniques it does use easy to see in isolation.

One sprite per card. Each card is a `Sprite` positioned at its grid slot. There are exactly two images at any moment: the shared card-back image and the card's own face (all generated at runtime with

`Image.createImage` and `Graphics` calls, so the sample needs no assets). Changing what a card shows is just `sprite.setImage(…)`.

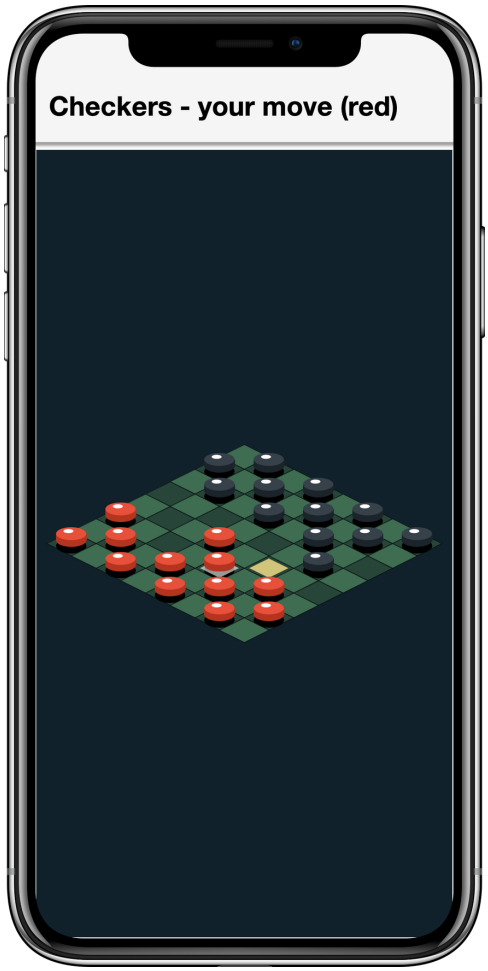
A flip you can fake with scale. A real card flip is a 3D rotation, but squeezing the sprite's horizontal scale through zero and swapping the image at the thin point reads exactly the same to the eye:

This is the same negative/positive scale trick the scroller uses to flip its runner, driven over time to play as an animation.

Polled taps against sprite bounds. There are no buttons to wire up; `update(double)` polls the `GameInput` edge state and hit-tests the tap against each card's bounding box:

The rest of the sample is a small state machine living in plain fields: the first and second face-up cards, a `mismatchTimer` that counts down in `update` before flipping a failed pair back, and a moves counter surfaced through the form title. Note the timer: in a game loop you don't schedule callbacks, you subtract `dt` from a countdown field each frame and act when it crosses zero.

37.7. Case study: Faux-3D checkers (isometric projection)



`BoardGameSample` plays a full game of checkers — moves, forced captures, chained jumps, crowning, and a small AI opponent — on a board that looks convincingly 3D — yet there is no perspective camera and no `Model` anywhere: every tile and piece is a flat `Sprite`. The 3D impression comes

entirely from **where** the flat art is placed, a technique (isometric or "2.5D" projection) that powered decades of strategy and tycoon games and still works just as well on a GPU sprite pipeline.

The projection. Board cell `(r, c)` maps to screen pixels by laying the grid out as 2:1 diamonds — columns step right-and-down, rows step left-and-down:

Picking is the inverse. Because the mapping is linear it inverts with two divisions, turning a tap back into a board cell — no per-tile hit testing required:

Depth from z-order. Cells further down the screen must draw over the cells behind them, and within one cell the stacking is tile, then shadow, then selection/move markers, then the piece. Both rules collapse into a single number — `(r + c)` strides the cells from back to front, and a small offset layers the sprites within a cell:

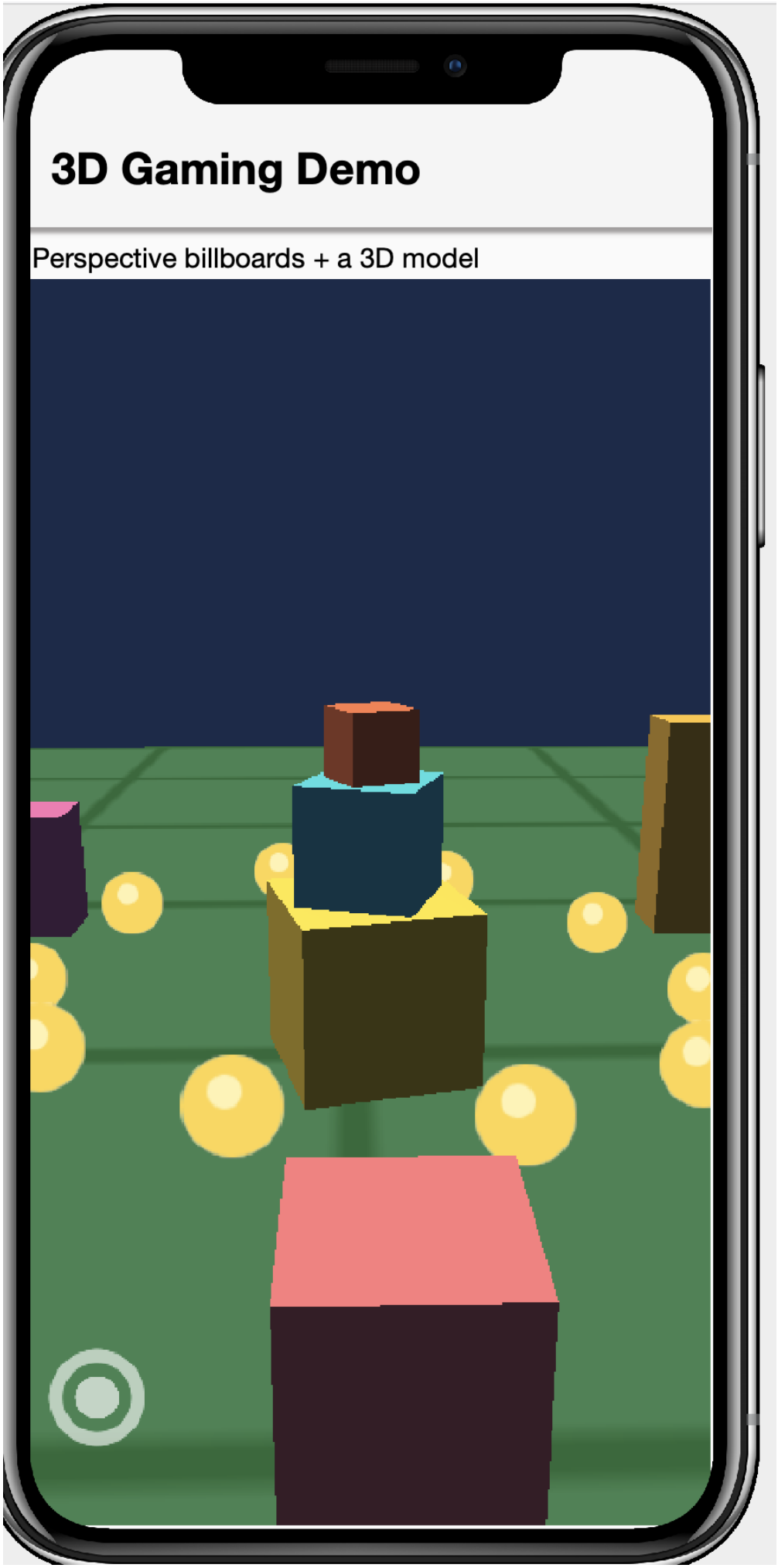
That pair of lines is also the whole "3D height" illusion: the piece is drawn raised off its tile while its shadow stays put, and the eye reads the gap as elevation. The piece art helps — an elliptical top face over a darker side rim, again just `fillArc` calls on a runtime-generated image.

Rebuild, don't mutate. After every move the sample throws away all its dynamic sprites (pieces, shadows, markers) and recreates them from the `board[][]` array in one `syncPieces()` pass. With a few dozen sprites this costs nothing, and it keeps the scene a pure function of the game state — there is no way for the display to drift out of sync with the rules, which eliminates the whole class of "the board shows X but the game thinks Y" bugs. The game logic itself (`destinations`, forced captures, chaining, crowning) never touches a sprite; it reads and writes the `int[][]` board only.

When fake 3D stops being enough — you want the camera to move through the world rather than look at it — the next section covers the real thing.

37.8. 3D and perspective

`GameView` renders on the GPU through the `com.codename1.gpu` package, so the same scene can be shown in 3D. Every view has a `GameCamera` [https://www.codenameone.com/javadoc/com/codename1/gaming/GameCamera.html] that starts in 2D mode; switching it to perspective turns flat sprites into camera-facing billboards in a 3D world and lets you draw 3D meshes alongside them. That's the basis for deeper gameplay modes — an over-the-shoulder racer, an isometric or top-down view with real depth, a 2.5D shooter.


3D Gaming Demo
Perspective billboards + a 3D model

`Gaming3DDemoSample` (above) is built entirely from the primitives this section describes — a textured ground plane, lit cube `Model` meshes, and billboarded `Sprite` coins and trees, all orbited by a perspective camera.

37.8.1. Perspective camera and billboard sprites

Call `GameCamera#setPerspective(float, float, float)` and position the camera with `setPosition` / `setTarget`. Once in perspective mode a `Sprite` is placed in world space with `Sprite#setPosition(double, double, double)` and is drawn as a billboard that always faces the camera, so your existing 2D art keeps working in 3D:

World coordinates are right-handed with y up (the `com.codename1.gpu` convention), and a sprite's size is world units rather than pixels — use `Sprite#setSize(float, float)` or `Sprite#setScale(float)` to pick a world-space size. Move the camera each frame in `update(double)` to follow the player.

37.8.2. 3D models

`Model` [https://www.codenameone.com/javadoc/com/codename1/gaming/Model.html] draws a real 3D mesh — a `com.codename1.gpu.Material` plus a position, rotation, and scale. Because GPU meshes need the `com.codename1.gpu.GraphicsDevice`, build them in `GameView#onSetup(com.codename1.gpu.GraphicsDevice)`, the render-thread hook that runs once before the first frame:

`onSetup` is also where you load real assets with `com.codename1.gpu.GltfLoader` (glTF `.glb`/`.gltf` models). Models are drawn opaque and depth-written before the alpha-blended billboards, and the billboards depth-test against them, so closer geometry correctly hides what's behind it. `GameView#getLight()` controls the directional light that shades lit material types.

37.8.3. Meshes and textures

The built-in meshes are `com.codename1.gpu.Primitives#cube(GraphicsDevice, float)` and `Primitives#quad(GraphicsDevice, float)`; a quad rotated flat makes a ground or a wall. `Model#setScale(float, float, float)` reshapes one cube mesh into crates, pillars or buildings of any proportion, so a whole skyline can share a single mesh. Lit materials multiply their base color by a texture, so you can detail a surface — a grid on the ground for depth, say — by drawing an `com.codename1.ui.Image` and uploading it with `com.codename1.gpu.GraphicsDevice#createTexture(com.codename1.ui.Image)`:

For richer geometry, load glTF (`.glb`/`.gltf`) assets with `com.codename1.gpu.GltfLoader` from `onSetup`.

37.9. Low latency audio: `SoundPool`

Music and video use the regular `MediaManager`, but rapid overlapping sound effects — gunshots, coins, footsteps — need a different tool. `SoundPool` [https://www.codenameone.com/javadoc/com/codename1/gaming/SoundPool.html] loads short clips once and triggers them with minimal latency, mixing several at the same time.

`play` returns a voice id (or `-1` if the pool is momentarily exhausted — it never blocks on the hot path). The parameters are: `volume` 0 to 1, `pan` -1 (left) to 1 (right), `rate` playback speed/pitch (1.0 is

normal), and `loop` (0 once, -1 forever, n extra repeats). `stop`, `pause`, `resume`, `stopAll`, `autoPause` and `autoResume` manage playback; `release()` frees the pool.

On platforms with a purpose built low latency audio engine — Android's `SoundPool`, iOS `AVAudioEngine`, the desktop `javax.sound.sampled` mixer and `WebAudio` in the browser — the pool uses it directly with full volume/pan/pitch support. Where none exists it falls back to a `MediaManager` based pool that still works everywhere but has higher latency and ignores pan and rate. `isNativeAccelerated()` tells you which path is active.

To react as a sound finishes — chaining clips, recycling a game object when its effect ends — register a `VoiceListener` [https://www.codenameone.com/javadoc/com/codename1/gaming/VoiceListener.html]. It's called on the EDT with the voice id as each voice completes. Not every native engine can report completion, so guard with `isVoiceCompletionSupported()`:



Load your sounds once, up front — ideally on a background thread with `SoundPool.loadAsync(…)` — and keep the `SoundEffect` references for the life of the game. Decoding on the fly defeats the purpose.

37.10. Physics

The `com.codename1.gaming.physics` [https://www.codenameone.com/javadoc/com/codename1/gaming/physics/package-summary.html] package adds 2D rigid body physics. It's an idiomatic wrapper around a pure Java port of the well known Box2D engine, so it runs on every platform — including iOS, where it's translated to C with no native code.

37.10.1. Units: Pixels, meters and the y-axis

Box2D is tuned for objects a few meters across, so the world works in **meters** internally while exposing **pixels** to you. The conversion is governed by `setPixelsPerMeter(float)` (default 30). The wrapper also flips the y-axis — Box2D points y up, the screen points y down — so your code stays in screen coordinates. The practical consequence: a positive gravity y pulls bodies **down** the screen.

37.10.2. Worlds and bodies

Create a `PhysicsWorld` [https://www.codenameone.com/javadoc/com/codename1/gaming/physics/PhysicsWorld.html], populate it with `PhysicsBody` [https://www.codenameone.com/javadoc/com/codename1/gaming/physics/PhysicsBody.html] objects, and step it once per frame from your `update`:

Bodies come in three kinds, the `BodyType` [https://www.codenameone.com/javadoc/com/codename1/gaming/physics/BodyType.html] values: `STATIC` (immovable — ground, walls), `KINEMATIC` (moved by you via velocity, unaffected by forces) and `DYNAMIC` (fully simulated). Bodies can be boxes, circles or polygons (`createBox`, `createCircle`, `createPolygon`) and respond to `setLinearVelocity`, `applyForce`, `applyLinearImpulse`, `applyTorque`, `setBullet` (continuous collision for fast objects), `setFixedRotation` and more.

For an arbitrary outline, `createShape(x, y, shape, type)` takes a Codename One `com.codename1.ui.geom.Shape` — typically the same `com.codename1.ui.geom.GeneralPath` you would draw with — and builds the collision fixtures from it, flattening any Bezier curves. Each subpath

becomes a fixture (so a figure with several outlines is a compound body); a closed convex subpath of up to eight points becomes a solid polygon usable by any body type, while a concave, larger, or open subpath becomes a one-sided edge chain (ideal for static terrain). This lets the hit-shape and the drawn shape come from one piece of geometry:

37.10.3. Driving sprites from physics

A body can drive a sprite directly. `Sprite` implements `PhysicsLinkable` [https://www.codenameone.com/javadoc/com/codename1/gaming/physics/PhysicsLinkable.html], so linking the two means `world.step(…)` updates the sprite's position and rotation automatically (in pixels, screen space) — and the scene draws it there:

37.10.4. Collisions

Register a `ContactListener` [https://www.codenameone.com/javadoc/com/codename1/gaming/physics/ContactListener.html] to be told when bodies start and stop touching. Callbacks fire from inside `step(…)` — on the game loop thread — so you can read and update game state directly, but you must not create or destroy bodies during the callback; defer that until after `step` returns.

If you need a feature the wrapper doesn't expose, `PhysicsWorld.getNativeWorld()` and `PhysicsBody.getNativeBody()` give you the underlying engine objects (which work in meters, y up).

37.10.5. Joints

Joints constrain how two bodies move relative to each other — hinges, ropes, axles, welds. `PhysicsWorld` creates them in pixel coordinates and returns a `PhysicsJoint` [https://www.codenameone.com/javadoc/com/codename1/gaming/physics/PhysicsJoint.html] handle:

`createWeldJoint` and `createPrismaticJoint` (a sliding axis) round out the set.

37.10.6. Debugging physics

When sprites and bodies drift apart it helps to see the raw simulation. Call `world.debugDraw(Graphics)` to overlay every fixture, joint and contact straight onto a Codename One `Graphics` — shapes, centers of mass and bounding boxes — in pixel space, with the y-axis flip already applied. Choose what to show with `setDebugDrawFlags(shapes, joints, boundingBoxes)`, and soften the translucent shape fill with `setDebugFillAlpha(alpha)`. Because it draws to an ordinary `Graphics` you can render it over a regular Codename One component, not only a `GameView`.

37.11. Putting it together: Anatomy of a game

Every sample follows the same shape, which is the recipe for your own game:

1. **Subclass `GameView`** and, in its constructor, create your `Sprite` and `AnimatedSprite` objects and add them to `getScene()`.
2. **Allocate GPU resources in `onSetup(GraphicsDevice)`** — meshes, textures and 3D `Model` objects — because those need the device.
3. **Add controls** (a joystick and buttons) once the view has a size.

4. **Advance the world in `update(double dt)`**: read `getInput()`, move sprites or step a `PhysicsWorld`, scale all movement by `dt`, and trigger `SoundPool` effects.
5. **Show it**: add the view to a `Form` and call `start()`.

The samples project has several runnable examples to copy from:

- **`CasualGameSample`** — sprites, per-sprite animation, collision and particle bursts in a top-down arena (see Section 37.1, “Sample games”).
- **`ScrollerGameSample`** — an `AnimatedSprite` walk cycle, a parallax backdrop and a scrolling `Scene` camera.
- **`Gaming3DDemoSample`** — lit cube `Model` objects, a textured ground and billboarded sprites under a perspective camera.
- **`GamingDemoSample`** — the physics tour, built on a `PhysicsWorld` with a floor and walls, tap-to-drop balls that bounce (each `Sprite` linked to its body) and a `SoundPool` blip whose pitch varies per drop.
- **`CardGameSample`** — a flat 2D Memory match: a grid of card `Sprite` objects with a flip animation and tap hit-testing.
- **`BoardGameSample`** — faux-3D checkers: flat sprites placed in an isometric projection, with screen-to-cell picking, full rules and a small AI.

Each is a single self-contained file that generates its art at runtime, so it needs no assets — a compact starting point you can copy and reshape.

37.12. Performance notes

- Keep `update` non-blocking — it runs on the render thread. Load assets off the render thread and hand them back with `CN.callSerially`.
- Reuse images: the renderer caches one GPU texture per `Image`, so sharing an image across many sprites uploads it once. Cache cut sprite frames (use `SpriteSheet`, which does it for you) rather than re-slicing every frame.
- Load every `SoundEffect` up front and reuse it.
- Tune `setPixelsPerMeter` so your moving bodies are on the order of a meter (tens of pixels) in size; bodies far larger or smaller than that make the simulation unstable.

The physics engine in `com.codename1.gaming.physics.box2d` is a derived work of JBox2D (a Java port of the Box2D engine), used under the BSD 2-Clause license; see the project `NOTICE` file for attribution.

Chapter 38. The Game Builder

The **Game Builder** is a visual level / map editor for the `com.codename1.gaming` API. You lay out a game world — tiles, characters, items, the camera — set per-object behavior values, preview it live, and **Build** it into your project as a data file plus a small Java companion that the runtime plays. It's to games what the GUI builder is to forms.

This chapter covers the whole feature: the underlying model, the editor UI, building a level, *defining game rules*, the supported *game types*, and the Maven workflow.

38.1. How it fits together

A game built with this tool has three layers:

Layer	Role	Who authors it
Level data (`*.game`)	A JSON description of the scene: grid, layers, placed elements + their properties, camera, lights.	The Game Builder (visual)
Companion class (`*.java`)	A thin `GameSceneView` subclass that loads the data and holds your game logic.	Generated once, then **you** edit the `onUpdate` body
Gaming engine (`com.codename1.gaming`)	Renders and runs the scene on the GPU, with pollable input, sound and optional Box2D physics.	The framework

The key idea: **the editor produces data, your companion adds rules, the engine runs it.** Because the level is data you can re-edit it any time without touching your code, and your `onUpdate` logic is preserved across regeneration.

38.1.1. The level model — `com.codename1.gaming.level`

Class	Purpose
`GameLevel`	The saveable level: `getMode()` (`MODE_2D` / `MODE_3D` / `MODE_BOARD`), grid, `layers()`, `elements()`, camera, `lights()`, `getTerrain()`. `load(json)` / `toJson()`.
`GameElement`	A placed thing: `id`, `name`, `assetId`, `layer`, a transform and a typed *property bag* (`getInt`/`getDouble`/`getString`/`getBoolean`). It's not a `Sprite`, so the same model works in 2D and 3D.
`Layer`	A named band: `KIND_TILE` (an `assetId`-per-cell grid), `KIND_ENTITY`, `KIND_MODEL`; plus `isVisible()` / `isLocked()`.
`AssetCatalog` / `AssetPack` / `AssetDef`	Resolve an `assetId` to artwork and default properties. Packs ship as JSON.
`GameSceneView`	A turnkey `GameView` that *plays a level*: it realizes 2D/board sprites and, in 3D, sets up the perspective camera, lights and a `Model` per element. Put game logic in `onUpdate(double dt)`.

Class	Purpose
`IsoProjection`, `TileLayer`, `LevelLight`, `TerrainGrid`	Board isometric math, fast single-tileset grids, and 3D lighting / terrain data.

Loading and playing a level at runtime is three lines:

38.2. Getting started

Create a scene from a **Java 17** Codename One project:

```
mvn cn1:create-game-scene -DclassName=com.example.game.Level1 -Dmode=2d
```

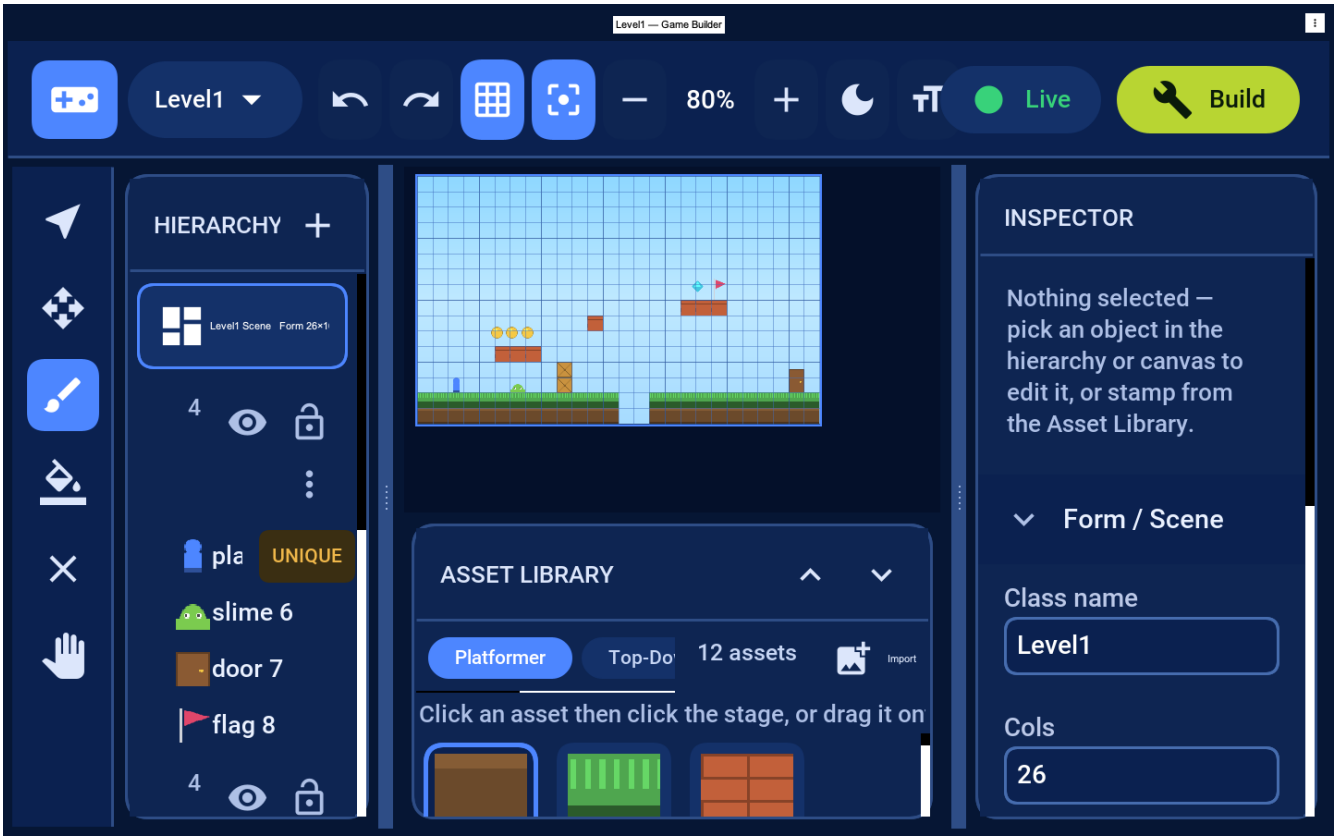
`-Dmode` is `2d` (default), `3d` or `board`. This writes `common/src/main/resources/games/Level1.game` (an empty level of that type) and `common/src/main/java/com/example/game/Level1.java` (the `GameSceneView` companion).

Open the editor bound to the project — the same way `cn1:guibuilder` launches the GUI builder:

```
mvn cn1:gamebuilder
```

The editor itself is delivered through Maven: the goal resolves the `com.codenameone:codenameone-gamebuilder` artifact from Maven Central (the same mechanism that ships the Designer / CSS-compiler jar with the plugin), so there is no separate install step. The `cn1:gamebuilder` goal only applies to Java 17 Codename One projects.

38.3. The editor at a glance



- **Top bar** — the project / scene dropdown (switch scenes, rename, or start a **new scene of any game type**), undo / redo, the **grid** and **snap** toggles, zoom, a **text-size** toggle (Small / Medium / Large), the </> button (**opens the scene's Java in your IDE**), **Live** (play in place) and **Build** (save into the project).
- **Tool rail** (left) — Select, Move, Brush, Fill, Erase, Pan.
- **Resizable panels** — drag the thin divider on the inner edge of the left panel or the inspector to resize it.
- **Hierarchy** — the scene card plus the layers. Each layer shows its item **count**, a visibility **eye** (hides the layer), a **lock** (prevents edits) and a ⋮ menu to **rename or delete** it; drag a placed object onto another layer (or above/below a sibling) to move or reorder it. Placed objects nest under their layer, and unique objects (the player) carry a **UNIQUE** badge.
- **Canvas** — the editable *stage* is the bright-bordered rectangle with the grid; painting only happens inside it.
- **Asset Library** (center-bottom) — pick a **pack** tab, then click a thumbnail to select an asset. Selecting a tile asset switches to a tile layer; selecting an actor switches to an entity layer, so the brush always works on the next click. **Import** (top-right) brings in your own artwork as a new asset.
- **Inspector** (right) — scene settings when nothing is selected, or the selected object's properties.

Because it's a desktop application, **every command is also available from the native menu bar** (drawn by the OS), grouped by convention: **File** (New 2D / 3D / Board, Open Scene, Save), **Edit** (Undo, Redo, Delete Selection), **View** (Toggle Grid / Snap, Zoom In / Out, Show Generated Code), **Tools** (Select / Move / Brush / Fill / Erase / Pan), **Layer** (Add Layer) and **Game** (Play / Stop). The standard accelerators work too — `Cmd/Ctrl+Z` undo, `Cmd/Ctrl+S` save, `Cmd/Ctrl+R` play, `Cmd/Ctrl+=` / `Cmd/Ctrl+-` zoom. Every on-screen control also has a tooltip.

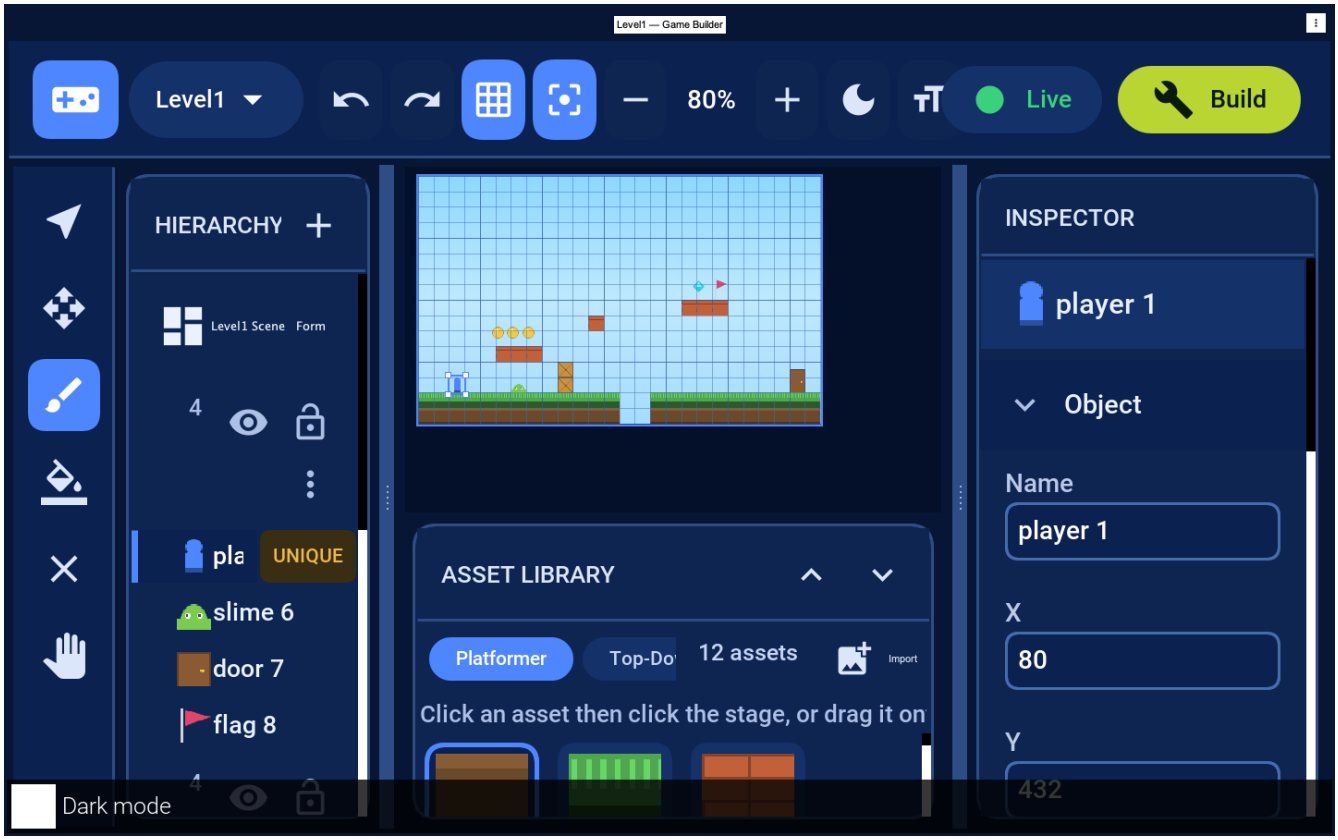
38.4. Building a level

1. **Choose a game type** — scene dropdown → *New scene* → **2D Platformer**, **3D Map** or **Board** (or scaffold with `-Dmode`).
2. **Paint terrain** — pick the **Brush**, click a tile in the Asset Library, then click/drag inside the stage. **Fill** flood-fills a region; **Erase** clears cells.
3. **Place characters and items** — click an actor asset and click on the stage to stamp it. With **Snap** on, actors land on grid cells. The **Player** is unique — stamping it again moves it.
4. **Organize with layers** — placed objects nest under their layer in the hierarchy, so you can always see which band each one is on. The **eye** hides a layer while you work (its name dims), the **lock** prevents edits to it, the chevron collapses its children, and + adds a layer. To move a selected object to a different layer (changing its draw order), use the **Layer** picker in the Inspector's *Transform* section.
5. **Undo** / **redo** every change with the toolbar arrows.
6. **Save** with **Build** — it writes `Level1.game` and, the first time, the companion `Level1.java`.

38.5. Defining game rules

Game rules live in two complementary places.

38.5.1. Data — set in the Inspector



Select an object to edit its **Transform** (name, position, **Layer** dropdown) and its **Behavior** properties — the typed values that parameterize how it acts:

Object	Behavior properties
Player / Hero	`lives`, `jumpHeight`
Coin / Gem	`value`
Enemy (slime)	`speed`, `patrol`
Door / Goal	`target` (next scene)
Torch	`lightRadius`
Card / Token / Dice	`suit`/`rank`, `player`/`cell`, `sides`

These defaults come from the **asset** — the Inspector's read-only **Asset** section shows the id, kind, size and `unique` flag, and **Edit asset defaults…** lets you change an asset's default size and properties (for example, tune `ground`/`coin`). Use **Add property…** to attach your own game metadata (`hitPoints`, `value`, …) to a single object, readable at runtime via `el.getInt(…)` / `el.getDouble(…)`. **This is the player** marks which element the preview and arrow keys drive (otherwise a `player`/`hero`/`spawn` asset is used).

Scene-wide rules are set in the Inspector when nothing is selected: **gravity**, **background** theme, grid **Cols** / **Rows** / **Tile size** and the **Mode**. These are saved into the `.game` file and are available at runtime on each `GameElement`.

38.5.2. Logic — written in the companion

The behavior *values* are data; the behavior *itself* is code you write in the generated companion's `onUpdate(double dt)`. `GameSceneView` realizes every element into a sprite whose `getUserData()` is the source `GameElement`, so you read the properties you set in the editor:

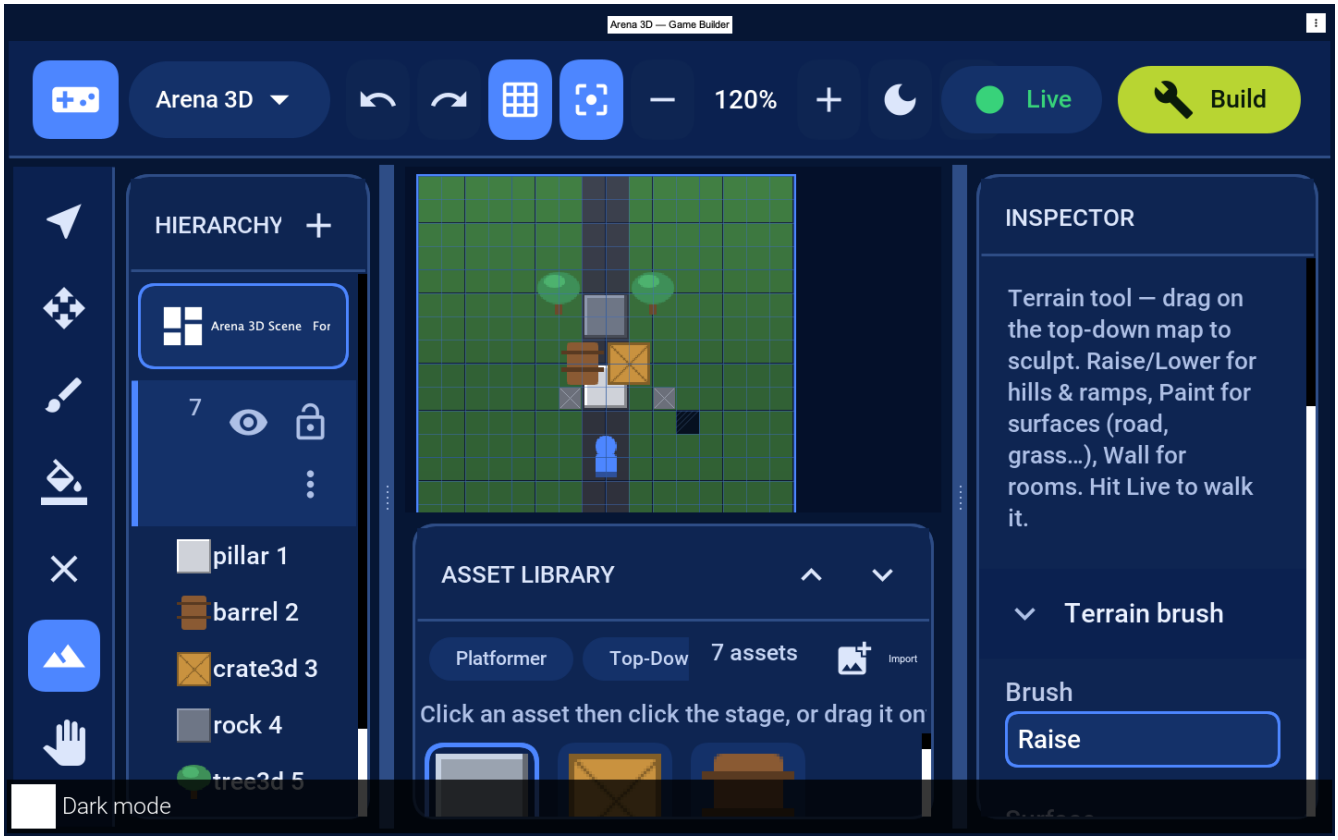
Win/lose conditions, scoring, enemy AI and level transitions (using a door's `target`) are all decided here. For rigid-body games use `com.codename1.gaming.physics` inside `onUpdate`. In short: **the editor decides the layout and the numbers; your companion decides the behavior; the engine runs it.**

38.6. Game types

The **mode** (chosen per scene) plus the **asset pack** determine how the same level model is realized:

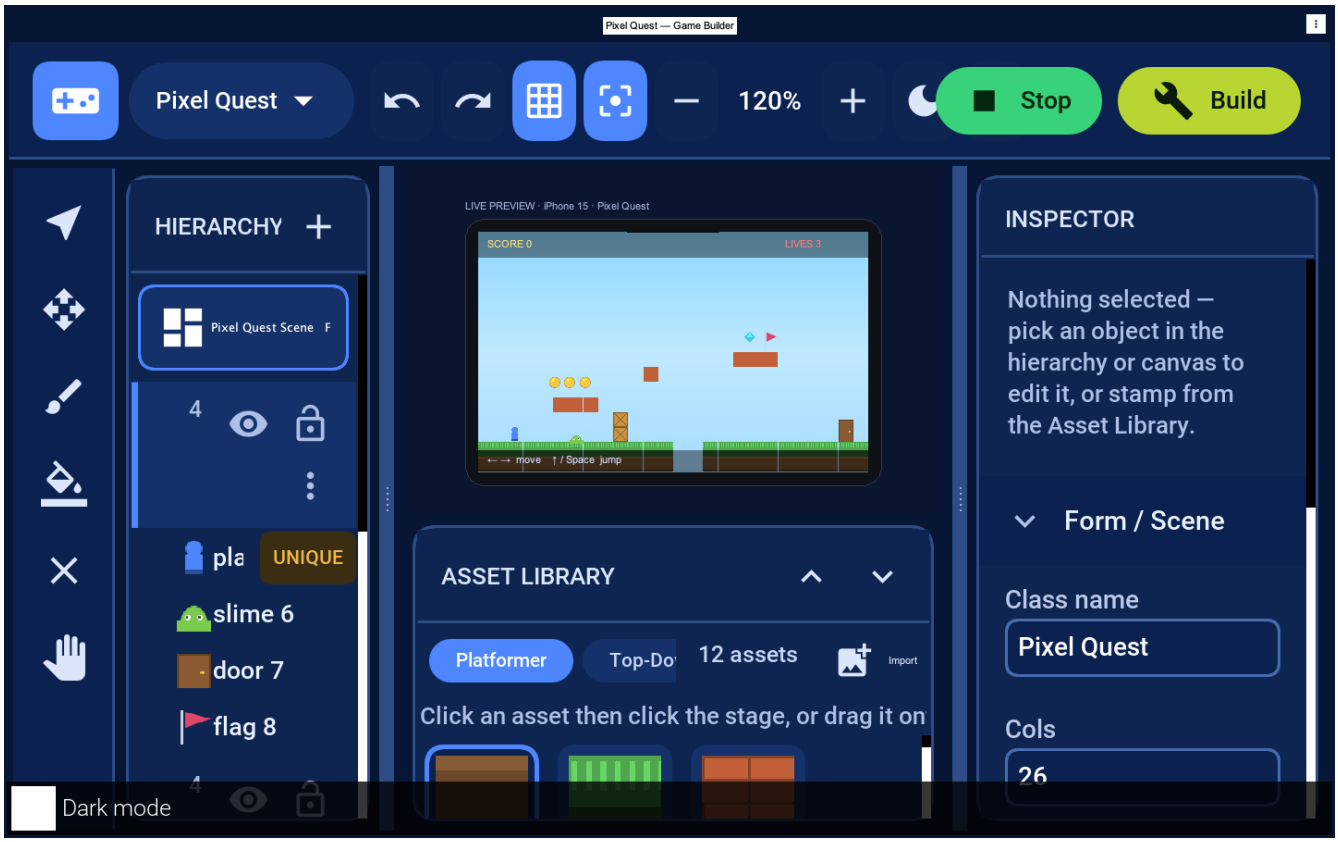
Mode	Realized as	Packs	Use for
`MODE_2D`	`Sprite`s in pixel space, tile grids	Platformer, Top-Down RPG	platformers, shooters, top-down adventures
`MODE_BOARD`	`Sprite`s placed through an isometric `IsoProjection`	Board & Card	board games, strategy, card games
`MODE_3D`	`Model`s under a perspective `GameCamera`, with `LevelLight`s and a `TerrainGrid`	3D Kit	3D maps, arena / tycoon layouts

Switch type from the scene dropdown's *New scene* buttons or scaffold with `-Dmode=2d|3d|board`. In 3D mode the Inspector also exposes the camera rig, lighting and terrain; placed elements become meshes (import your own glTF via `GltfLoader` from the companion's `buildModels` hook).



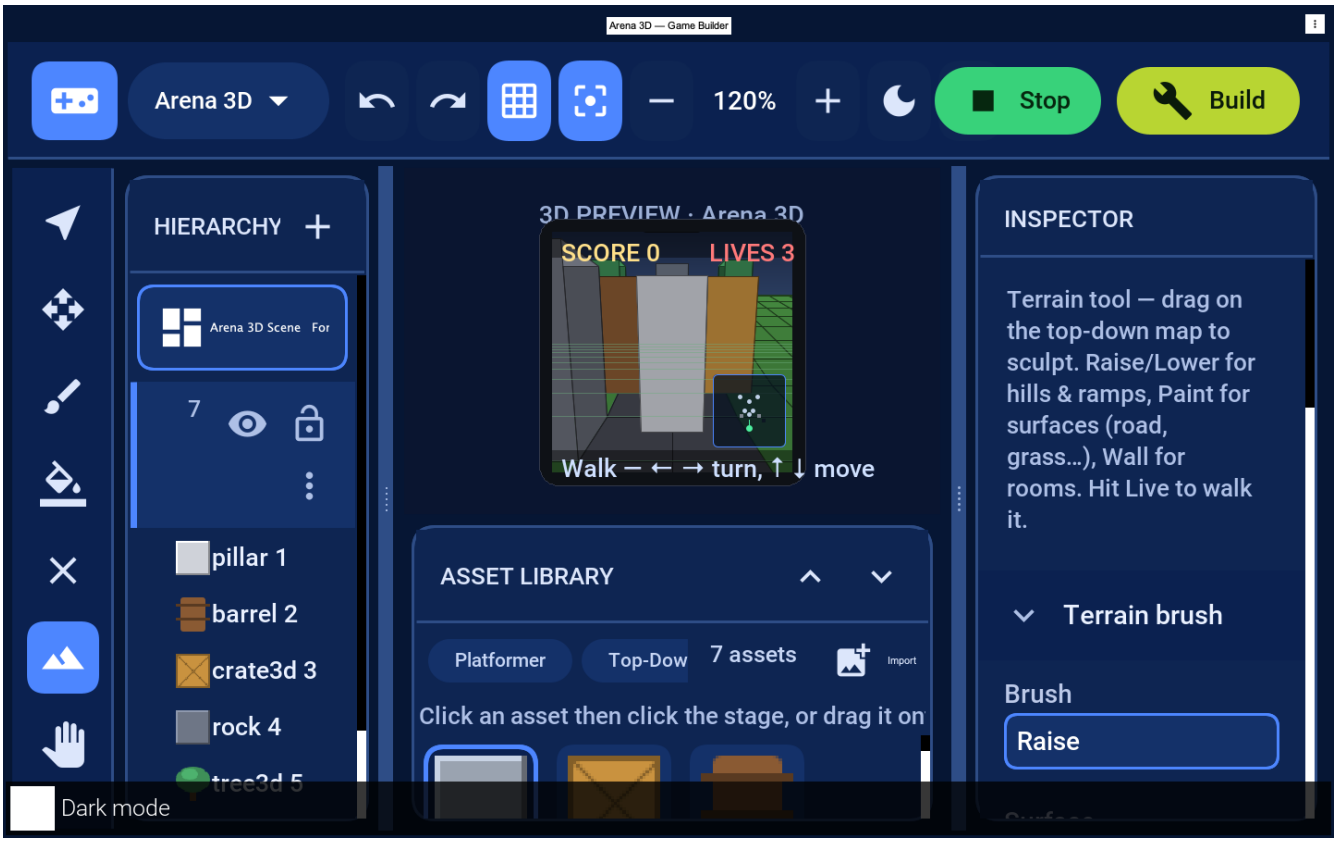
In **3D mode** the board is edited top-down (placement, terrain and lighting), and the grid autofits the canvas so it stays usable at any size and zoom. Each placed element reads as a single object on the grid rather than a flat sprite filling the board.

38.7. Live preview



Press **Live** to play the level in place inside a device frame with a HUD. The preview ships **default behaviors** so common cases work without any code: in a 2D platformer the **arrow keys** move the player and **Up** / **Space** jumps (gravity and tile collision are simulated), coins/gems are **collected for score**, and enemies **patrol** and cost a life on contact. The on-screen hint shows the controls, and the **SCORE** / **LIVES** HUD is live. Press **Stop** to return to editing — playing never mutates the level.

For a **3D map** the preview switches to a true perspective view — a receding ground grid with the placed meshes rendered as depth-sorted objects under an orbiting camera — so you see 3D gameplay rather than a flat top-down board:



The in-editor preview is for quick iteration; for full-fidelity play on device, run the companion in your app, where the scene is rendered by the GPU-accelerated `GameSceneView`.

38.8. Build, save and run

Build writes the level to `src/main/resources/games/<Scene>.game` and (once) the companion `<Scene>.java`. To show the scene in your app, instantiate the companion and `start()` it:

The level is plain data, so you can ship many `.game` files and load whichever level the player selects.

38.9. Maven goals

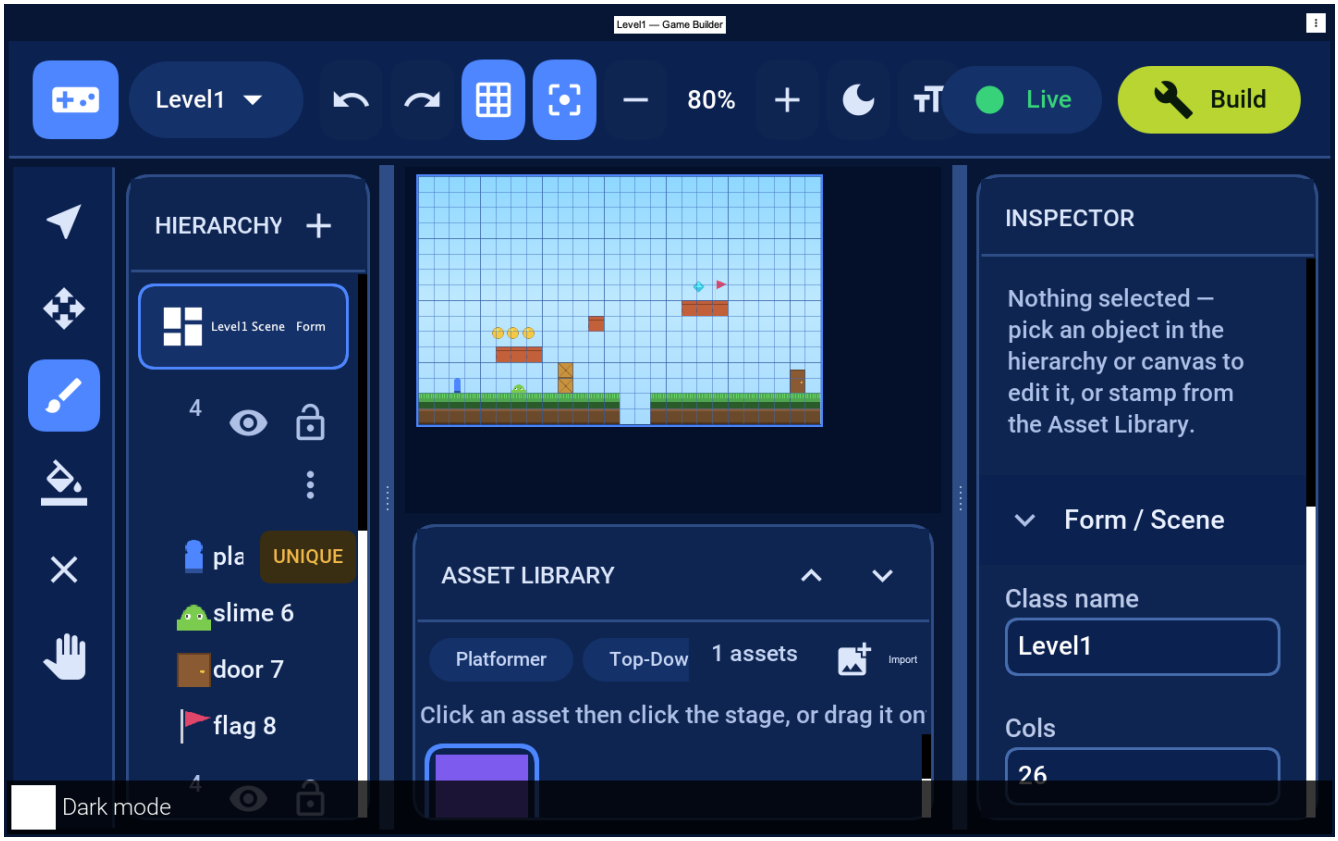
Goal	Purpose
`cn1:create-game-scene -DclassName=… -Dmode=2d|3d|board`	Scaffold a `.game` + companion (Java 17 projects only).
`cn1:gamebuilder`	Launch the editor bound to the current project.

38.10. Extending

- **Asset packs** are JSON (`{"packs":[{"id","name","assets":[{"id","kind","w","h","color","unique","source","defaults"}]}]}`) loaded by `AssetCatalog.load`. Add your own pack with real art.
- **Asset art** follows the SVG/Lottie approach: reference source art per asset and let the build translate it to code/raster; where no art is supplied the catalog draws a recognizable placeholder so a level always renders.
- **Behaviors** can be factored into reusable helpers driven entirely from the property bag, keeping levels data-only.

38.10.1. Importing your own art

To bring in custom artwork, click **Import** in the Asset Library header and pick an image.



The image is added to a **Custom** pack as a new asset (a **tile** when a tile layer is active, otherwise an **actor**), selected and ready to paint or stamp immediately. When the editor is bound to a project, the picture is copied into `src/main/resources/games/assets/` and recorded in `games/custompack.json`, so it survives reloading and ships with the project — the generated game resolves it through the same `AssetCatalog`.

Chapter 39. Game Assets

Every visible thing in a Game Builder level — the ground a player walks on, a coin to collect, an enemy, a 3D crate — is drawn from an **asset**. This chapter explains what an asset is, how assets are organized into packs and resolved by the catalog, where their artwork comes from, and how to create your own (either by authoring a pack JSON entry or by importing an image inside the editor). It closes with a note on *terrain materials*, a related-but-separate concept, and pointers for finding more art.

The classes live in `com.codename1.gaming.level`: `AssetDef`, `AssetPack` and `AssetCatalog`.

39.1. What an asset is

An asset is a **reusable template** — an `AssetDef` — not a thing in your level. It says "this is what a *coin* looks like and what default values a coin has," and any number of coins in any number of levels reference that one definition by id.

An `AssetDef` carries:

Field	Meaning
`getId()`	The stable string key (for example, `coin`, `player`, `ground`). A placed element references this id, the catalog indexes by it, and imported art is filed under it.
`getName()`	The human-readable display name shown in the editor palette (for example, `Coin`). Defaults to the id.
`getKind()`	`AssetDef.KIND_TILE` (painted into a cell of a tile layer's grid) or `AssetDef.KIND_ACTOR` (a free-form entity / model). `isTile()` is a convenience.
`getWidth()` / `getHeight()`	The natural pixel size of the art (default 32×32).
`getColor()`	A base ARGB colour used to render a flat placeholder until real artwork is available (default `0xff888888`).
`isUnique()`	Whether a level should hold at most one of these (for example, the player / spawn point).
`getSource()`	An optional reference to the source art (a resource path, an SVG/Lottie key, or the `assets/<id>.png` path written for an imported image).
`defaultProperties()`	The default authoring property values copied onto a freshly placed element — a coin's `value`, a player's `lives` and `jumpHeight`, and more.

39.1.1. Asset (template) versus placed element (instance)

Keep this distinction clear:

- An **asset** (`AssetDef`) is the shared template. There is exactly one `coin` definition.
- A **placed element** (`GameElement`) is one instance in a level. It stores its own position, size and

property values, and references the template through `GameElement#getAssetId()`.

When you drop a coin into a level, the editor creates a `GameElement` whose `assetId` is `coin` and copies the asset's `defaultProperties()` into the element's own property bag (see `StarterPacks.addActor`, which does exactly this with `el.properties().putAll(def.defaultProperties())`). You can then tweak *that one* coin's `value` without affecting the template or any other coin. Changing the asset's size or art, conversely, affects how every instance that references it's drawn.

39.2. How assets are structured

Assets are grouped into packs, and packs are indexed by a catalog:

- `AssetPack` — a named, ordered group of `AssetDef`s (`add(def)`, `get(id)`, `assets()`). The order is the order they appear in the palette. A `GameLevel` names the pack it draws from via `GameLevel#getAssetPack()`.
- `AssetCatalog` — holds one or more packs and answers two questions for any asset id: `def(id)` returns the `AssetDef` (used for kind, size and default props), and `image(id)` returns the artwork. Real art is supplied via `setImage(id, image)`; until then `image(id)` returns a cached solid-colour placeholder sized from the def and tinted with its `getColor()`, so a level always realizes to something visible.

39.2.1. The built-in packs

The editor ships four starter packs in the classpath resource `/gamebuilder-packs.json`, loaded by `StarterPacks.loadCatalog()` (which calls `AssetCatalog.load(InputStream)`). The same catalog drives both the editor palette and a generated game's runtime loader.

Pack id	Name	Asset ids
`platformer`	Platformer	`ground`, `grass`, `brick`, `crate`, `ladder`, `spike` (tiles); `coin`, `gem`, `player`, `slime`, `door`, `flag` (actors)
`topdown`	Top-Down RPG	`floor`, `wall`, `water`, `tree`, `path` (tiles); `hero`, `npc`, `chest`, `torch` (actors)
`board`	Board & Card	`boardtile`, `start` (tiles); `card`, `token`, `dice` (actors)
`kit3d`	3D Kit	`block`, `crate3d`, `barrel`, `pillar`, `tree3d`, `rock`, `spawn` (all actors)

`StarterPacks.defaultPackFor(mode)` picks `kit3d` for `MODE_3D`, `board` for `MODE_BOARD` and `platformer` otherwise.

The JSON format is `{"packs":[{"id":..,"name":..,"assets":[ … ]}]}`. Here is one real asset entry from the `platformer` pack — the player:

```
{ "id": "player", "name": "Player", "kind": "actor",
  "w": 28, "h": 32, "color": "#4D86FF", "unique": true,
  "defaults": { "lives": 3, "jumpHeight": 96 } }
```

Each JSON key maps onto an `AssetDef` field (see `AssetDef.fromMap`): `kind` is `"tile"` or `"actor"`, `w`/`h` are the pixel size, `color` is `#rrggbb`, `unique` is a boolean, `source` is the optional art reference, and `defaults`

becomes `defaultProperties()`.

39.3. Where the art comes from

An `AssetDef` names an art **format** — `TYPE_IMAGE`, `TYPE_SHEET` or `TYPE_MESH` — and a `getSource()` art file. `AssetCatalog.resolveArt()` loads each asset's art from that file into the catalog, so the same definition drives the editor palette and a shipped game.

39.3.1. Static images (`TYPE_IMAGE`)

A single `.png`/`.jpg`. `resolveArt()` decodes it; `image(id)` returns it and the level realizer wraps it in a `com.codename1.gaming.Sprite`. The starter packs ship one image file per asset (at the flat resource root, loaded as `/<id>.png`), so every starter sprite — ground, the player, the mountains — is a real file you can open in any editor and replace.

39.3.2. Sprite sheets (`TYPE_SHEET`)

One image laid out as a grid of equal frames, plus a frame size and rate (`frameW`, `frameH`, `fps`). `resolveArt()` builds a `com.codename1.gaming.SpriteSheet`; the realizer creates an animated `com.codename1.gaming.AnimatedSprite` that the scene advances each frame. The starter **coin** is a six-frame sheet, so it spins at runtime with no code; `image(id)` returns its first frame for the editor palette.

39.3.3. Meshes (`TYPE_MESH`)

A glTF/`.glb` model for a 3D level. `resolveArt()` keeps the file's bytes and `GameSceneView.buildModels` loads them with `com.codename1.gpu.GltfLoader`, realizing a `com.codename1.gaming.Model` (and falling back to a primitive cube for an asset that has no mesh). Sprite sheets and glTF loading are existing Codename One features; the asset model just makes them first-class so the editor and runtime treat them as assets.

39.3.4. Imported art (the editor's Import button)

The **Import** action opens the gallery (`Display#openGallery`) and hands the chosen image's bytes to `GameBuilder.registerImportedAsset(…)`, which builds an image `AssetDef` (kind from the active layer), adds it to a **Custom** pack and persists the PNG to `<gamesDir>/assets/` plus the pack definition to `<gamesDir>/custompack.json`. On the next launch `ProjectIO.loadCustomPack` reads them back, so an imported image survives a reload. Sprite sheets and glTF meshes are added the same way at the file level — drop the file into `<gamesDir>/assets/` and add a pack entry with the matching `type` (and frame size for a sheet); `resolveArt` then loads it like any other asset. (A dedicated sheet/mesh import UI is a natural follow-up.)

39.3.5. Generating the starter images

The bundled starter PNGs are produced once by a build-time generator (`StarterArtGenerator`, which draws each sprite with `Graphics` through `AssetArt`) and committed as resources — there's no art-drawing code in the shipped runtime. Re-run the generator after changing a starter sprite or pack.

In every case the catalog falls back to the flat-colour placeholder when an asset's art is missing, so a level is never un-renderable.

39.4. Creating your own asset

There are two practical paths.

39.4.1. Path 1: Author a pack JSON entry

Add an entry to a pack's `assets` array using the same format as the built-ins. For example, a collectible key:

```
{
  "id": "key",
  "name": "Key",
  "kind": "actor",
  "w": 24,
  "h": 24,
  "color": "#E0C040",
  "unique": false,
  "defaults": { "opens": "door", "value": 1 }
}
```

Field by field:

- `id` — unique string key your placed elements and any art will reference. Required.
- `name` — display name in the palette. Optional; defaults to the id.
- `kind` — `"tile"` to paint into grid cells, `"actor"` for a free-form entity. Anything other than `"tile"` is treated as an actor.
- `w` / `h` — natural pixel size. Optional; default 32.
- `color` — `#rrggbb` placeholder colour used until real art is supplied. Optional.
- `unique` — `true` to limit the level to one instance. Optional; default `false`.
- `defaults` — a map of default property values copied onto each placed element.
- `source` — the art file the runtime loads: a `.png`/`.jpg` (image, sheet) or a `.glb`/`.gltf` (mesh). Optional; the starter packs default it to `<id>.png`.
- `type` — `"image"`, `"sheet"` or `"mesh"`. Optional; inferred from `source` (a `.glb` is a mesh) or `frameW` (a sheet) when omitted.
- `frameW` / `frameH` / `frames` / `fps` — sprite-sheet frame size, frame count (0 = all) and playback rate, for a `"sheet"` asset.

You can add this to a copy of `gamebuilder-packs.json`, or build a pack in code with `new AssetPack(id, name).add(new AssetDef(…))` and register it with `catalog.addPack(pack)`. To give it real art, point `source` at a bundled art file and call `catalog.resolveArt()`, or supply an image directly with `catalog.setImage("key", image)`.

39.4.2. Path 2: Import an image in the editor

The no-code path: open your level in the Game Builder, select the layer you want the asset to belong to (a tile layer for tiles, an entity/actor layer otherwise), and use the **Import** action ("Import your own image as a new asset"). Pick a PNG (or other supported image), and the editor creates the `AssetDef` and files it under the **Custom** pack. Once bound to a project, the editor also saves the PNG and a pack-JSON entry under the project's `games/assets/` directory as described above. The new asset appears in the palette and is immediately placeable.

39.5. Terrain materials

3D terrain surfaces use a different, pluggable concept: **materials**, not sprite assets. A `Material` (`com.codename1.gaming.level.Material`) describes a surface — a base colour, whether it's `solid` (blocks movement), a `friction` multiplier, and an optional `artId` for textured rendering — and is referenced everywhere by its string id.

Materials are resolved through `MaterialRegistry`, a process-wide registry that ships six built-ins (`MaterialRegistry.GRASS`, `ROAD`, `STONE`, `SAND`, `WATER`, `DIRT`). `WATER` is registered as solid. Applications add their own without changing the level format:

Unknown ids resolve to a neutral gray placeholder, so a level that references a not-yet-registered material still loads. Materials cover terrain cells and `TerrainFeature`s; the sprite/model assets covered above are a separate system.

39.6. Where to get more

- **Reuse the built-in packs.** The four starter packs cover platformer, top-down RPG, board/card and a small 3D kit out of the box — start there and only add what you need.
- **Import your own art.** Any PNG (or supported sprite image) can be imported through the editor's **Import** action and becomes a first-class asset in your project's **Custom** pack. Authoring at the asset's natural pixel size keeps it crisp.
- **Author packs as JSON.** For larger sets, hand-author or generate `AssetPack` JSON entries alongside the built-ins so the whole pack loads through `AssetCatalog.load`.
- **Use external game-art sources.** Plenty of open game-art libraries and your own design tools (vector editors, pixel-art tools) can produce sprite sheets and tiles. Whichever source you use, **you are responsible for honoring its license** — check the terms for attribution and commercial-use requirements before shipping art you didn't create.

Chapter 40. Events

Most events in Codename One are routed via the high-level events (for example, action listener) but sometimes you need access to low-level events (for example, when drawing via Graphics) that provide more fine grained access. Typically working with the higher level events is far more potable since it might map to different functionality on different devices.

40.1. High-level events

High-level events are broadcast using an `addListener`/`setListener` - publish/subscribe system. Most of them are channeled via the EventDispatcher [https://www.codenameone.com/javadoc/com/codename1/ui/util/EventDispatcher.html] class which further simplifies that and makes sending events far easier.



All events are fired on the Event Dispatch Thread, the `EventDispatcher` makes sure of that.

40.1.1. Chain of events

Since all events fire on the EDT some complexities occur. E.g.:

You have two listeners monitoring the same event (or related events for example, pointer event and button click event both of which will fire when the button is touched).

When the event occurs you can run into a scenario like this:

1. First event fires
2. It shows a blocking dialog or invokes an "AndWait" API
3. Second event fires after the dialog was dismissed!

This happens because events are processed in-order per cycle. Since the old EDT cycle is stuck (because of the `Dialog`) the rest of the events within the cycle can't complete. New events are in a new EDT cycle so they can finish fine!

A workaround to this issue is to wrap the code in a `callSerially`, you shouldn't do this universally as this can create a case of shifting the problem to the next EDT cycle. But, using callSerially will allow the current cycle to flush which should help.

Another workaround for the issue is avoiding blocking calls within an event chain.

40.1.2. Action events

The most common high-level event is the ActionListener [https://www.codenameone.com/javadoc/com/codename1/ui/events/ActionListener.html] which allows binding a generic action event to pretty much anything. This is so ubiquitous in Codename One that it's even used for networking (as a base class) and for some low-level event options.

For example, you can bind an event callback for a Button [https://www.codenameone.com/javadoc/com/codename1/ui/Button.html] by using:

```
Button b = new Button("Click Me");
b.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ev) {
        // button was clicked, you can do anything you want herenull
    }
});
```

Or thanks to the Java 8 lambdas you can write it as:

```
Button b = new Button("Click Me");
b.addActionListener((ev) -> {
    // button was clicked, you can do anything you want here.
});
```

Notice that the click will work whether the button was touched using a mouse, finger or keypad shortcut seamlessly with an action listener. Many components work with action events for example, buttons, text components, slider etc.

Quite a few high-level event types exist that are more specific to requirements.

Types of action events

When an action event is fired it's given a type, but this type might change as the event evolves for example, a command triggered by a pointer event won't include details of the original pointer event.

You can get the event type from getEventType() [https://www.codenameone.com/javadoc/com/codename1/ui/events/ActionEvent.html#getEventType--], this also gives you a rather exhaustive list of the possible event types for the action event.

Modern gesture components also publish custom action types. For example, SwipeableContainer [https://www.codenameone.com/javadoc/com/codename1/ui/SwipeableContainer.html] dispatches ActionEvent.Type.Swipe [https://www.codenameone.com/javadoc/com/codename1/ui/events/ActionEvent.Type.html#Swipe-] when the top component is fully opened, allowing code to react to swipe gestures without monitoring low-level drags. Listening for these higher level events keeps gesture handling portable across touch and desktop targets.

Source of event

`ActionEvent` has a source object, what that source is depends on the event type. For most component based events this is the component but there are some nuances.

The `getComponent()` method might not get the actual component. In case of a lead component such as MultiButton [https://www.codenameone.com/javadoc/com/codename1/components/MultiButton.html] the underlying Button [https://www.codenameone.com/javadoc/com/codename1/ui/Button.html] will be returned and not the `MultiButton` itself.

To get the component that you would logically think of as the source component use the

`getActualComponent()` method.

Event consumption

An `ActionEvent` can be consumed, once consumed it will no longer proceed down the chain of event processing. This is useful for some cases where you would like to block behavior from proceeding down the path.

For example, the event dispatch thread allows you to listen to errors on the EDT using:

```
Display.getInstance().addEdtErrorHandler((e) -> {
    Exception err = (Exception)e.getSource();
    // null
});
```

This will work great but you will still get the default error message from the EDT over that exception. To prevent the event from proceeding to the default error handling you can do this:

```
Display.getInstance().addEdtErrorHandler((e) -> {
    e.consume();
    Exception err = (Exception)e.getSource();
    // null
});
```

Notice that you can check if an event was already consumed using the `isConsumed()` method but it's pretty unlikely that you will receive a consumed event as the system will stop sending it.

NetworkEvent

NetworkEvent [https://www.codenameone.com/javadoc/com/codename1/io/NetworkEvent.html] is a subclass of `ActionEvent` that's passed to `actionPerformed` callbacks made in relation to generic network code. For example:

```
NetworkManager.getInstance().addErrorListener(new ActionListener<NetworkEvent>() {
    public void actionPerformed(NetworkEvent ev) {
        // now we have access to the methods on NetworkEvent that provide more information about the network specific
flags
    }
});
```

Or with Java 8 lambdas:

```
NetworkManager.getInstance().addErrorListener((ev) -> {
    // now we have access to the methods on NetworkEvent that provide more information about the network specific flags
});
```

The `NetworkEvent` allows the networking code to reuse the `EventDispatcher` infrastructure and to simplify event firing through the EDT. But you should notice that some code might not be

equivalent for example, you could do this to read the input stream:

```
ConnectionRequest r = new ConnectionRequest() {
    @Override
    protected void readResponse(InputStream input) throws IOException {
        // read the input stream
    }
};
```

Or you can do something similar using this code:

```
ConnectionRequest r = new ConnectionRequest();
r.addResponseListener((e) -> {
    byte[] data = (byte[])e.getMetaData();
    // work with the byte data
});
```

These seem similar but they have one important distinction. The latter code is invoked on the EDT, so if `data` is big it might slow down processing. The `ConnectionRequest` is invoked on the network thread and so can process any amount of data without slowing down the UI.

MessageEvent and the Cross-Platform message bus

Bridging to native code often means passing messages between Codename One Java and platform specific code. The cross-platform message bus provides a high-level abstraction for that by letting you post messages to the native layer and subscribe for messages that originate there. Use Display.postMessage() [https://www.codenameone.com/javadoc/com/codename1/ui/Display.html#postMessage-com.codename1.ui.events.MessageEvent-] to send an MessageEvent [https://www.codenameone.com/javadoc/com/codename1/ui/events/MessageEvent.html] to the platform, and Display.addMessageListener() [https://www.codenameone.com/javadoc/com/codename1/ui/Display.html#addMessageListener-com.codename1.ui.events.ActionListener-] to receive events that arrive from native code, JavaScript bridges, or background services. Message events include helpers like `isPromptForAudioRecorder()`, `isPromptForAudioPlayer()`, and `getPromptPromise()` that allow you to hook into permission prompts emitted by the JavaScript port so custom UI can respond to platform requests while keeping the event dispatch on the EDT. This API complements `NetworkEvent` by covering native-to-Java messaging without requiring direct low-level callbacks.

40.1.3. DataChangeListener

The DataChangedListener [https://www.codenameone.com/javadoc/com/codename1/ui/events/DataChangedListener.html] is used in several places to show that the underlying model data has changed:

- TextField [https://www.codenameone.com/javadoc/com/codename1/ui/TextField.html] - the text field provides an action listener but that "fires" when the data input is complete. `DataChangeListener` fires with every key entered and thus allows functionality such as "auto complete" and is indeed used internally in the Codename One AutoCompleteTextField.

- TableModel [https://www.codenameone.com/javadoc/com/codename1/ui/table/TableModel.html] & ListModel [https://www.codenameone.com/javadoc/com/codename1/ui/list/ListModel.html] - the model for the Table [https://www.codenameone.com/javadoc/com/codename1/ui/table/Table.html] class notifies the view that its content has changed via this event, thus allowing the UI to refresh.

An exhaustive example of search implemented using the `DataChangedListener` appears in the Toolbar section [https://www.codenameone.com/manual/components.html#Advanced-search-code].

40.1.4. FocusListener

The focus listener allows you to track the "selected" or focused component. It isn't as useful as it used to be in feature phones.

You can bind a focus listener to the `Component` itself and receive an event when it gained focus, or you can bind the listener to the `Form` and receive events for every focus change event within the hierarchy.

40.1.5. ScrollListener

ScrollListener [https://www.codenameone.com/javadoc/com/codename1/ui/events/ScrollListener.html] allows tracking scroll events so UI elements can be adapted if necessary.

Scrolling is seamless and this event isn't necessary, but if developers wish to "shrink" or "fade" an element on scrolling this interface can be used to achieve that. Notice that you should bind the scroll listener to the actual scrollable component and not to an arbitrary component.

For example, in this code from the `Flickr` demo the Toolbar [https://www.codenameone.com/javadoc/com/codename1/ui/Toolbar.html] is faded based on scroll position:

```
public class CustomToolbar extends Toolbar implements ScrollListener {
    private int alpha;

    public CustomToolbar() {
    }

    public void paintComponentBackground(Graphics g) {
        int a = g.getAlpha();
        g.setAlpha(alpha);
        super.paintComponentBackground(g);
        g.setAlpha(a);
    }

    public void scrollChanged(int scrollX, int scrollY, int oldscrollX, int oldscrollY) {
        alpha = scrollY;
        alpha = Math.max(alpha, 0);
        alpha = Math.min(alpha, 255);
    }
}
```



A better way of implementing this exact effect uses title animations illustrated here [https://www.codenameone.com/manual/components.html#title-animations-section].

40.1.6. SelectionListener

The SelectionListener [https://www.codenameone.com/javadoc/com/codename1/ui/events/SelectionListener.html] event is used to broadcast list model selection changes to the list view. Since list supports the `ActionListener` event callback its the better option since it's more coarse grained.

`SelectionListener` gets fired too often for events and that might result in a performance penalty. When running on non-touch devices list selection could be changed with the keypad and a specific fire button click would fire the action event, for those cases `SelectionListener` made a lot of sense. But, in touch devices this API isn't as useful.

40.1.7. StyleListener

StyleListener [https://www.codenameone.com/javadoc/com/codename1/ui/events/StyleListener.html] allows components to track changes to the style objects. For example, if the developer does something like:

This will trigger a style event that will eventually lead to the component being repainted. This is quite important for the component class but not an important event for general user code. It's recommended that developers don't bind a style listener.

40.1.8. Component state change events

Component instances now publish lifecycle hooks that fire when they become initialized on a form and when they're removed. You can subscribe with Component.addStateChangeListener() [https://www.codenameone.com/javadoc/com/codename1/ui/Component.html#addStateChangeListener-com.codename1.ui.events.ActionListener-] to receive ComponentStateChangeEvent [https://www.codenameone.com/javadoc/com/codename1/ui/events/ComponentStateChangeEvent.html] instances that show whether the component is transitioning to the initialized state. This is useful for running setup or teardown logic alongside focus, scroll, and selection listeners.

40.1.9. Event dispatcher

When creating your own components and objects you sometimes want to broadcast your own events, for that purpose Codename One has the EventDispatcher [https://www.codenameone.com/javadoc/com/codename1/ui/util/EventDispatcher.html] class which saves a lot of coding effort in this regard. For example, if you wish to provide an ActionListener [https://www.codenameone.com/javadoc/com/codename1/ui/events/ActionListener.html] event for components you can add this to your class:

```
private final EventDispatcher listeners = new EventDispatcher();

public void addActionListener(ActionListener a) {
    listeners.addListener(a);
}
public void removeActionListener(ActionListener a) {
    listeners.removeListener(a);
```

```
}
```

Then when you need to broadcast the event use:

40.2. Low-level events

Low-level events map to "system" events directly. Touch events are considered low-level since they might expose platform specific nuances to your code.

For example, one platform might send many events during drag while another might send a few. The high-level event handling hides those complexities but some of them trickle down into the low-level event handling.



Codename One tries to hide some complexities from the low-level events as well. But, due to the nature of the event types it's a more challenging task.

Low-level events can be bound in one of 3 ways:

- Use one of the add listener methods in Form [https://www.codenameone.com/javadoc/com/codename1/ui/Form.html] for example, `addPointerPressedListener`.
- Override one of the event callbacks on `Form`
- Override one of the event callbacks on a Component [https://www.codenameone.com/javadoc/com/codename1/ui/Component.html].



When you override event callbacks on a `Component` the `Component` in question must be focusable and have focus at that point. This can be an advantage for some use cases as it will save you the need of handling unrelated events.

Each of those has advantages and disadvantages, specifically:

- 'Form' based events and callbacks deliver pointer events in the 'Form' coordinate space.
- 'Component' based events require focus
- 'Form' based events can block existing functionality from proceeding through the event chain for example, you can avoid calling super in a form event and thus block other events from happening (for example, block a listener or component event from triggering).

Table 10. Event type map

	Listener	Override Form	Override Component
Coordinate System	Form	Form	Component
Block current functionality	Yes, just avoid super	Partially (event consume)	No

40.2.1. Low-level event types

There are two basic types of low-level events: Key and Pointer.



Key events are relevant to physical keys and won't trigger on virtual keyboard keys, to track those use a TextField [https://www.codenameone.com/javadoc/com/codename1/ui/TextField.html] with a `DataChangeListener` as mentioned above.

The pointer events (touch events) can be intercepted by overriding one or more of these methods in `Component` or `Form`. Notice that unless you want to block functionality you should probably invoke `super` when overriding:

Notice that most pointer events have a version that accepts an array as an argument, this allows for multi-touch event handling by sending all the touched coordinates. Desktop and pen-enabled devices can also trigger hover events without a press. To respond to those you can override the `pointerHover*` callbacks on `Form` or `Component`, which are invoked before a button receives focus or a drag begins on those platforms.

While you can override `longPointerPress`, there is no need. The dedicated Component.addLongPressListener() [https://www.codenameone.com/javadoc/com/codename1/ui/Component.html#addLongPressListener-com.codename1.ui.events.ActionListener-] helper wires long press detection into an action listener so you can keep gesture logic in the high-level API.

Drag lifecycles also expose a completion hook. When `Component.addDragFinishedListener()` is registered it receives ActionEvent.Type.DragFinished [https://www.codenameone.com/javadoc/com/codename1/ui/events/ActionEvent.Type.html#DragFinished-] once the framework has completed its cleanup, allowing you to reset state or trigger follow-up actions that should occur after the drag image is hidden.

40.2.2. Drag event sanitation

Drag events are quite difficult to handle across devices. Some devices send a ridiculous number of events for even the lightest touch while others send too little. It seems like too many drag events wouldn't be a problem, but if you drag over a button then it might disable the buttons action event (since this might be the user trying to scroll).

Drag sensitivity is about the component being dragged which is why you have the method `getDragRegionStatus` that allows you to "hint" to the drag API whether you're interested in drag events or not and if so in which directional bias.

For example, if your component is a painting app where you're trying to draw using drag gestures you would use code such as:

```
public class MyComponent extends Component {
    protected int getDragRegionStatus(int x, int y) {
        return DRAG_REGION_LIKELY_DRAG_XY;
    }
}
```

This indicates that you want all drag events on both AXIS to be sent as soon as possible. Notice that this doesn't disable event sanitation.

40.3. BrowserNavigationCallback

The BrowserNavigationCallback [https://www.codenameone.com/javadoc/com/codename1/ui/events/BrowserNavigationCallback.html] isn't quite an "event" but there is no real "proper" classification for it.



The callback method of this interface is invoked off the EDT! You must **NEVER** block this method and must not access UI or Codename One sensitive elements in this method!

The browser navigation callback is invoked directly from the native web component as it navigates to a new page. Because of that it's invoked on the native OS thread and gives you a unique opportunity to handle the navigation ourselves as you see fit. That's why it MUST be invoked on the native thread, since the native browser is pending on your response to that method, spanning an invokeAndBlock/callSerially would be to slow and would bog down the browser.

Chapter 41. Device Input and Form Factors

Codename One runs on a wide range of hardware, from phones and tablets to desktops, foldables, and devices driven by a mouse or a stylus. This chapter covers the cross-platform APIs that expose that hardware: rich pointer detail (mouse buttons, pen pressure and tilt, the pointing device type), the mouse wheel, foldable device posture, and multi-display / desktop-window awareness.

These APIs are additive: on a device or platform that doesn't report a given capability the value falls back to a sensible default, so a single code base keeps working everywhere.

41.1. Rich pointer events

The classic `pointerPressed`, `pointerReleased` and `pointerDragged` callbacks carry only the x/y coordinates of the touch. Modern devices provide far more detail: which mouse button was used, the kind of pointing device (finger, mouse, stylus or eraser), the pressure applied, the stylus tilt, the contact area and the keyboard modifiers held at the time.

That detail is exposed through the immutable `PointerEvent` [https://www.codenameone.com/javadoc/com/codename1/ui/events/PointerEvent.html] snapshot. There are two ways to read it.

41.1.1. From an ActionEvent

Any pointer listener receives an `ActionEvent`; call `getPointerEvent()` on it:

41.1.2. By polling

When you're inside a pointer callback you can also poll the current values directly from `CN` (or `Display`), mirroring the existing `CN.isShiftKeyDown()` family:



Values have safe defaults. On a plain touch screen the button is `BUTTON_PRIMARY`, the pressure is `1.0` and the type is `TYPE_TOUCH` (or `TYPE_UNKNOWN` on ports that don't classify the device).

41.2. Two and three button mice

`PointerEvent.getButton()` returns one of `BUTTON_PRIMARY`, `BUTTON_SECONDARY`, `BUTTON_MIDDLE`, `BUTTON_BACK` or `BUTTON_FORWARD`. `CN.getPressedButtonMask()` returns a bitmask of every button currently held (`MASK_PRIMARY`, `MASK_SECONDARY`, …), which is useful during a drag.

The real button is reported on every desktop port - macOS, Windows and Linux - and in the browser, so the right, middle and (where the mouse has them) back and forward buttons are all distinguished. On a plain touch screen there is only the primary button.

For the common case of a context menu, use the unified `addContextMenuListener`. It fires for a secondary (right) mouse click, a stylus barrel button click and a long press, so the same code handles desktop and touch:

41.3. Mouse wheel and trackpad scrolling

By default the mouse wheel and trackpad scroll a scrollable container automatically. To react to the wheel yourself - for example control plus wheel to zoom, or horizontal scrolling - add a `addMouseWheelListener`. The listener receives a `WheelEvent` [https://www.codenameone.com/javadoc/com/codename1/ui/events/WheelEvent.html]; consuming it prevents the default scroll.

A positive `getDeltaY()` reveals content above (the user scrolled down); a positive `getDeltaX()` reveals content to the left. `isPrecise()` is true for high resolution devices such as trackpads. On the desktop simulator, hold shift while turning the wheel to produce horizontal scrolling.

`WheelEvent` is a single universal scroll-gesture API: it's fed by the mouse wheel and trackpad on the desktop, by the wheel / external trackpad on Android, and by the trackpad, Magic Mouse and mouse wheel on iOS and iPadOS (captured natively and routed through the same pipeline). Continuous trackpad scrolling reports `isPrecise() == true`, while a notched wheel reports `false`. On the Apple Watch the Digital Crown also feeds this same pipeline, so a list scrolls under the crown and a mouse wheel listener receives the crown as a `WheelEvent`.

41.4. Trackpad gestures (magnify and rotate)

Beyond scrolling, the trackpad reports magnify (pinch) and rotate gestures. Override `pinch(float scale)` to zoom and `rotation(float radians)` to rotate; both return true to mark the gesture as handled. `pinch` is also driven by a two finger pinch on touch screens, so the same code works on mobile and the desktop.

The trackpad gestures are wired on every desktop port: the macOS trackpad, the Windows touch screen and gesture-capable touchpads (through the system `WM_GESTURE` messages), and the Linux touchpad (through the GTK touchpad gesture events). `scale` is an incremental zoom factor (just above or below `1.0`) and `radians` is the incremental rotation since the previous callback, so accumulate them to track the total.



A Windows precision touchpad usually reports a pinch as control plus wheel rather than a system gesture, so it also arrives through the universal wheel API. Handle that case with a `addMouseWheelListener` that checks `isControlDown()` (shown above) if you target those devices.

41.5. Stylus and pen support

When a stylus is used (Apple Pencil, S-Pen, Surface Pen) the pointer type is `TYPE_STYLUS`, or `TYPE_ERASER` for the eraser end. The `PointerEvent` then also carries:

- `getPressure()` - normalized pressure between `0.0` and `1.0`.
- `getTiltX()` / `getTiltY()` - tilt in degrees (altitude and azimuth on iOS).
- `getContactSize()` - normalized size of the contact area.

For drawing surfaces a dedicated `addStylusListener` fires on press, drag and release only when the pointer is a stylus or eraser:

Pen hover (the pen moving above the screen without touching, available on devices such as the S-Pen and the M2 iPad with Apple Pencil) flows through the existing `pointerHover` callbacks; the snapshot reports `isHovering()` as true.

Touch-enabled desktops are handled too: on a Windows or Linux PC with a touch screen the finger reports `TYPE_TOUCH`, and a Windows pen reports `TYPE_STYLUS`, so the same drawing and gesture code runs on a touch laptop as on a phone. `Display.getInstance().isTouchScreenDevice()` is true on those machines.

41.6. Foldable devices and posture

Foldable and dual screen devices (Galaxy Fold, Galaxy Flip, Pixel Fold, Surface Duo) change shape at runtime. Query the live posture through `DevicePosture` [https://www.codenameone.com/javadoc/com/codename1/ui/DevicePosture.html]:

- `getPosture()` - `POSTURE_FLAT`, `POSTURE_HALF_OPENED`, `POSTURE_CLOSED` or `POSTURE_UNKNOWN`.
- `getHingeAngle()` - hinge angle in degrees (`-1` when unknown).
- `getFoldOrientation()` - `FOLD_ORIENTATION_VERTICAL`, `FOLD_ORIENTATION_HORIZONTAL` or `FOLD_ORIENTATION_NONE`.
- `getFoldBounds(Rectangle)` - the region the hinge occludes, in display coordinates, or null when there is no separating fold. Treat it like `Form.getSafeArea()`: keep interactive content out of it, or split a master and detail layout across the two halves.

React to fold changes with a posture listener:

To exercise this in the desktop simulator, use the **Simulate → Foldable** menu: enable foldable mode, then pick a posture (flat, half-opened or closed), the fold orientation (vertical or horizontal) and the hinge angle. The running app receives the same posture changes and listener callbacks it would on a real foldable device.

Foldable support on Android is opt-in so it never adds weight to apps that don't use it. Enable it with the `android.foldableSupport=true` build hint (the version can be overridden with `android.windowVersion`). Note that the hinge **angle** isn't available on Android, so `getHingeAngle()` returns `-1` there.

41.7. Desktop windowing and external displays

Some mobile devices run in a desktop windowing mode where the app shares the screen with other windows: Samsung DeX, Android desktop windowing and iPad Stage Manager. `CN.isDesktopMode()` reports this. It's distinct from `CN.isDesktop()`, which reports a genuine desktop platform (Windows, macOS or Linux).

`CN.getDisplayCount()` returns the number of attached displays and `CN.isExternalDisplayConnected()` is a convenience for `getDisplayCount() > 1`. On the desktop port these reflect the real monitors attached to the machine.

41.8. Apple TV remote

On Apple TV the Siri Remote is delivered through the standard key handling, so the same `keyPressed`/`keyReleased` and game-key code you write for other platforms works on the TV. The directional buttons map to the arrow / game keys, the click maps to enter (fire), the menu button maps to the back key, and play/pause maps to the media play/pause key. No TV specific API is required.

Chapter 42. File system, storage, network, and parsing

42.1. JAR resources

Resources packaged within the "JAR" don't belong in this section of the developer guide, but they're often confused with `Storage` and `FileSystemStorage`, so this is a good place to clarify them.

You can place arbitrary files within the `src` directory of a Codename One project. The file is packaged into the final distribution. In standard Java SE you can do something like:

```
InputStream i = getClass().getResourceAsStream("/myFile");
```

This isn't guaranteed to work on all platforms and will probably fail on some. Instead, use something like:

```
InputStream i = Display.getInstance().getResourceAsStream(getClass(), "/myFile");
```

This isn't the limitation. You can't use hierarchies, so something like this fails:

```
InputStream i = Display.getInstance().getResourceAsStream(getClass(), "/res/myFile");
```

You can't use relative paths either, so this also fails (notice the missing first slash):

```
InputStream i = Display.getInstance().getResourceAsStream(getClass(), "myFile");
```

Those limitations exist for portability. iOS and Android handle resources differently, so supporting the full Java SE semantics is unrealistic.



This is even worse sometimes. Because of the way iOS works with resources, some file names can fail. For example, names with the `@` character or more than one `.` can cause problems (for example, `file.ext.act`).

Just like in Java SE, the entries within the "JAR" are read and can't be modified. You can access the stream, but not the actual file.

42.2. Storage

Storage [https://www.codenameone.com/javadoc/com/codename1/io/Storage.html] is accessed through the `Storage` class. It's a flat, filesystem-like interface that can list, delete, and write named storage entries.

The Storage API also provides convenient methods to write objects to Storage and read them back with readObject and writeObject.



Objects in Storage are deleted when an app is uninstalled, but they're retained between application updates. A notable exception is Android, which on some devices keeps objects in Storage after uninstalling an app. To force Android to remove them on uninstall, use the build hint android.allowBackup=false.

The sample code below demonstrates listing storage content, adding and viewing entries, and deleting entries:

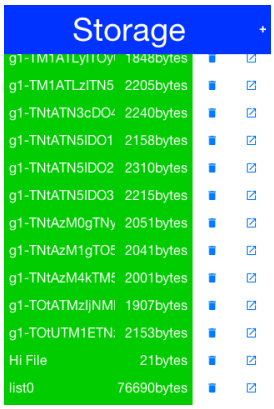


Figure 255. List of files within the storage

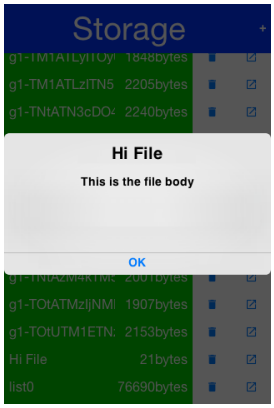


Figure 256. Content of a file added to the storage

42.2.1. The preferences API

Storage also offers a simple API in the form of the Preferences [https://www.codenameone.com/javadoc/com/codename1/io/Preferences.html] class. The Preferences class lets developers store simple variables, strings, numbers, booleans, and similar values without writing storage code. This is a common use case in applications. For example, you might need to store a server token:

You can then read the token like this:

```
String token = Preferences.get("token", null);
```

The backing store filename defaults to "codenameone.properties", but you can override it by calling Preferences.setPreferencesLocation(String) before your app touches the API. This is useful when

you need separate preference namespaces per user or when you want to plug in an encrypted storage layer.

When you have a batch of updates, prefer the `Preferences.set(Map<String,Object>)` overload so that all values are persisted with a single disk write.

Preferences can also tell interested code when a key changes. Register a `PreferenceListener` with `addPreferenceListener()` to react to updates made in other parts of your code or triggered remotely through `Codename One Push`. Remember to remove listeners you no longer need to avoid leaks.



This gets somewhat confusing with primitive numbers for example: if you use `Preferences.set("primitiveLongValue", myLongNumber)` then invoke `Preferences.get("primitiveLongValue", 0)` you might get an exception!
This would happen because the value is physically a `Long` object but you're trying to get an `Integer`. The workaround is to remain consistent and use code like this `Preferences.get("primitiveLongValue", (long)0)`.

42.3. File system

FileSystemStorage [https://www.codenameone.com/javadoc/com/codename1/io/FileSystemStorage.html] provides file system access. It maps to the underlying OS's file system API providing most of the common operations expected from a file API somewhat in the vain of `java.io.File` & `java.io.FileInputStream` for example: opening, renaming, deleting etc.

Notice that the file system API is somewhat platform specific in its behavior. All paths used the API should be absolute otherwise they aren't guaranteed to work.

The main reason `java.io.File` & `java.io.FileInputStream` weren't supported directly has a lot to do with the richness of those two APIs. They effectively allow saving a file anywhere, but mobile devices are far more restrictive and don't allow apps to see/change files that are owned by other apps.

42.3.1. File paths & app home

All paths in `FileSystemStorage` are absolute, this simplifies the issue of portability since the concept of relativity and current working directory aren't portable.

All URLs use the / as their path separator you try to enforce this behavior even on Windows.

Directories end with the / character and thus can be distinguished by their name.

The `FileSystemStorage` API provides a `getRoots()` call to list the root directories of the file system (you can then "dig in" through the `listFiles` API). For example, this is confusing and unintuitive for developers.

To simplify the process of creating/reading files Codename One provides the `getAppHomePath()` method. This method allows you to get the path to a directory where files can be stored/read.

You can use this directory to place an image to share as you did in the share sample

[https://www.codenameone.com/manual/components.html#sharebutton-section].



A common Android hack is to write files to the SDCard storage to share them among apps. Android 4.x disabled the ability to write to arbitrary directories on the SDCard even when the appropriate permission was requested.

42.3.2. Native file chooser

Use `Display.openFileChooser()` or the shorter `CN.openFileChooser()` when the user needs to select a general document such as a PDF, text file, private key, or import file. This is different from `Display.openGallery()`, which is intended for photo and video library media.

The callback source is a `String` path that can be read with `FileSystemStorage.openInputStream()`. If the user cancels the picker the event source is `null`.

```
CN.openFileChooser(evt -> {
    if(evt == null || evt.getSource() == null) {
        return;
    }
    String path = (String)evt.getSource();
    try(InputStream in = FileSystemStorage.getInstance().openInputStream(path)) {
        String text = Util.readToString(in, "UTF-8");
        // import the selected file
    } catch(IOException err) {
        Log.e(err);
    }
}, "pdf,txt");
```

The second argument is a comma-separated accept list. It can contain filename extensions such as `pdf,txt,p8`, MIME types such as `application/pdf`, or `null`// for any file. JavaSE uses the native AWT file dialog. Android uses the system document picker and copies content URIs into an app-readable file path when needed. Ports without a native document picker fall back to a Codename One file tree.

No photo-library or media permissions are added by this API. On Android the system document picker grants read access to the selected document. On iOS this API should be implemented with a document picker, not the photo library; `ios.NSPhotoLibraryUsageDescription` is only needed for `openGallery()`.

A more advanced usage of the `FileSystemStorage` API can be a `FileSystemStorage Tree`:

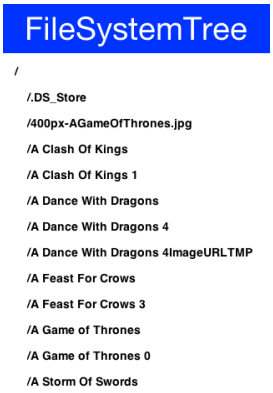


Figure 257. Simple sample of a tree for the FileSystemStorage API

42.3.3. Storage vs. file system

The question of storage vs. file system is often confusing for novice mobile developers. This embeds two separate questions:

- Why are there 2 APIs where one would have worked?
- Which one should you pick?

The main reasons for the 2 APIs are technical. Many OSes provide 2 ways of accessing data specific to the app and this is reflected within the API. For example: on Android the `FileSystemStorage` maps to APIs such as `java.io.FileInputStream` whereas the `Storage` maps to `Context.openFileInput()`.

The secondary reason for the two APIs is conceptual. `FileSystemStorage` is more powerful and in a sense provides more ways to fail, this is compounded by the complex on-device behavior of the API. `Storage` is designed to be friendlier to the uninitiated and more portable.

You should pick `Storage` unless you have a specific need that prevents it. Some APIs such as `Capture` expect a `FileSystemStorage` URI so in those cases this would also be a need.

Another case where `FileSystemStorage` is beneficial is the case of hierarchy or native API usage. If you need a directory structure or need to communicate with a native API the `FileSystemStorage` approach is easier.



In some OSes the `FileSystemStorage` API can find the content of the `Storage` API. As one is implemented on top of the other. This is undocumented behavior that can change at any moment!

To summarize the differences between the 3 file storage options:

Table 11. Compare Storage, FileSystem & Jar Resources

Option	Storage	File System	JAR Resource
Main Use Case	General application Data	low-level access	Ship data within the app
Initial State	Blank	Blank	As defined by developer

Option	Storage	File System	JAR Resource
Modifiable	Yes	Yes	No
Supports Hierarchies	No	Yes	No

42.4. SQL

Most new devices contain one version of sqlite or another; sqlite is a lightweight SQL database designed for embedding into devices. For portability recommend avoiding SQL altogether since it's both fragmented between devices (different sqlite versions) and isn't supported on all devices.

In general SQL seems overly complex for most embedded device programming tasks.

Portability Of SQLite

SQLite is supported on iOS, Android, JavaScript, and Desktop/Simulator builds. The JavaScript port relies on the browser's WebSQL implementation, so it may be unavailable in environments that have already removed that feature.

The biggest issue with SQLite portability is in iOS. The SQLite version for most platforms is threadsafe and as a result stable. But, the iOS version isn't!

This might not seem like a big deal, but if you forget to close a connection the GC might close it for you thus producing a crash. This is such a common occurrence that Codename One logs a warning when the GC collects a database resource on the simulator.

SQL is pretty powerful and well suited for common tabular data. The Codename One SQL API is similar in spirit to JDBC but considerably simpler since many of the abstractions of JDBC designed for pluggable database architecture make no sense for a local database.

The Database [https://www.codenameone.com/javadoc/com/codename1/db/Database.html] API is a high-level abstraction that allows you to open an arbitrary database file using syntax such as:

Some SQLite apps ship with a "ready made" database. You allow you to replace the DB file by using the code:

```
String path = Display.getInstance().getDatabasePath("databaseName");
```

You can then use the `FileSystemStorage` class to write the content of your DB file into the path. Notice that it must be a valid SQLite file!



`getDatabasePath()` isn't supported in the JavaScript port. It will always return null.

This is useful for applications that need to synchronize with a central server or applications that ship with a large database as part of their core product.

Working with a database is pretty trivial, the application logic below can send arbitrary queries to

the database and present the results in a `Table`. You can probably integrate this code into your app as a debugging tool:

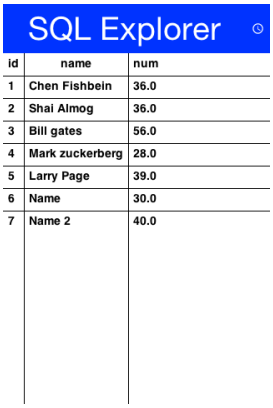


Figure 258. Querying the temp demo generated by the SQLDemo application

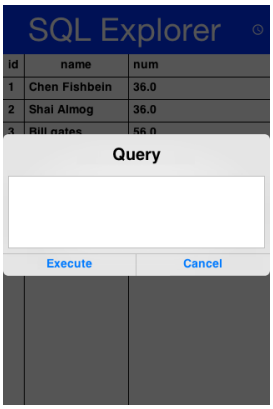


Figure 259. Issuing a query

42.5. Network manager & connection request

One of the more common problems in Network programming is spawning a new thread to handle the network operations. In Codename One this is done seamlessly and becomes unessential thanks to the NetworkManager [http://www.codenameone.com/javadoc/com/codename1/io/NetworkManager.html].

`NetworkManager` effectively alleviates the need for managing network threads by managing the complexity of network threading. The connection request class can be used to ease web service requests when coupled with the JSON/XML parsing capabilities.

To open a connection one needs to use a ConnectionRequest [http://www.codenameone.com/javadoc/com/codename1/io/ConnectionRequest.html] object, which has some similarities to the networking mechanism in JavaScript but is obviously somewhat more elaborate.

You can send a get request to a URL using something like:

Notice that you can also implement the same thing and much more by avoiding the response listener code and instead overriding the methods of the `ConnectionRequest` class which offers many points to override for example:



Notice that overriding `buildRequestBody(OutputStream)` will work for `POST` requests and will replace writing the arguments



You don't need to close the output/input streams passed to the request methods. They're implicitly cleaned up.

`NetworkManager` also supports synchronous requests which work in a similar way to `Dialog` through the `invokeAndBlock` call and thus don't block the EDT [1] illegally. For example: you can do something like this:

Notice that in this case the `addToQueueAndWait` method returned after the connection completed. Also notice that this was totally legal to do on the EDT!

42.5.1. Timeouts & retries

Each request can override the global timeout using `setTimeout(int)`, which expects a duration in milliseconds. This is useful when you're talking to endpoints that occasionally need a longer window than the default NetworkManager setting. You can also configure how many times Codename One should retry a failing call by using `setSilentRetryCount(int)`. When the silent retry limit is reached the standard error handling kicks in (for example: listeners fire, fail dialogs show, etc.).

42.5.2. Threading

By default the `NetworkManager` launches with a single network thread. This is enough for simple applications that don't do too much networking but if you need to fetch many images concurrently and perform web services in parallel this might be an issue.



Once you increase the thread count there is no guarantee of order for your requests. Requests might not execute in the order with which you added them to the queue!

To update the number of threads use:

```
NetworkManager.getInstance().updateThreadCount(4);
```

All the callbacks in the `ConnectionRequest` occur on the network thread and **not on the EDT**!

One exception to this rule exists: the `postResponse()` method, designed to update the UI after the networking code completes.



Never change the UI from a `ConnectionRequest` callback. You can either use a listener on the `ConnectionRequest`, use `postResponse()` (which is the exception to this rule) or wrap your UI code with `callSerially`.

42.5.3. VPN detection (best effort)

`NetworkManager` includes two APIs for querying VPN state:

```
NetworkManager nm = NetworkManager.getInstance();
```

```
if (nm.isVPNDetectionSupported()) {
    boolean vpnActive = nm.isVPNActive();
}
```



This is a heuristic signal, not a security control.

Both Android and iOS implementations rely on platform networking metadata and interface-name patterns (for example: `tun`, `ppp`, `utun`, `ipsec`). This means the result can be wrong in either direction:

- False negatives: Some VPNs may hide or avoid these patterns.
- False positives: Other virtual/tunnel interfaces may look like VPNs.

Use this API for UX or diagnostics (for example: warnings, telemetry, or logging), not for access control or fraud prevention by itself.

42.5.4. Arguments, headers & methods

HTTP/S is a complex protocol that expects complex encoded data for its requests. Codename One tries to simplify and abstract most of these complexities behind common sense APIs while still providing the full low-level access you would expect from such an API.

Arguments

HTTP supports many "request methods," most commonly `GET` & `POST` but also a few others such as `HEAD`, `PUT`, `DELETE` etc.

Arguments in HTTP are passed differently between `GET` and `POST` methods. That's what the `setPost` method in Codename One determines, whether arguments added to the request should be placed using the `GET` semantics or the `POST` semantics.

If you continue your example from above you can do something like this:

This will implicitly add a get argument with the content of `value`. Notice that you don't care what value is. It's implicitly HTTP encoded based on the get/post semantics. In this case it will use the get encoding since you passed `false` to the constructor.

A simpler implementation could do something like this:

This would be almost identical but doesn't provide the convenience for switching back and forth between `GET`/`POST` and it isn't as fluent.

You can skip the encoding in complex cases where server code expects illegal HTTP characters (this happens) using the `addArgumentNoEncoding` method. You can also add many arguments with the same key using `addArgumentArray`.

Methods

As you explained above, the `setPost()` method allows you to manipulate the get/post semantics of a request. This implicitly changes the `POST` or `GET` method submitted to the server.

For example, if you wish to have finer grained control over the submission process for example: for making a `HEAD` request you can do this with code like:

Headers

When communicating with HTTP servers you often pass data within headers for authentication/authorization but also to convey various properties.

Some headers are built-in as direct APIs for example: content type is directly exposed within the API since it's a pretty common use case. You can set the content kind of post request using:

You can also add any arbitrary header type you want, for example: a common use case is basic authorization where the authorization header includes the Base64 encoded user/password combination as such:

This can be tedious to do if you want all requests from your app to use this header. For this use case you can use:

Server headers

Server returned headers are a bit trickier to read. You need to subclass the connection request and override the `readHeaders` method for example:

Here you can extract the headers one by one to handle complex headers such as cookies, authentication etc.

Error handling

As you noticed above practically all the methods in the `ConectionRequest` throw `IOException`. This allows you to avoid the `try/catch` semantics and let the error propagate up the chain so it can be handled uniformly by the application.

There are two distinct places where you can handle a networking error:

- The `ConnectionRequest` - by overriding callback methods
- The `NetworkManager` error handler

Notice that the `NetworkManager` error handler takes precedence thus allowing you to define a global policy for network error handling by consuming errors.

For example: to block all network errors from showing anything to the user you could do something like this:

The error listener is invoked first with the NetworkEvent [https://www.codenameone.com/javadoc/com/codename1/io/NetworkEvent.html] matching the error. Consuming the event prevents it from propagating further down the chain into the `ConnectionRequest` callbacks.

You can also override the error callbacks of the various types in the request for example: for a server error code you can do:



The error callback is triggered in the network thread!

> As a result it can't access the UI to show a `Dialog` or anything like that.

Another approach is to use the `setFailSilently(true)` method on the `ConnectionRequest`. This will prevent the `ConnectionRequest` from displaying any errors to the user. It's a powerful strategy if you use the synchronous version of the APIs for example:



> This code will work with the synchronous "AndWait" version of the method since the response code will take a while to return for the non-wait version.

Error stream

When you get an error code that isn't 200/300 you ignore the result. This is problematic as the result might contain information you need. For example: many webservices provide further XML/JSON based details describing the reason for the error code.

Calling `setReadResponseForErrors(true)` will trigger a mode where even errors will receive the `readResponse` callback with the error stream. This also means that APIs like `getData` and the listener APIs will also work in case of error.

42.5.5. GZIP

Gzip is a common compression format based on the lz algorithm, it's used by web servers around the world to compress data.

Codename One supports GZipInputStream [https://www.codenameone.com/javadoc/com/codename1/io/gzip/GZIPInputStream.html] and GZipOutputStream [https://www.codenameone.com/javadoc/com/codename1/io/gzip/GZIPOutputStream.html], which allow you to compress data seamlessly into a stream and extract compressed data from a stream. This is useful and can be applied to every arbitrary stream.

Codename One also features a GZConnectionRequest [https://www.codenameone.com/javadoc/com/codename1/io/gzip/GZConnectionRequest.html], which will automatically unzip an HTTP response if it's indeed gzipped. Notice that some devices (iOS) always request gzip'ed data and always decompress it for you, but for iOS it doesn't remove the gziped header. The `GZConnectionRequest` is aware of such behaviors so it's better to use that when connecting to the network (if applicable).

By default `GZConnectionRequest` doesn't request gzipped data (unzips it when its received) but its pretty easy to do so add the HTTP header `Accept-Encoding: gzip` for example:

Do the rest as usual and you should have smaller responses from the servers.

42.5.6. File upload

MultipartRequest [https://www.codenameone.com/javadoc/com/codename1/io/MultipartRequest.html] tries to simplify the process of uploading a file from the local device to a remote server.

You can always submit data in the `buildRequestBody` but this is flaky and has some limitations in devices/size allowed. HTTP standardized file upload capabilities through the multipart request protocol, this is implemented by countless servers and is well documented. Codename One supports this out of the box:



`MultipartRequest` is a `ConnectionRequest` most stuff you expect from there should work. Even addArgument etc.

Since you assume most developers reading this will be familiar with Java here is the way to implement the multipart upload in the servlet API:

42.5.7. Parsing

Codename One has many built in parsers for JSON, XML, CSV & Properties formats. You can use those parsers to read data from the Internet or data that's shipping with your product. For example: use the CSV data to setup default values for your application.

All your parsers are designed with simplicity and small distribution size; they don't check and will fail in odd ways when faced with broken data. The main logic behind this is that validation takes up CPU time on the device where CPU is a precious resource.

Parsing CSV

CSV is probably the easiest to use, the "Comma Separated Values" format is a list of values separated by commas (or some other character) with new lines to show another row in the table. These map well to an Excel spreadsheet or database table and are supported by default in all spreadsheets.

To parse a CSV use the CSVParser [https://www.codenameone.com/javadoc/com/codename1/io/CSVParser.html] class as such:

CSV Parsing

Col 1	Col 2	Col 3	Col 4
1997	Ford	E350	Super, "luxurious" truck

Figure 260. CSV parsing results, notice the escaped parentheses and comma

The data contains a two-dimensional array of the CSV content. You can change the delimiter character by using the `CSVParser` constructor that accepts a character.



Notice that you used `CharArrayReader` from the `com.codename1.io` package for this sample. You would want to use `java.util.InputStreamReader` for real world data.

JSON

The JSON ("JavaScript Object Notation") format is popular on the web for passing values to/from webservices since it works so well with JavaScript. Parsing JSON is as easy but has two different variations. You can use the JSONParser [https://www.codenameone.com/javadoc/com/codename1/io/JSONParser.html] class to build a tree of the JSON data as such:

The response is a `Map` containing a nested hierarchy of `Collection` (`java.util.List`), Strings and numbers to represent the content of the submitted JSON. To extract the data from a specific path iterate the `Map` keys and recurs into it.

The sample below uses results from an API of ice and fire [https://anapioficeandfire.com/] that queries

structured data about the "Song Of Ice & Fire" book series. Here is a sample result returned from the API for the query http://www.anapioficeandfire.com/api/characters?page=5&pageSize=3 :


```
[
  {
    "url": "http://www.anapioficeandfire.com/api/characters/13",
    "name": "Chayle",
    "culture": "",
    "born": "",
    "died": "In 299 AC, at Winterfell",
    "titles": [
      "Septon"
    ],
    "aliases": [],
    "father": "",
    "mother": "",
    "spouse": "",
    "allegiances": [],
    "books": [
      "http://www.anapioficeandfire.com/api/books/1",
      "http://www.anapioficeandfire.com/api/books/2",
      "http://www.anapioficeandfire.com/api/books/3"
    ],
    "povBooks": [],
    "tvSeries": [],
    "playedBy": []
  },
  {
    "url": "http://www.anapioficeandfire.com/api/characters/14",
    "name": "Gillam",
    "culture": "",
    "born": "",
    "died": "",
    "titles": [
      "Brother"
    ],
    "aliases": [],
    "father": "",
    "mother": "",
    "spouse": "",
    "allegiances": [],
    "books": [
      "http://www.anapioficeandfire.com/api/books/5"
    ],
    "povBooks": [],
    "tvSeries": [],
    "playedBy": []
  },
  {
    "url": "http://www.anapioficeandfire.com/api/characters/15",
```

```
    "name": "High Septon",
    "culture": "",
    "born": "",
    "died": "",
    "titles": [
      "High Septon",
      "His High Holiness",
      "Father of the Faithful",
      "Voice of the Seven on Earth"
    ],
    "aliases": [
      "The High Sparrow"
    ],
    "father": "",
    "mother": "",
    "spouse": "",
    "allegiances": [],
    "books": [
      "http://www.anapioficeandfire.com/api/books/5",
      "http://www.anapioficeandfire.com/api/books/8"
    ],
    "povBooks": [],
    "tvSeries": [
      "Season 5"
    ],
    "playedBy": [
      "Jonathan Pryce"
    ]
  }
]
```

You will place that into a file named "anapioficeandfire.json" in the src directory to make the next sample simpler:

```
Form hi = new Form("JSON Parsing", new BoxLayout(BoxLayout.Y_AXIS));
JSONParser json = new JSONParser();
try(Reader r = new InputStreamReader(Display.getInstance().getResourceAsStream(getClass(), "/anapioficeandfire.json"),
"UTF-8");) {
    Map<String, Object> data = json.parseJSON(r);
    java.util.List<Map<String, Object>> content = (java.util.List<Map<String, Object>>)data.get("root"); ①
    for(Map<String, Object> obj : content) { ②
        String url = (String)obj.get("url");
        String name = (String)obj.get("name");
        java.util.List<String> titles =  (java.util.List<String>)obj.get("titles"); ③
        if(name == null || name.length() == 0) {
            java.util.List<String> aliases = (java.util.List<String>)obj.get("aliases");
            if(aliases != null && aliases.size() > 0) {
                name = aliases.get(0);
            }
        }
        MultiButton mb = new MultiButton(name);
        if(titles != null && titles.size() > 0) {
            mb.setTextLine2(titles.get(0));
        }
        mb.addActionListener((e) -> Display.getInstance().execute(url));
```

```
            hi.add(mb);
        }
    } catch(IOException err) {
        Log.e(err);
    }
    hi.show();
```

① The `JSONParser` returns a `Map` which is great if the root object is a `Map` but sometimes its a list of elements (as is the case above). In this case a special case "root" element is created to contain the actual list of elements.

② You rely that the entries are all maps, this might not be the case for every API type.

③ Notice that the "titles" and "aliases" entries are both lists of elements. You use `java.util.List` to avoid a clash with `com.codename1.ui.List`.



Figure 261. Parsed JSON result, clicking the elements opens the URL from the JSON



The structure of the returned map is sometimes unintuitive when looking at the raw JSON. The easiest thing to do is set a breakpoint on the method and use the inspect variables capability of your IDE to inspect the returned element hierarchy while writing the code to extract that data

An alternative approach is to use the static data parse() method of the `JSONParser` class and implement a callback parser for example:

```
JSONParser.parse(reader, callback);
```

Notice that a static version of the method is used! The callback object is an instance of the `JSONParseCallback` interface, which includes many methods. These methods are invoked by the parser to show internal parser states, this is like the way traditional XML SAX event parsers work.

Writing JSON — `JSONWriter`

For typed DTO serialization use the `@Mapped` annotation framework (see `JSON / XML Object Mapping`) plus `Mappers.toJson(…)`. For ad-hoc maps / lists where a build-time `Mapper` would be overkill, the `com.codename1.io.JSONWriter` class is the complement of `JSONParser`:

For tiny request bodies the fluent builders are usually faster to read than a `Map` literal:

Strings are double-quoted with the standard backslash escapes for `"`, `\`, `\n`, `\r`, `\t`, `\b`, `\f`, and control chars below `0x20` are emitted as `\u00xx`. The writer doesn't pretty-print; if you need indented output,

run the result through an external formatter at debug time.

XML parsing

The XMLParser [https://www.codenameone.com/javadoc/com/codename1/xml/XMLParser.html] started its life as an HTML parser built for displaying mobile HTML. That usage has since been deprecated but the parser can still parse many HTML pages and is "loose" in verification. This is both good and bad as the parser will work with invalid data without complaining.

The simplest usage of `XMLParser` looks a bit like this:

The Element [https://www.codenameone.com/javadoc/com/codename1/xml/Element.html] contains children and attributes. It represents a tag within the XML document and even the root document itself. You can iterate over the XML tree to extract the data from within the XML file.

You've had a great sample of working with `XMLParser` in the Tree Section [https://www.codenameone.com/manual/components.html#tree-section] of this guide.

`XMLParser` has the complimentary XMLWriter [https://www.codenameone.com/javadoc/com/codename1/xml/XMLWriter.html] class which can generate XML from the `Element` hierarchy. This allows developers to mutate (change) the elements and save them to a writer stream.

XPath processing

The Result [https://www.codenameone.com/javadoc/com/codename1/processing/Result.html] class provides a subset of XPath [https://www.w3schools.com/xml/xpath_intro.asp], but it isn't limited to XML documents, it can also work with JSON documents, and even with raw `Map` objects.

Start by demonstrating how to process a response from the Google Reverse Geocoder API [https://developers.google.com/maps/documentation/geocoding/]. Start with this XML snippet:

```
<?xml version="1.0" encoding="UTF-8"?>
<GeocodeResponse>
  <status>OK</status>
  <result> <!-- (irrelevant content removed) -->
      <address_component>
          <long_name>London</long_name>
          <short_name>London</short_name>
          <type>locality</type>
          <type>political</type>
      </address_component>
      <!-- (irrelevant content removed) -->
     <address_component>
          <long_name>Ontario</long_name>
          <short_name>ON</short_name>
          <type>administrative_area_level_1</type>
          <type>political</type>
      </address_component>
      <address_component>
         <long_name>Canada</long_name>
```

```
        <short_name>CA</short_name>
        <type>country</type>
        <type>political</type>
      </address_component>
  </result>
</GeocodeResponse>
```

You want to extract some data above into simpler string results. You can do this using:

If you're at all familiar with processing responses from webservices, you will notice that what would require many lines of code of selecting and testing nodes in regular java can now be done in a single line using the new path expressions.

In the code above, input can be any of:

- `InputStream` directly from `ConnectionRequest.readResponse(java.io.InputStream)`.
- XML or JSON document in the form of a {@code String}
- XML DOM Element [https://www.codenameone.com/javadoc/com/codename1/xml/Element.html] returned from XMLParser [https://www.codenameone.com/javadoc/com/codename1/xml/XMLParser.html]
- JSON DOM `Map` returned from JSONParser [https://www.codenameone.com/javadoc/com/codename1/io/JSONParser.html]

To use the expression processor when calling a webservice, you could use something like the following to parse JSON (notice this is interchangeable between JSON and XML):

```
Form hi = new Form("Location", new BoxLayout(BoxLayout.Y_AXIS));
hi.add("Pinpointing Location");
Display.getInstance().callSerially(() -> {
    Location l = Display.getInstance().getLocationManager().getCurrentLocationSync();
    ConnectionRequest request = new ConnectionRequest("http://maps.googleapis.com/maps/api/geocode/json", false) {
        private String country;
        private String region;
        private String city;
        private String json;

        @Override
        protected void readResponse(InputStream input) throws IOException {
                Result result = Result.fromContent(input, Result.JSON);
                country = result.getAsString("/results/address_components[types='country']/long_name");
                region = result.getAsString(
"/results/address_components[types='administrative_area_level_1']/long_name");
                city = result.getAsString("/results/address_components[types='locality']/long_name");
                json = result.toString();
        }

        @Override
        protected void postResponse() {
            hi.removeAll();
            hi.add(country);
            hi.add(region);
            hi.add(city);
            hi.add(new SpanLabel(json));
            hi.revalidate();
        }
    };
    request.setContentType("application/json");
```

```
    request.addRequestHeader("Accept", "application/json");
    request.addArgument("sensor", "true");
    request.addArgument("latlng", l.getLatitude() + "," + l.getLongitude());

    NetworkManager.getInstance().addToQueue(request);
});
hi.show();
/* omitted */
```

The returned JSON looks something like this (notice it's snipped because the data is too long):

```
{
  "status": "OK",
  "results": [
    {
      "place_id": "ChIJJ5T9-iFawokRTPGaOginEO4",
      "formatted_address": "280 Broadway, New York, NY 10007, USA",
      "address_components": [
        {
          "short_name": "280",
          "types": ["street_number"],
          "long_name": "280"
        },
        {
          "short_name": "Broadway",
          "types": ["route"],
          "long_name": "Broadway"
        },
        {
          "short_name": "Lower Manhattan",
          "types": [
            "neighborhood",
            "political"
          ],
          "long_name": "Lower Manhattan"
        },
        {
          "short_name": "Manhattan",
          "types": [
            "sublocality_level_1",
            "sublocality",
            "political"
          ],
          "long_name": "Manhattan"
        },
        {
          "short_name": "New York",
          "types": [
            "locality",
            "political"
          ],
          "long_name": "New York"
```

```
        },
        {
          "short_name": "New York County",
          "types": [
            "administrative_area_level_2",
            "political"
          ],
          "long_name": "New York County"
        },
        {
          "short_name": "NY",
          "types": [
            "administrative_area_level_1",
            "political"
          ],
          "long_name": "New York"
        },
        {
          "short_name": "US",
          "types": [
            "country",
            "political"
          ],
          "long_name": "United States"
        },
        {
          "short_name": "10007",
          "types": ["postal_code"],
          "long_name": "10007"
        },
        {
          "short_name": "1868",
          "types": ["postal_code_suffix"],
          "long_name": "1868"
        }
      ],
      "types": ["street_address"],
      "geometry": {
        "viewport": {
          "northeast": {
            "lng": -74.0044642197085,
            "lat": 40.7156470802915
          },
          "southwest": {
            "lng": -74.0071621802915,
            "lat": 40.7129491197085
          }
        },
        "location_type": "ROOFTOP",
        "location": {
          "lng": -74.00581319999999,
```

```
          "lat": 40.7142981
        }
      }
    }
    /* SNIPED the rest */
  ]
}
```



Figure 262. Running the geocode sample above in the simulator

The XML processor handles global selections by using a double slash anywhere within the expression, for example:



Notice that Google's JSON webservice uses plural form for each of the node names in that API (ie. results, address_components, and types) where they don't in the XML services (that's result, address_component etc.)

Second example

It's also possible to do some more complex expressions. You'll use the following XML fragment for the next batch of examples:

```
<rankings type="aus" gender="male" date="2011-12-31">
    <player id="1036" coretennisid="6752" rank="1"
       delta="0" singlespoints="485000" doublespoints="675"
       deductedpoints="0" totalpoints="485675">
        <firstname>Bernard</firstname>
        <lastname>Tomic</lastname>
        <town>SOUTHPORT</town>
        <state>QLD</state>
        <dob>1992-10-21</dob>
    </player>
    <player id="2585" coretennisid="1500" rank="2"
       delta="0" singlespoints="313500" doublespoints="12630"
       deductedpoints="0" totalpoints="326130">
        <firstname>Mathew</firstname>
        <lastname>Ebden</lastname>
        <town>CHURCHLANDS</town>
        <state>WA</state>
```

```
        <dob>1987-11-26</dob>
    </player>
    <player id="6457" coretennisid="287" rank="3"
        delta="0" singlespoints="132500" doublespoints="1500"
        deductedpoints="0" totalpoints="134000">
       <firstname>Lleyton</firstname>
       <lastname>Hewitt</lastname>
       <town>EXETER</town>
       <state>SA</state>
       <dob>1981-02-24</dob>
    </player>
    <!-- ... etc ... -->
</rankings>
```

Above, if you want to select the IDs of all players that are ranked in the top 2, you can use an expression like:



Notice above that the expression is using an attribute for selecting both rank and id. In JSON documents, if you try to select an attribute, it will look for a child node under the attribute name you ask for)

If a document is ordered, you might want to select nodes by their position, for example:

It's also possible to select parent nodes, by using the `..` expression. For example:

Above, you globally find a lastname element with a value of 'Hewitt', then grab the parent node of lastname which happens to be the player node, then grab the ID attribute from the player node. Or, you could get the same result from the following simpler statement:

It's also possible to nest expressions, for example:

In the above example, if the player node had an address object, you'd be selecting all players from Canada. This is a simple example of a nested expression, but they can get much more complex, which will be required as the documents themselves get more complex.
Moving on, to select a node based on the existence of an attribute:

Above, you selected the IDs of all ranked players. Conversely, you can select the non-ranked players like this:



Logical not (!) operators aren't implemented)

You can also select by the existence of a child node

Above, you selected all players that have a middle name.
 Keep in mind that the Codename One path expression language isn't a full implementation of XPath 1.0, but does already handle many of the most useful features of the specification.

Properties files

Properties [https://www.codenameone.com/javadoc/com/codename1/io/Properties.html] files are standard key/value pairs encoded into a text file. This file format is familiar to Java developers and the Codename One specific version tries to be as close as possible to the original Java implementation.

Notice that properties file both in Java proper and in Codename One don't support non-ascii characters. To encode Unicode values into the properties file format you should use the `native2ascii` tool that ships with the JDK.

One major difference between standard Java properties and the ones in Codename One is that Codename One sorts properties alphabetically when saving. Java uses random order based on the `Hashtable` natural ordering.

This was done to provide consistency for saved files.

42.6. Debugging network connections

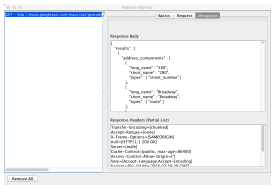


Figure 263. Debugging Network Connections

Codename One includes a Network Monitor tool which you can access through the simulator menu option. This tool reflects all the requests made through the connection requests and displays them in the left pane. This allows you to track issues in your code/web service and see everything the is "going through the wire."

This is a useful tool for optimizing and for figuring out what's happening with your server connection logic.

42.6.1. Simpler downloads

A common task is file download to storage or filesystem.

The Util [https://www.codenameone.com/javadoc/com/codename1/io/Util.html] class has simple utility methods:

`downloadUrlToFileSystemInBackground`, `downloadUrlToStorageInBackground`, `downloadUrlToFile` & `downloadUrlToStorage`.

These all delegate to a feature in ConnectionRequest [https://www.codenameone.com/javadoc/com/codename1/io/ConnectionRequest.html]:

`ConnectionRequest.setDestinationStorage(fileName)` & `ConnectionRequest.setDestinationFile(fileName);`

Both of which simplify the whole process of downloading a file.

Downloading images

Codename One has many ways to download an image and the general recommendation is the URLImage [https://www.codenameone.com/javadoc/com/codename1/ui/URLImage.html]. For example, the `URLImage` assumes that you know the size of the image in advance or that you're willing to resize it. In that regard it works great for some use cases but not so much for others.

The download methods mentioned above are great alternatives but they're a bit verbose when working with images and don't provide fine grained control over the `ConnectionRequest` for example: making a `POST` request to get an image.



Adding global headers is another use case but you can use addDefaultHeader [https://www.codenameone.com/javadoc/com/codename1/io/NetworkManager.html#addDefaultHeader-java.lang.String-java.lang.String-] to add those.

To make this process simpler there is a set of helper methods in ConnectionRequest that downloads images directly [https://www.codenameone.com/javadoc/com/codename1/io/ConnectionRequest.html#downloadImageToStorage-java.lang.String-com.codename1.util.SuccessCallback-].

These methods complement the `Util` methods but go a bit further and feature terse syntax for example: you can download a `ConnectionRequest` to `Storage` using code like this:

This effectively maps the `ConnectionRequest` directly to a SuccessCallback [https://www.codenameone.com/javadoc/com/codename1/util/SuccessCallback.html] for further processing.

URLImage caching

`URLImage` is great. It changed the way you do some things in Codename One.

For example, when you introduced it you didn't have support for the cache filesystem or for the JavaScript port. The cache filesystem is probably the best place for images of `URLImage` so supporting that as a target is an obvious choice, but JavaScript seems to work so why would it need a special case?

JavaScript already knows how to download and cache images from the web. `URLImage` is actually a step back from the things a good browser can do so why not use the native abilities of the browser when you're running there and fallback to using the cache filesystem if it's available and as a last resort go to storage...

That's what the new method of `URLImage` does:

A few important things you need to notice about this method:

- It returns **Image** and not **URLImage**. This is crucial. Down casting to `URLImage* will work on the simulator but might fail in some platforms (for example: JavaScript) so don't do that!
  Since this is implemented natively in JavaScript you need a different abstraction for that platform.
- It doesn't support image adapters and instead uses a simplified resize rule. Image adapters work on `URLImage` since you have a lot of control in that class. For example, in the browser your control is limited and so an adapter won't work.

If you do use this approach it would be far more efficient when running in the JavaScript port and will make better use of caching in most OSes.

42.7. Rest API

The `Rest` API makes it easy to invoke a RESTful webservice without many of the nuances of `ConnectionRequest`. You can use it to define the HTTP method and start building based on that. To get a parsed JSON result from a URL you could do:

For a lot of REST requests this will fail because you need to add an HTTP header indicating that you accept JSON results. You have a special case support for that:

POST requests use the same builder pattern:

Notice the usage of post and the body builder method. MANY methods exist in the builder class that cover pretty much everything you would expect and then some when it comes to the needs of rest services.

Some highlights that are easy to miss:

- `.priority(byte)` lets you change the underlying `ConnectionRequest` priority when you need certain calls to jump the queue.
- `.cookiesEnabled(boolean)` controls whether cookies are persisted for the request when you need stateless behavior.
- `.useBoolean(boolean)` and `.useLongs(boolean)` toggle how the JSON parser materializes number and boolean types inside the resulting `Map`, which is handy when your back end is strict about data types.
- `.cacheMode(…)` and `.postParameters(…)` expose the same knobs as `ConnectionRequest`, keeping you in the fluent API even for advanced tweaks.

42.7.1. Top-level JSON arrays — `fetchAsJsonList`

When the endpoint returns a top-level JSON array (for example `[{...}, {...}]`), use `fetchAsJsonList` rather than `fetchAsJsonMap`. The underlying `JSONParser` wraps top-level arrays under a synthetic `"root"` key; `fetchAsJsonList` unwraps that envelope for you so the callback receives the array directly:

There is no separate builder for the ambiguous "object or array" case: when the response shape isn't known up-front, use `fetchAsJsonMap` and branch on `data.get("root") instanceof List`.

42.7.2. Typed responses — `fetchAsMapped` and `fetchAsMappedList`

`Rest.fetchAsJsonMap` returns a generic `Map<String, Object>` that the caller casts and inspects key by key. Once your DTOs are `@Mapped`-annotated (see `JSON / XML Object Mapping`), `fetchAsMapped` returns the typed object directly:

For endpoints that return a list of DTOs, use `fetchAsMappedList`:

Both builders fall back to the untyped `fetchAsJsonMap` / `fetchAsJsonList` path internally and then route every map element through the build-time mapper that the `@Mapped` annotation processor generated. If a mapper isn't registered for the requested class (typical cause: the `process-annotations` Mojo didn't run on the class, or the class isn't `@Mapped`-annotated), the callback completes with `null` data — so always inspect `response.getResponseCode()` rather than `response.getResponseData()` to differentiate a server error response from the missing-mapper case.

The code in the kitchen sink webservice sample was updated to use this API. The result is shorter and more readable without sacrificing anything.

42.7.3. Rest in practice - twilio

The best way to explain the usage of this API is through a concrete "real world" example. Twilio provides many great telephony-oriented webservices to developers. One of those is an SMS sending webservice which can be useful for things such as "device activation."

To get started you would need to sign up to Twilio [http://twillo.com/] and have the following 3 variable values:

```
String accountSID = "----------------";
String authToken = "---------------";
String fromPhone = "your Twilio phone number here";
```



You can open a trial Twilio account and it tags all your SMS's. Notice you would need to use a You based number if you don't want to pay

You can now send hello world as an SMS to the end user. Once this is in place sending an SMS through REST is a matter of using the `Rest` API:

Notice that this is equivalent of this "curl" command:

```
curl 'https://api.twilio.com/2010-04-01/Accounts/[accountSID]/Messages.json' -X POST \
--data-urlencode 'To=[destinationPhone]' \
--data-urlencode 'From=[fromPhone]' \
--data-urlencode 'Body=Hello World' \
-u [accountSID]:[AuthToken]
```

That's pretty cool as the curl command maps almost directly to the `Rest` API call!

What you do here is actually pretty trivial, you open a connection the api messages URL. You add arguments to the body of the post request and define the basic authentication data.

The result is in JSON form you ignore it since it isn't that important but it might be useful for error handling. This is a sample response (redacted keys):

```
{
    "sid": "[sid value]",
```

```
    "date_created": "Sat, 09 Sep 2017 19:47:30 +0000",
    "date_updated": "Sat, 09 Sep 2017 19:47:30 +0000",
    "date_sent": null,
    "account_sid": "[sid value]",
    "to": "[to phone number]",
    "from": "[from phone number]",
    "messaging_service_sid": null,
    "body": "Sent from your Twilio trial account - Hello World",
    "status": "queued",
    "num_segments": "1",
    "num_media": "0",
    "direction": "outbound-api",
    "api_version": "2010-04-01",
    "price": null,
    "price_unit": "USD",
    "error_code": null,
    "error_message": null,
    "uri": "/2010-04-01/Accounts/[sid value]/Messages/SMe802d86b9f2246989c7c66e74b2d84ef.json",
    "subresource_uris": {
        "media": "/2010-04-01/Accounts/[sid value]/Messages/[message value]/Media.json"
    }
}
```

Notice the error message entry which is null meaning there was no error, if there was an error you'd have a message there or an error code that isn't in the 200-210 range.

This should display an error message to the user if there was a problem sending the SMS:

42.8. Webservice wizard

The Webservice Wizard can be invoked directly from the plugin. It generates stubs for the client side that allow performing simple method invocations on the server. It also generates a servlet that can be installed on any servlet container to intercept client side calls.

The types of values that can be passed through the webservice wizard protocol have limits, but it's highly efficient since it's a binary protocol and extensible through object externalization. All methods are provided both as asynchronous and synchronous calls for the convenience of the developer.

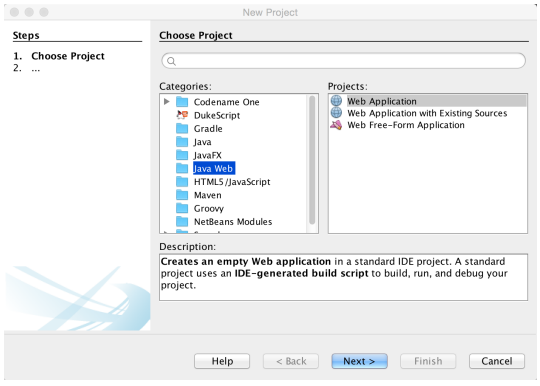


Figure 264. The first step in creating a client/server connection using the webservice wizard is to create a web application

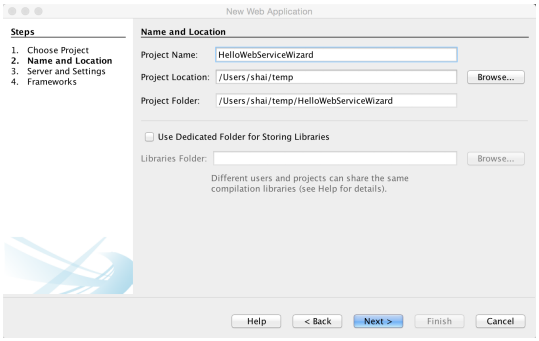


Figure 265. Any name will do

You should have a server setup locally. The example uses Tomcat since it's trivial and the requirements are modest, but many great Java webservers exist and this should work with all them!

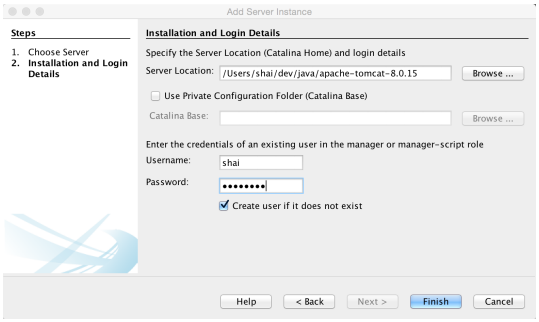


Figure 266. Setup your webserver in the IDE

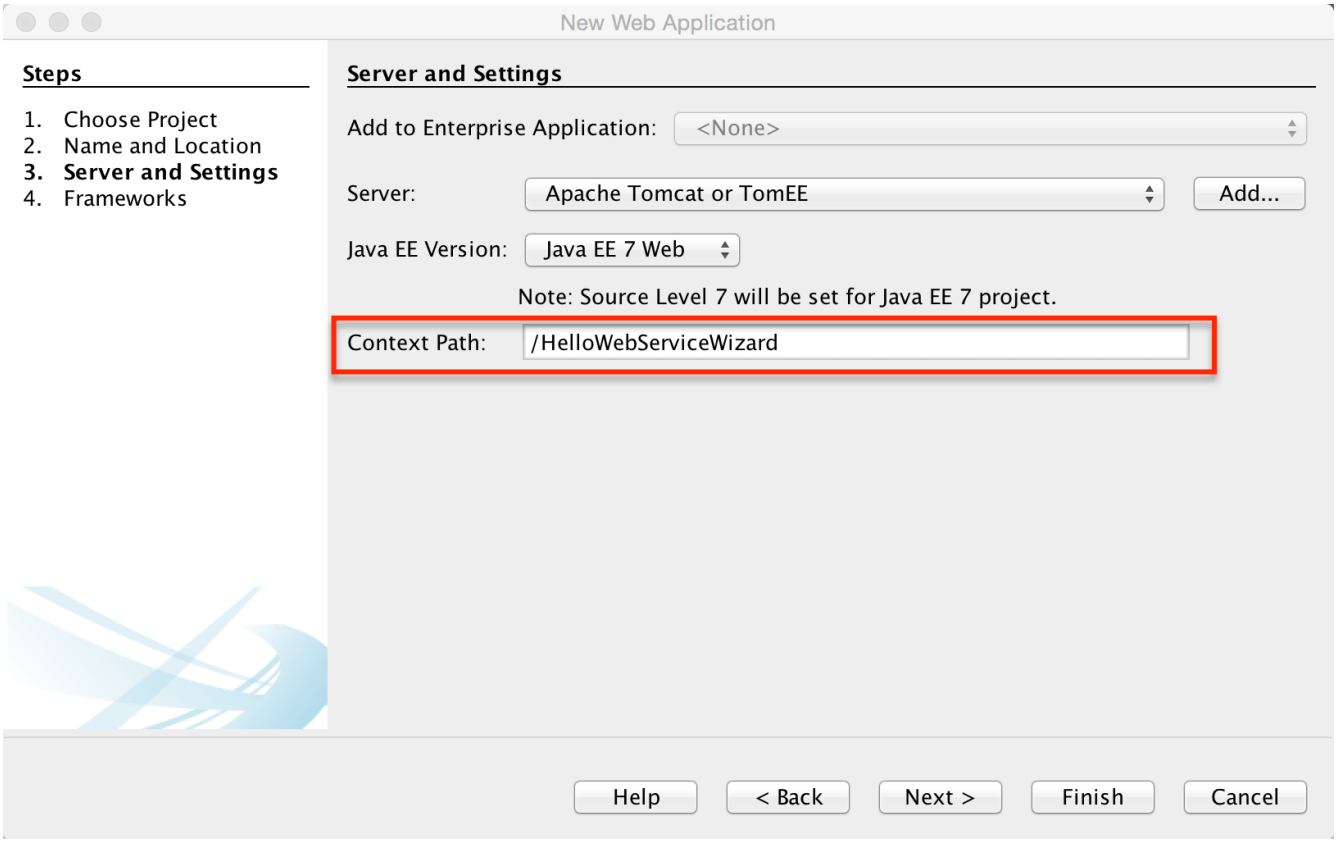


Figure 267. Configure the application server to the newly created app, notice the application context value which you will use later

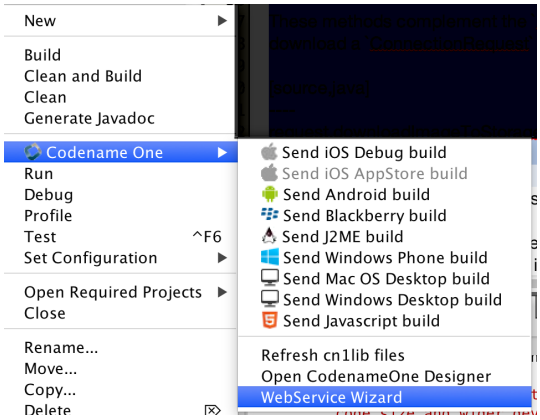


Figure 268. In the main Codename One project right-click and select the WebService Wizard option

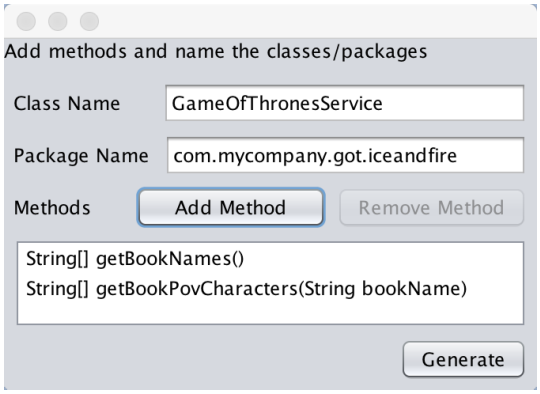


Figure 269. Set the package and class name for the webservice abstraction (notice this isn't your main class name) and then add the methods you want in the webservice



Figure 270. Add the methods and their arguments/return types. Once you finished adding all those press the "Generate" button



The types of arguments are pretty limited but you can pass an arbitrary `Externalizable` object which can be "anything"

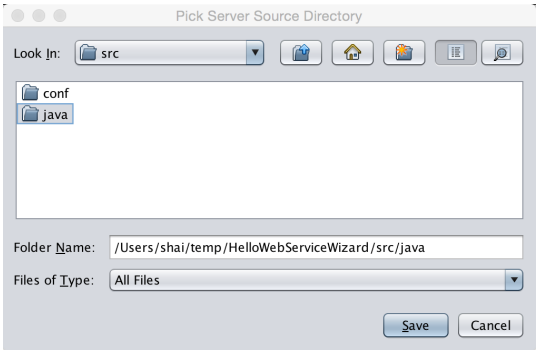


Figure 271. Pick the directory in the server project to which the source files will be generated by default this is the src/java directory under the project you created in the first step

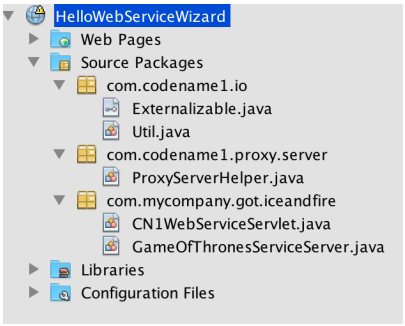


Figure 272. If you saved to the right location the server project directory should look like this

You can now open the `GameOfThronesServiceServer.java` file in the server and it looks like this:

```
public class GameOfThronesServiceServer {
    public static String[] getBookNames() {
        // your code goes herenull
        return null;
    }

    public static String[] getBookPovCharacters(String bookName) {
        // your code goes herenull
        return null;
    }
}
```

All you need to do is fill in the code, for this example you will implement the first method for simplicity:

```
public class GameOfThronesServiceServer {
    public static String[] getBookNames() {
        return new String[] {
            "A Game of Thrones", "A Clash Of Kings", "A Storm Of Swords", "A Feast For Crows",
            "A Dance With Dragons", "The Winds of Winter", "A Dream of Spring"
        };
    }

    public static String[] getBookPovCharacters(String bookName) {
        // your code goes herenull
        return null;
```

```
    }
}
```

Now lets open the client side code, in the `GameOfThronesService.java` file you see this

The destination URL needs to point at the actual server which you will recall from the new project creation should include `HelloWebServiceWizard`. You can fix the URL to:

```
private static final String DESTINATION_URL = "http://localhost:8080/HelloWebServiceWizard/cn1proxy";
```

You would need to update the host name of the server for running on a device otherwise the device would need to live within your internal network and point to your IP address.

It's now time to write the actual client code that calls this. Every method you defined above is now defined as a static method within the `GameOfThronesService` class with two permutations. One is a synchronous permutation that behaves as expected. It blocks the calling thread while calling the server and might throw an `IOException` if something failed.

This kind of method (synchronous method) is easy to work with since it's legal to call it from the event dispatch thread and it's easy to map it to application logic flow.

The second kind of method uses the async JavaScript style callbacks and accepts the callback interface. It returns and doesn't throw any exception. It will call onSuccess/onError based on the server result.

You can pick either one of these approaches based on your personal preferences. Here you show both uses with the server API:

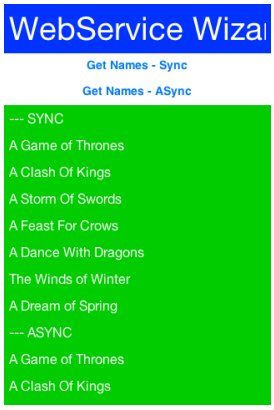


Figure 273. The final result of the WebService Wizard code

42.9. Connection request caching

Caching server data locally is a huge part of the advantage a native app has over a web app. This is non-trivial as it requires a delicate balance if you want to test the server resource for changes.

HTTP provides two ways to do that the ETag [https://en.wikipedia.org/wiki/HTTP_ETag] and Last-Modified [https://developer.mozilla.org/en-You/docs/Web/HTTP/Headers/Last-Modified]. While both are great they're non-trivial to use and by no definition seamless.

You added an experimental feature to connection request that allows you to set the caching mode to one of 4 states either globally or per connection request:

- **OFF** is the default meaning no caching.
- **SMART** means all get requests are cached intelligently and caching is "mostly" seamless
- **MANUAL** means that the developer handles the actual caching but the system won't do a request on a resource that's already "fresh"
- **OFFLINE** will fetch data from the cache and won't try to go to the server. It will generate a 404 error if data isn't available

You can toggle these in the specific request by using `setCacheMode(CachingMode)` and set the global default using `setDefaultCacheMode(CachingMode)`.



Caching applies to `GET` operations, it won't work for `POST` or other methods

There are many methods of interest to keep an eye for:

42.9.1. getCachedData()

This returns the cached data. This is invoked to implement `readResponse(InputStream)` when running offline or when you detect that the local cache isn't stale.

The smart mode implements this and will fetch the right data. For example, the manual mode doesn't store the data and relies on you to do so. In that case you need to return the data you stored at this point and must implement this method for manual mode.

42.9.2. Cacheunmodified()

This is a callback that's invoked to show a cache hit, meaning that you already have the data.

The default implementation still tries to call all the pieces for compatibility (for example: `readResponse`). For example, if this is unnecessary you can override that method with a custom implementation or even a blank implementation to block such a case.

42.9.3. Purgecache & purgeCacheDirectory

These methods are pretty self-explanatory. Notice one caveat though...

When you download a file or a storage element you don't cache them and rely on the file/storage element to be present and serve as "cache." When purging you won't delete a file or storage element you downloaded and thus these might remain.

For example, you do remove the `ETag` and `Last-Modified` data so the files might get refreshed the next time around.

42.10. Cached data service

The CachedDataService [https://www.codenameone.com/javadoc/com/codename1/io/services/

CachedDataService.html] allows caching data and updating it if the data changed on the server. The download APIs won't check for update if there is a local cache of the data (for example: `URLImage` always uses the local copy). This isn't a bad thing, it's pretty efficient.

For example, it might be important to update the image if it changed but you still want caching.

The `CachedDataService` will fetch data if it isn't cached locally and cache it. When you "refresh" it will send a special HTTP request that will send back the data if it has been updated since the last refresh:

42.11. Externalizable objects

Codename One provides the Externalizable [https://www.codenameone.com/javadoc/com/codename1/io/Externalizable.html] interface, which is like the Java SE `Externalizable` interface. This interface allows an object to declare itself as `Externalizable` for serialization (so an object can be stored in a file/storage or sent over the network). For example, due to the lack of reflection and use of obfuscation these objects must be registered with the Util [https://www.codenameone.com/javadoc/com/codename1/io/Util.html] class.

Codename One doesn't support the Java SE Serialization API due to the size issues and complexities related to obfuscation.

The major objects that are supported by default in the Codename One `Externalizable` are: `String`, `Collection`, `Map`, `ArrayList`, `HashMap`, `Vector`, `Hashtable`, `Integer`, `Double`, `Float`, `Byte`, `Short`, `Long`, `Character`, `Boolean`, `Object[]`, `byte[]`, `int[]`, `float[]`, `long[]`, `double[]`.

Externalizing an object such as h below should work fine:

```
Map<String, Object> h = new HashMap<>();
h.put("Hi","World");
h.put("data", new byte[] {(byte)1});
Storage.getInstance().writeObject("Test", h);
```

For example, notice that some things aren't polymorphic for example: if you will externalize a `String[]` you will get back an `Object[]` since `String` arrays aren't detected by the implementation.



The externalization process caches objects so the app will seem to work and fail on restart!

Implementing the `Externalizable` interface is important when you want to store a proprietary object. In this case you must register the object with the `com.codename1.io.Util` class so the externalization algorithm will be able to recognize it by name by invoking:



You should do this early on in the app for example: in the `init(Object)` but you shouldn't do it in a static initializer within the object as that might never be invoked!

An `Externalizable` object **must** have a **default public constructor** and must implement the

following 4 methods:

The `getVersion()` method returns the current version of the object allowing the stored data to change its structure in the future (the version is then passed when internalizing the object). The object ID is a `String` uniquely representing the object; it corresponds to the class name (in the example above the Unique Name should be `MyClass`).



It's a common mistake to use `getClass().getName()` to implement `getObjectId()` and it would **seem to work** in the simulator. This isn't the case though!
Since devices obfuscate the class names this becomes a problem as data is stored in a random name that changes with every release.

Developers need to write the data of the object in the externalize method using the methods in the data output stream and read the data of the object in the internalize method for example:

Since strings might be null sometimes you also included convenience methods to implement such externalization. This effectively writes a boolean before writing the UTF to show whether the string is null:

Assuming you added a new date field to the object you can do the following. Notice that a `Date` is a `long` value in Java that can be null. For completeness the full class is presented below:

Notice that you need to check for compatibility during the reading process as the writing process always writes the latest version of the data.

42.12. UI bindings & utilities

Codename One provides many tools to simplify the path between networking/IO & GUI. A common task of showing a wait dialog or progress sign while fetching network data can be simplified by using the InfiniteProgress [https://www.codenameone.com/javadoc/com/codename1/components/InfiniteProgress.html] class for example:

The process of showing a progress bar for a long IO operation such as downloading is automatically mapped to the IO stream in Codename One using the SliderBridge [https://www.codenameone.com/javadoc/com/codename1/components/SliderBridge.html] class.



You can simulate network delays and disconnected network in the Simulator menu

The `SliderBridge` class can bind a `ConnectionRequest` to a `Slider` and effectively show the progress of the download. For example:

```
Form hi = new Form("Download Progress", new BorderLayout());
Slider progress = new Slider();
Button download = new Button("Download");
download.addActionListener((e) -> {
    ConnectionRequest cr = new ConnectionRequest("https://www.codenameone.com/img/blog/new_icon.png", false);
    SliderBridge.bindProgress(cr, progress);
    NetworkManager.getInstance().addToQueueAndWait(cr);
    if(cr.getResponseCode() == 200) {
```

```
        hi.add(BorderLayout.CENTER, new ScaleImageLabel(EncodedImage.create(cr.getResponseData())));
        hi.revalidate();
    }
});
hi.add(BorderLayout.SOUTH, progress).add(BorderLayout.NORTH, download);
hi.show();
```





Figure 274. SliderBridge progress for downloading the image in the slow network mode

42.13. Logging & crash protection

Codename One includes a Log [https://www.codenameone.com/javadoc/com/codename1/io/Log.html] API that allows developers to invoke `Log.p(String)` or `Log.e(Throwable)` to log information to storage.

As part of the premium cloud features it's possible to invoke Log.sendLog() to email a log directly to the developer account. Codename One can do that seamlessly based on changes printed into the log or based on exceptions that are uncaught or logged for example:

This code will send a log every 2 minutes to your email if anything was changed. You can place it within the init(Object) method of your application.

For a production application you can use `Log.REPORTING_PRODUCTION` which will email the log on exception.

42.14. Sockets

At this moment Codename One supports TCP sockets. Server socket (listen/accept) is available on Android and the simulator but not on iOS.

You can check if Sockets are supported using the `Socket.isSupported()`. You can test for server socket support using `Socket.isServerSocketSupported()`.

To use sockets you can use the `Socket.connect(String host, int port, SocketConnection eventCallback)` method.

To listen on sockets you can use the `Socket.listen(int port, Class scClass)` method which will instantiate a `SocketConnection` instance (scClass is expected to be a subclass of `SocketConnection`) for every incoming connection.

This simple example allows you to create a server and a client assuming the device supports both:

42.15. Properties

In standard Java you have a POJO (Plain Old Java Object) which has getters/setters for example: you can have a simple `Meeting` class like this:

```
public class Meeting {
     private Date when;
     private String subject;
     private int attendance;

     public Date getWhen() {
        return when;
    }

     public String getSubject() {
        return subject;
     }

     public int getAttendance() {
        return attendance;
     }

     public void setWhen(Date when) {
        this.when = when;
    }

     public void setSubject(String subject) {
        this.subject  = subject;
     }

     public void setAttendance(int attendance) {
        this.attendance = attendance;
     }
}
```

That's a classic POJO and it's the force that underlies JavaBeans and a few tools in Java.

The properties are effectively the getters/setters for example: `subject`, `when` etc. But properties have many features that are crucial:

- They can be manipulated in runtime by a tool that had no knowledge of them during compile time
- They're observable - a tool can monitor changes to a value of a property
- They can have meta-data associated with them

These features are crucial since properties allow you all kinds of magic for example: hibernate/ORM uses properties to bind Java objects to a database representation, jaxb does it to parse XML directly into Java objects and GUI builders use them to customize UI's visually.

POJO's don't support most of that so pretty much all Java based tools use a lot of reflection & bytecode manipulation. This works but has a lot of downsides for example: say you want to map an object both to the Database and to XML/JSON.

Would the bytecode manipulation collide?

Would it result in duplicate efforts?

And how do you write custom generic code that uses such abilities? Do you need to manipulate the VM?

42.15.1. Properties in Java

These are all abstract ideas, lets look at how you think properties should look in Java and how you can gain from this moving forward.

This is the same class as the one above written with properties:

```
public class Meeting implements PropertyBusinessObject {
     public final Property<Date,Meeting> when = new Property<>("when");
     public final Property<String,Meeting> subject = new Property<>("subject");
     public final Property<Integer,Meeting>  attendance = new Property<>("attendance");
     private final PropertyIndex idx = new PropertyIndex(this, "Meeting", when, subject, attendance);

    @Override
    public PropertyIndex getPropertyIndex() {
        return idx;
    }
}
```

This looks a bit like a handful, so start with usage which might clarify a few things, then dig into the class itself.

When you used a POJO you did this:

With properties you do this:

Encapsulation

At first glance it looks like you created public fields (which you did) but if you will look at the declaration you will notice the `final` keyword:

This means that this code won't compile:

All setting/getting must happen through the set/get methods and they can be replaced. For example: this is valid syntax that prevents setting the property to null and defaults it to an empty string:



You will discuss the reason for returning the `Meeting` instance below

Introspection & observability

Since `Property` is a common class it's pretty easy for introspective code to manipulate properties. For example, it can't detect properties in an object without reflection.

That's why you have the index object and the `PropertyBusinessObject` interface (which defines `getPropertyIndex`).

The `PropertyIndex` class provides metadata for the surrounding class including the list of the properties within. It allows enumerating the properties and iterating over them making them accessible to all tools.

Furthermore all properties are observable with the property change listener. You can write this to instantly print out any change made to the property:

42.15.2. The cool stuff

That's the simple stuff that can be done with properties, but they can do **much** more!

For starters all the common methods of `Object` can be implemented with almost no code:

This is easy thanks to introspection…

You already have some simple code that can convert an object to/from JSON Maps for example: this can fill the property values from parsed JSON:

And vice versa:

You also have a simple ORM solution that maps values to table columns and can create tables. It's no hibernate but sqlite isn't big iron so it might be good enough.

Constructors

One of the problematic issues with constructors is that any change starts propagating everywhere. If you have fields in the constructor and you add a new field later you need to keep the old constructor for compatibility.

You added a new syntax:

That's why every property in the definition needed the `Meeting` generic and the set method returns the `Meeting` instance…

You're pretty conflicted on this feature and are thinking about removing it.

Without this feature the code would look like this:

Seamless serialization

Suppose you have an object called `Contacts` which includes contact information for example:

```
public class Contact implements PropertyBusinessObject {
    public final IntProperty<Contact> id  = new IntProperty<>("id");
```

```
    public final Property<String, Contact> name = new Property<>("name");
    public final Property<String, Contact> email = new Property<>("email");
    public final Property<String, Contact> phone = new Property<>("phone");
    public final Property<Date, Contact> dateOfBirth = new Property<>("dateOfBirth", Date.class);
    public final Property<String, Contact> gender  = new Property<>("gender");
    public final IntProperty<Contact> rank  = new IntProperty<>("rank");
    public final PropertyIndex idx = new PropertyIndex(this, "Contact", id, name, email, phone, dateOfBirth, gender,
rank);

    @Override
    public PropertyIndex getPropertyIndex() {
        return idx;
    }

    public Contact() {
        name.setLabel("Name");
        email.setLabel("E-Mail");
        phone.setLabel("Phone");
        dateOfBirth.setLabel("Date Of Birth");
        gender.setLabel("Gender");
        rank.setLabel("Rank");
    }
}
```

Standard Java Objects can be serialized in Codename One by implementing the Codename One `Externalizable` interface. You also need to register the `Externalizable` object so the implementation will be aware of it. Codename One business objects are seamlessly `Externalizable` and you need to register them.

For example: you can do something like this in your `init(Object)` method:

After you do that once you can write/read contacts from storage if you so want:

This will obviously also work for things like `List<Contact>` etc.

Seamless SQL storage

Writing SQL code can be tedious. Which is why `SQLMap` is such an important API for some of you. `SQLMap` allows CRUD (Create, Read, Update, Delete) operations on the built-in SQLite database using property objects.

If you continue the example from above to show persistence to the SQL database you can do something like this:

In the above code you do the following:



Notice that now altering a created table isn't possible so if you add a new property you might need to detect that and do an `alter` call manually

You can then add entries to the contact table using:

You can update an entry using:

And delete an entry using:

Listing the entries is more interesting:

The arguments for the `select` method are:

- The object type
- The attribute by which you want to sort the result (can be null)
- Whether sort is ascending
- Number of elements to fetch
- Page to start with - in this case if you have more than 1000 elements you can fetch the next page using `sm.select(c, c.name, true, 1000, 1)`

There are many more configurations where you can fine tune how a specific property maps to a column etc.

What's still missing

The `SQLMap` API is simplistic and doesn't try to be Hibernate/JPA for mobile. Basic things aren't available now and won't work. This isn't necessarily a problem as mobile databases don't need to be as powerful as server databases.

Relational Mappings/JOIN

Right now you can't map an object to another object in the database with the typical one-many, one-one etc. relationships that you could do with JPA. The `SQLMap` API is simplistic and isn't suited for that level of mapping now.

If there is demand for this it's something you might add moving forward but your goal isn't to re-invent hibernate.

Threading

SQLite is sensitive to threading issues on iOS. You ignored the issue of threading and issue all calls in process. This can be a problem for larger data sets as the calls would go on the EDT.

This is something you might want to fix for the generic SQLite API so low-level SQL queries will work with your mapping in a sensible way.

Alter

Right now you don't support table altering to support updated schemas. This is doable and shouldn't be too hard to implement so if there is demand for doing it you will probably add support for this.

Complex SQL/Transactions

You ignored functions, joins, transactions and a lot of other SQL capabilities.

You can use SQL directly to use all these capabilities for example: if you begin a transaction before inserting/updating or deleting this will work as advertised but if a rollback occurs your mapping will be unaware of that so you will need to re-fetch the data.

You will notice you mapped autoincrement so you will try to map things that make sense for

various use cases, if you've such a use case you'd appreciate pull requests and feedback on the implementation.

Caching/Collision

As mentioned above, you don't cache anything and there might be a collision if you select the same object twice you will get two separate instances that might collide if you update both (one will "win").

That means you need to pay attention to the way you cache objects to avoid a case of a modified version of an object kept with an older version.

Preferences binding

Some objects make sense as global objects, you can use the `Preferences` API to store that data directly but then you don't have the type safety that property objects bring to the table. That's where the binding of property objects to preferences makes sense. For example: say you have a global `Settings` property object you can bind it to preferences using:

If settings has a property called `companyName` it would bind into `Preferences` under the `Settings.companyName` entry.

You can do some more elaborate bindings such as:

This would customize all entry keys to start with `MySettings-` instead of `Settings.`. This would also set the company name entry to `company` so in this case instead of `Settings.companyName` you'd have `MySettings-company`.

UI binding

One of the bigger features of properties are their ability to bind UI to a property. For example: if you continue the sample above with the `Contact` class, say you have a text field on the form and you want the property (which you mapped to the database) to have the value of the text field. You could do something like this:

That would work but what if you changed the property value, that wouldn't be reflected in the text field?

Also that works for text field but what about other types for example: numbers, check boxes, pickers etc. This becomes a bit more tedious with those.

Binding makes this all seamless. For example: the code above can be written as:

The cool thing is that this works with many component types and property types almost magically. Binding works by using an adapter class to convert the data to/from the component. The adapter itself works with a generic converter for example: this code:

Seems like the one above but it takes a String that's returned by the text field and seamlessly converts it to the integer needed by rank. This also works in the other direction…

You can build a UI that would allow you to edit the `Contact` property in memory:



Figure 275. Properties form for the contact

Binding object & auto commit

You skipped a couple of fact about the `bind()` method. It has an additional version that accepts a `ComponentAdapter` which allows you to adapt the binding to any custom 3rd party component. That's a bit advanced for now but I might discuss this later.

For example, the big thing "skipped" earlier was the return value... `bind` returns a `UiBinding.Binding` object when performing the bind. This object allows you to manipulate aspects of the binding specifically unbind a component and also manipulate auto commit for a specific binding.

Auto commit determines if a property is changed instantly or on `commit`. This is useful for a case where you have an "OK" button and want the changes to the UI to update the properties when "OK" is pressed (this might not matter if you keep different instances of the object). When autocommit is on (the default which you can change through `setAutoCommit` in the `UiBinding`) changes reflect instantly, when it's off you need to explicitly call `commit()` or `rollback()` on the `Binding` class.

`commit()` applies the changes in the UI to the properties, `rollback()` restores the UI to the values from the properties object (useful for a "reset changes" button).

Binding also includes the ability to "unbind" this is important if you have a global object that's bound to a UI that's discarded. Binding might hold a hard reference to the UI and the property object might create a memory leak.

By using the `disconnect()` method in `Binding` you can separate the UI from the object and allow the GC to clean up.

UI generation

Up until now this was pretty cool but if you looked at the UI construction code above you would see that it's pretty full of boilerplate code. The thing about boilerplate is that it shows where automation can be applied, that's the exact idea behind the magical "InstantUI" class. This means that the UI above can be generated using this code:

This still carries most of the flexibilities of the regular binding for example: you can still get a binding object using:

You might not have noticed this but in the previous verbose code you had lines like:

You might be surprised to know that this will still work seamlessly without doing anything, as

would the picker component used to pick a date…

The picker component implicitly works for date type properties, numeric constraints and numbers are implicitly used for number properties and check boxes are used for booleans.

However, how do you know to use an email constraint for the email property?

You've some special case defaults for some common property names, so if your property is named email it will use an email constraint by default. If it's named url or password etc. It will do the "right thing" unless you explicitly state otherwise. You can customize the constraint for a specific property using something like:

This will override the defaults you've in place. The goal of this tool is to have sensible "magical" defaults that "work."

[1] Event Dispatch Thread

Chapter 43. Push notifications

Codename One push uses a typed message envelope and an explicit client binding. Your main application class doesn't implement a push interface. The same application code receives messages from Android, iOS, Huawei devices, and supported web browsers.

You can use the managed BuildCloud service or install an adapter for custom native push. A custom-provider adapter is a low-level escape hatch: it bypasses BuildCloud registration and delivery completely, so your server owns tokens, credentials, targeting, retries, and provider errors.

43.1. Create and configure an application

Open the Codename One Console and select **Push**. Create one Push application for each logical application and environment. For example, keep production and staging separate so credentials, audiences, and analytics remain isolated.

The **Overview** page displays the client binding key and one readiness card for every supported provider. Copy the key into `PushClient.builder()`. Select any readiness card, including a green **Ready** card, to open **Settings** on that provider's tab.

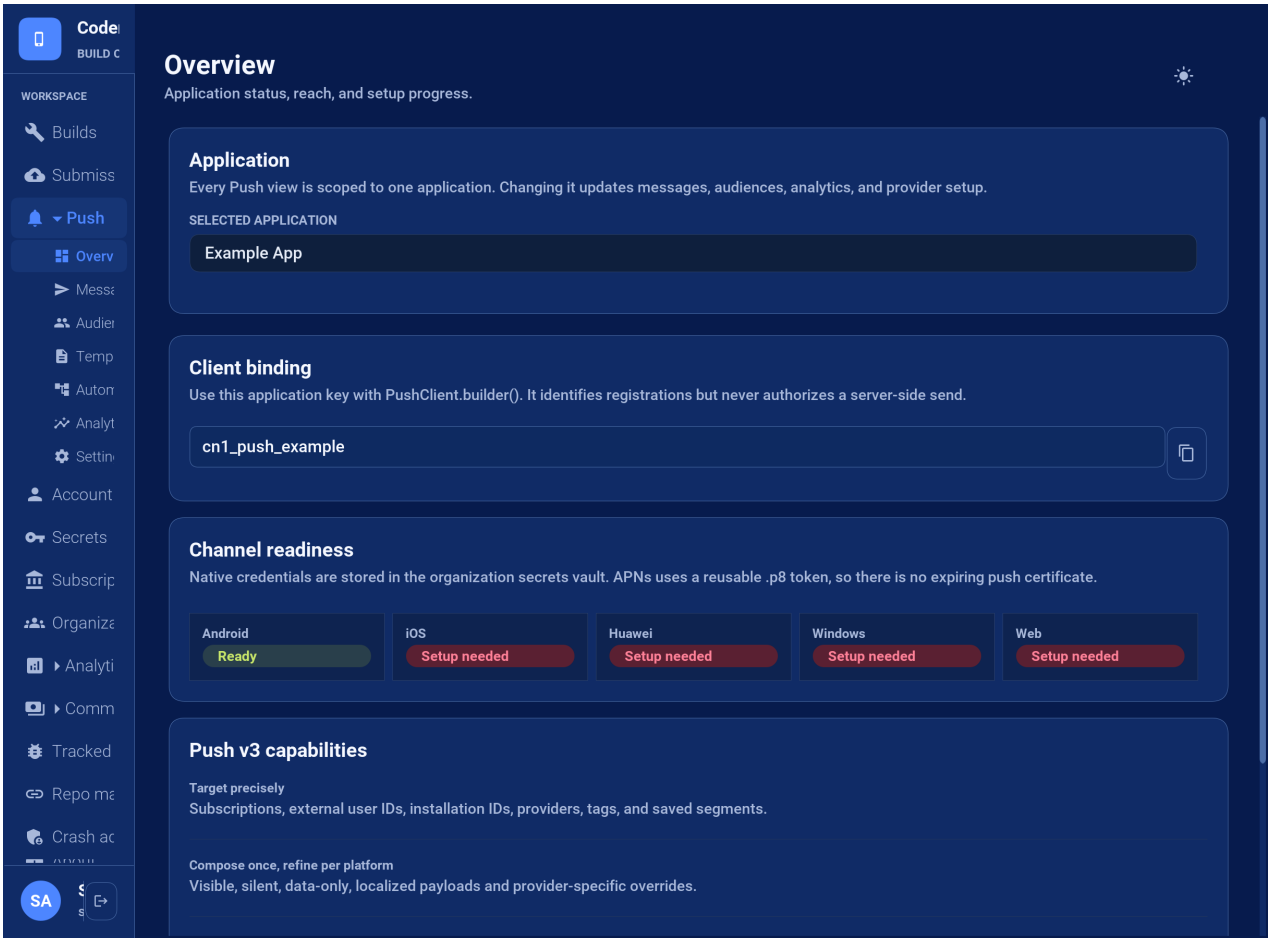


Figure 276. Push Overview shows the copyable application key and actionable provider readiness

Complete the provider setup tabs you need:

- **Android / FCM**: upload the Firebase service-account JSON file. The server uses the FCM HTTP v1 API.

- **iOS / APNs**: save the Apple team ID, key ID, bundle ID, and `.p8` signing key. Token authentication replaces expiring push certificates.
- **Huawei**: save the AppGallery Connect application ID and client secret. Add `agconnect-services.json` to the project.
- **Windows / WNS**: save the Package SID and client secret associated with the Store identity.
- **Web Push**: save the VAPID public and private keys and subject.

The managed backend includes WNS delivery for explicit Windows targets. It doesn't revive or generate bindings for the unsupported UWP port.

Credentials are encrypted in the BuildCloud secrets vault and are write-only. The setup UI shows whether each provider is configured but never reads a secret back. Rotate a credential by saving its replacement.

Cloud builds detect `PushClient` usage and generate the typed native bootstrap automatically. On Android, include `google-services.json`, `agconnect-services.json`, or both. A build with both configurations uses FCM when Google Play Services are available on the device and falls back to Huawei Push Kit otherwise. No provider-selection build hint is required. The `android.messagingService=fcm` and `android.messagingService=huawei` hints remain available only when an application intentionally needs to force one provider.

iOS enables push support automatically when `PushClient` is referenced; set `ios.includePush=false` only when an installed custom-provider adapter intentionally owns registration.

The Console displays an application key. This key identifies an application during device registration; it isn't a server API key and doesn't authorize sending messages. Keep send API keys on your server.

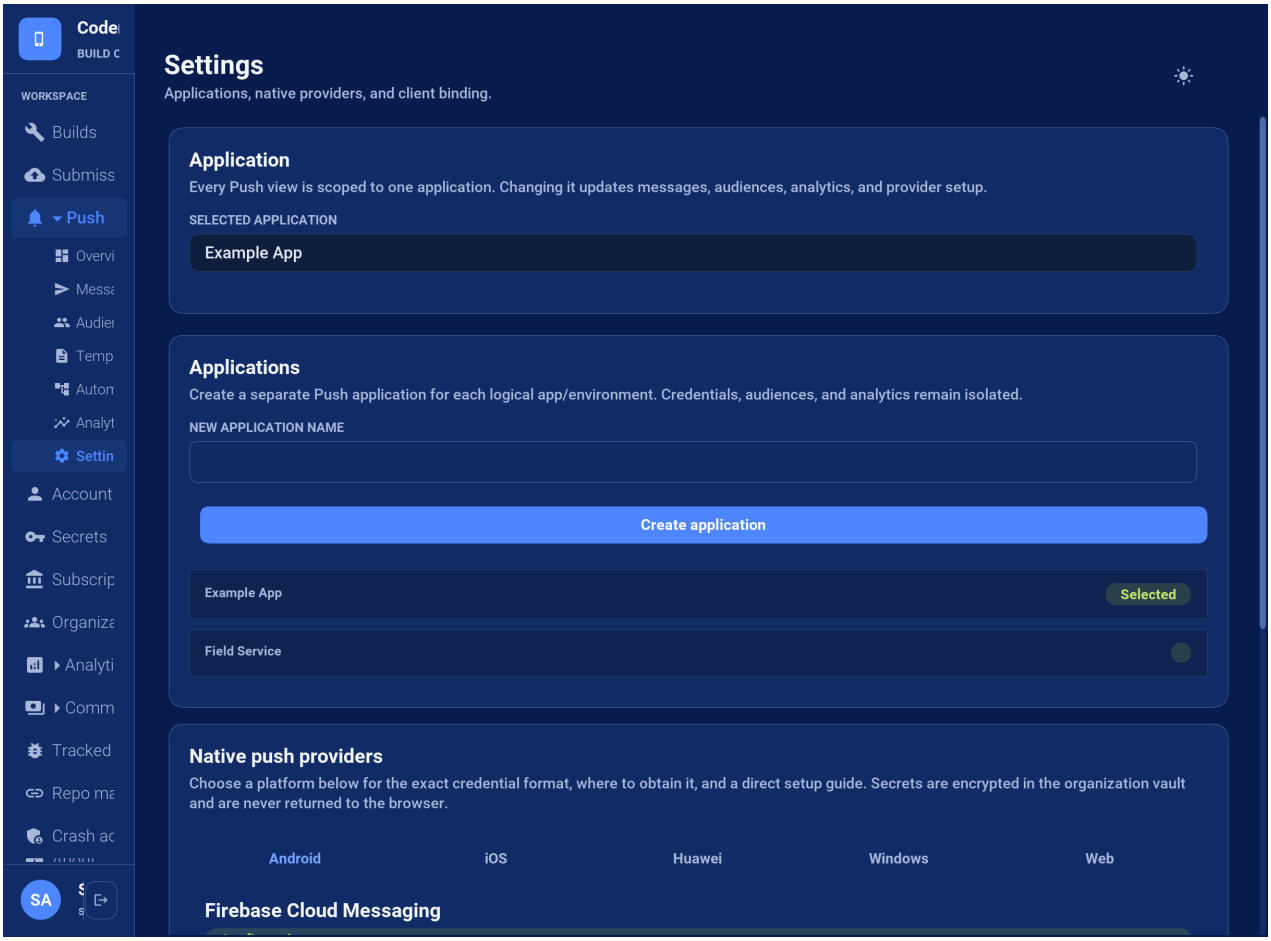


Figure 277. Provider Settings explains each credential and links to the provider console

43.2. Bind the client

Create one `PushClient` in `init()`, retain it in a field on the main application class, and call `register()` from `start()` after showing any permission rationale required by your UX:

```
private PushClient push;

public void init(Object context) {
    push = PushClient.builder("cn1_push_application_key")
            .listener(new PushListener() {
                @Override
                public void onRegistration(PushSubscription subscription) {
                    Log.p("Push ready on " + subscription.getTransportId());
                }

                @Override
                public void onMessage(PushMessage message) {
                    if (message.getDeepLink() != null) {
                        Log.p("Route to " + message.getDeepLink());
                    }
                }

                @Override
                public void onError(PushError error) {
```

```
                    Log.p(error.getCode() + ": " + error.getMessage());
                }
            })
            .build();
}

public void start() {
    push.register();
}

public void stop() {
    // Keep the subscription and listener active while the app is paused.
}

public void disableNotifications() {
    // Use only for an explicit user opt-out.
    push.unregister();
}
```

`start()` can run more than once during one process lifetime. `register()` is idempotent, so the example can call it on every start without creating duplicate native registrations. Don't call `unregister()` from `stop()`: it removes the device subscription. Call it only for an explicit user opt-out or account-removal workflow.

Registration is asynchronous. With managed delivery, the framework registers the native token with BuildCloud automatically. `PushSubscription` exposes the transport identifier, opaque native token, platform, installation ID, expiry, and capability list for diagnostics. Application code shouldn't parse the token or use it as a user identity.

On Android, the native bootstrap can obtain and persist an FCM or Huawei token before `start()` installs the `PushClient` callback. `register()` replays that persisted token after activating the listener and then asks the provider to refresh it, so the documented `init()`/`start()` lifecycle doesn't lose early registration.

43.2.1. Listener lifecycle and early messages

`PushClient.Builder.listener()` is mandatory, and `build()` fails immediately if it's missing. The client retains that listener until deregistration completes. Codename One doesn't locate listeners with reflection or `Class.forName()`: the application must build, retain, and register the client explicitly. Only one `PushClient` can be active in a process; a second client reports an `active_client` error instead of replacing the first listener.

Every `PushListener` and `PushRegistrationSink` callback runs on the Codename One EDT:

- `onRegistration()` runs after initial native registration and can run again when a provider rotates its token. Replace the previous server-side token for that installation.
- `onMessage()` receives one parsed schema-3 envelope. A foreground notification can arrive immediately. A visible background notification normally arrives after the user opens it.
- `onError()` reports registration and envelope errors. Use `PushError.getCode()` for decisions;

diagnostic text isn't a stable contract.

Native ports preserve a cold-start notification until the runtime can deliver it. If the runtime starts before the application calls `register()`, `PushClient` keeps a bounded process-local queue and replays it after activation. If the application never registers its client, it has provided no listener and can't expect application callbacks. Silent/background execution remains subject to each OS's power and scheduling rules, so refresh authoritative state whenever the application resumes.

43.3. Build a message

`PushMessage` is the canonical message model on both sides of the wire:

```
PushMessage message = PushMessage.builder()
        .title("Order shipped")
        .body("Order 4815 is on its way")
        .deepLink("myapp://orders/4815")
        .imageUrl("https://example.com/orders/4815.png")
        .collapseKey("order-4815")
        .ttlSeconds(3600)
        .data("orderId", "4815")
        .build();
```

Common fields have consistent meanings on every provider. Add provider-specific settings under the `platform` object only when the common model isn't sufficient. Unknown settings are ignored by unrelated providers.

```
{
  "schema": 3,
  "title": "Order shipped",
  "body": "Order 4815 is on its way",
  "deepLink": "myapp://orders/4815",
  "collapseKey": "order-4815",
  "ttl": 3600,
  "data": { "orderId": "4815" },
  "platform": {
    "apns": { "badge": 3, "sound": "default", "interruptionLevel": "time-sensitive" },
    "fcm": { "channelId": "orders", "priority": "high" },
    "huawei": { "channelId": "orders", "importance": "HIGH" },
    "wns": { "type": "toast" },
    "web": { "urgency": "high" }
  }
}
```

Don't place secrets or irreplaceable data in a notification. Providers, operating systems, and lock-screen UI may retain or expose payloads.

43.4. Send from a server

Create an API key with the `push` scope in the Console. Submit a typed message and one or more provider targets to `POST /api/v3/push/messages`:

```
curl -X POST https://cloud.codenameone.com/api/v3/push/messages \
  -H 'Authorization: Bearer YOUR_SERVER_API_KEY' \
  -H 'Content-Type: application/json' \
  -d '{
    "appId": "APPLICATION_ID",
    "targets": [
      {"provider":"fcm", "token":"OPAQUE_DEVICE_TOKEN"},
      {"provider":"apns", "token":"OPAQUE_DEVICE_TOKEN"}
    ],
    "message": {
      "schema":3,
      "title":"Order shipped",
      "body":"Order 4815 is on its way",
      "data":{"orderId":"4815"}
    }
  }'
```

A successful admission returns HTTP `202` with durable delivery IDs. Invalid targets are deactivated when the provider identifies them as permanent failures.

The complete request, response, authentication, error, target, and message schemas are in the Push sending OpenAPI 3.0 specification [https://cloud.codenameone.com/openapi/push-v3.yaml]. Import that file into Swagger UI, an OpenAPI code generator, or an API client instead of reproducing DTOs from this chapter.

Use the Console for operator-driven messages, stored audiences, campaigns, analytics, and automations.

43.5. Build audiences in the Console

Managed audiences are application-scoped. They don't contain copied device tokens. A saved segment stores a dynamic filter, and BuildCloud resolves that filter against active subscriptions when a campaign launches.

Your authenticated server assigns an external user ID and tags to an installation with `PUT /api/v3/push/apps/{appId}/installations/{installationId}`. Tags should be stable application facts such as `subscription=active`, `region=emea`, `role=technician`, or `testUser=true`. Don't let an untrusted client grant itself a privileged tag.

```
{
  "externalUserId": "customer-4815",
  "tags": {
    "subscription": "active",
```

```
    "region": "emea",
    "testUser": "false"
  }
}
```

Open **Push** > **Audience**. The tag and value pickers contain values discovered from current active subscriptions for the selected application. Choose an optional platform, choose one tag rule, enter a segment name, and select **Save segment**. The new segment appears immediately under **Saved segments** with its current recipient estimate. It also becomes available in the Messages recipient picker.

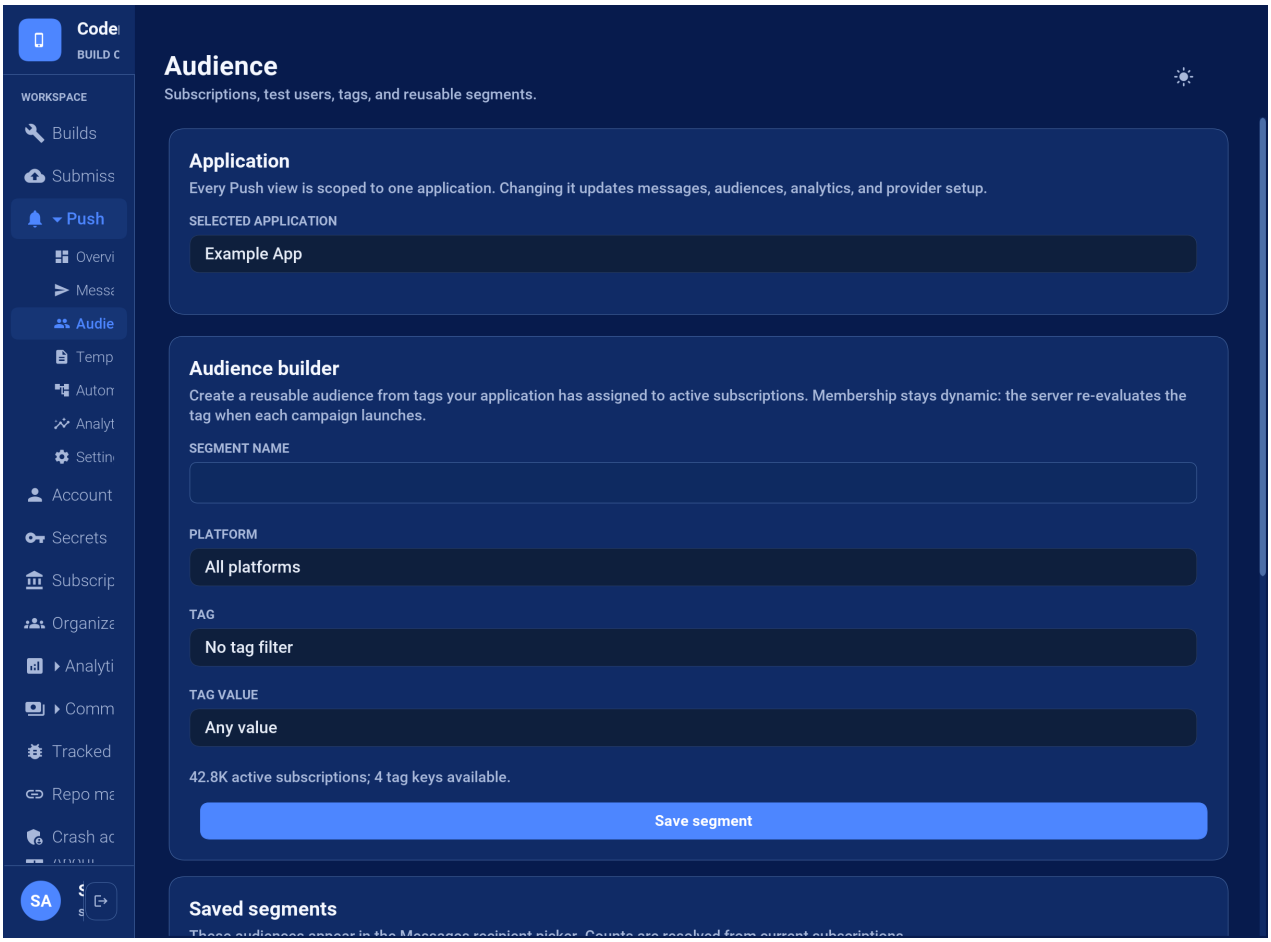


Figure 278. Audience uses discovered tag values and shows every saved segment

Segments are evaluated at send time. Changing a subscription's tags changes future membership without editing the segment. A count is an estimate of the current state; registrations, subscription removals, and tag updates can change it before admission.

For a safe rollout, tag internal devices with `testUser=true`, create a test-user segment, and send the message there first. Create the production segment only from server-controlled tags.

43.6. Send a message from the Console

Open **Push** > **Messages** and complete the three numbered sections:

1. **Audience**: choose **All subscribed devices** or a saved segment. The Console displays the current recipient estimate. Individual provider tokens aren't entered in this workflow.

2. **Message**: enter a notification title and message. A visible notification requires message text. For a silent notification, enable **Silent / data-only notification** and provide an additional-data key and value. The Console builds valid JSON; operators don't paste JSON into the primary composer.
3. **Delivery**: choose how long providers may retain an undelivered message. Console messages use safe cross-platform defaults and enter the durable queue immediately.

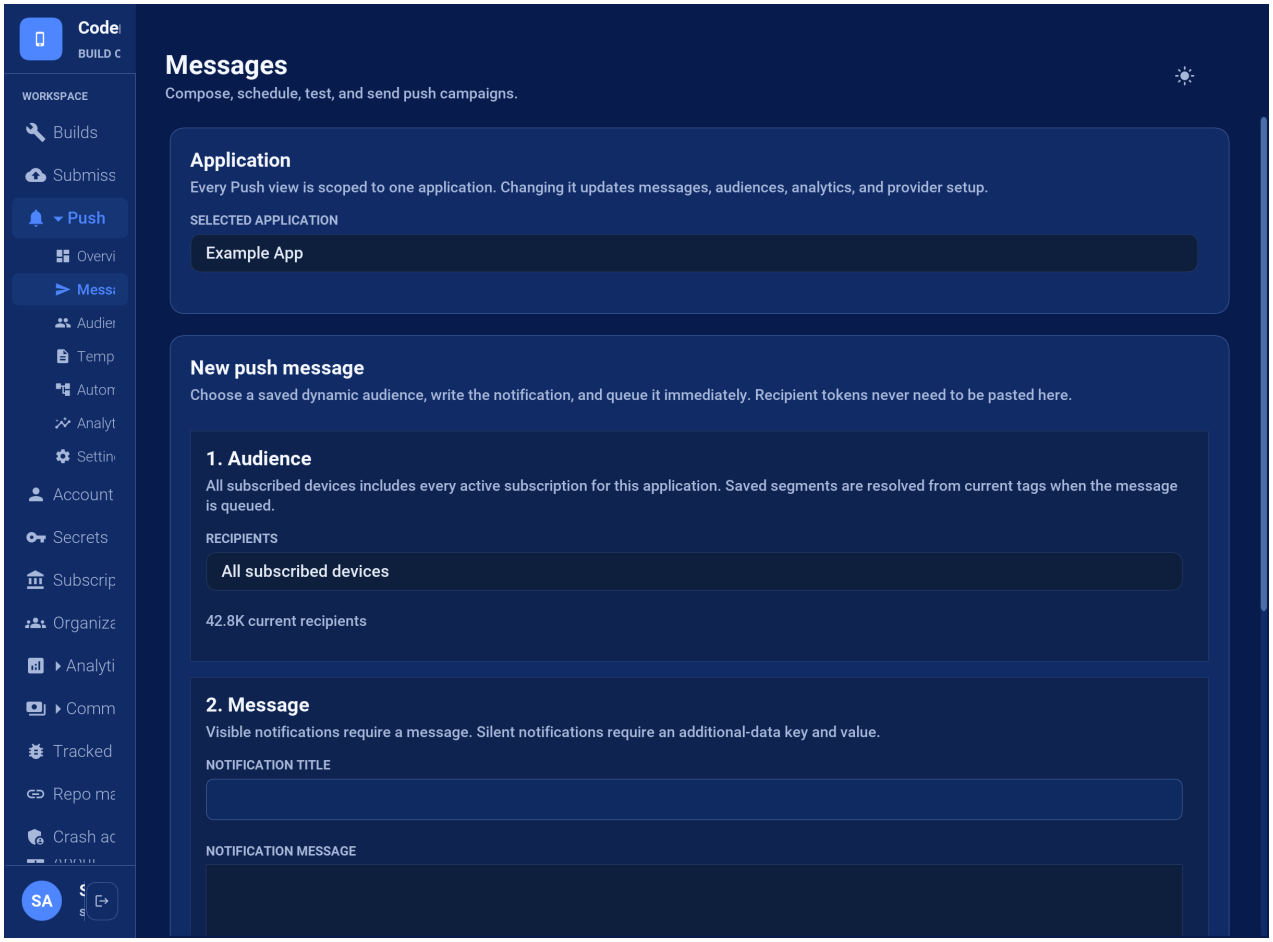


Figure 279. The restrictive message composer accepts a saved audience and typed fields

Select **Queue message** only after checking the audience name and estimate. BuildCloud creates the campaign, resolves the segment, and admits the recipients to the durable delivery queue. The provider may accept a notification that the device never displays, so inspect **Push** > **Analytics** for provider acceptance and failures without treating acceptance as device delivery.

43.7. Event automations

An automation is an enabled rule containing `trigger.event`, a schema-3 `message`, an optional `audience`, and an optional `delaySeconds`. If no audience is stored in the rule, the event must identify an `externalUserId` or `installationId`. Text fields in the message can insert event properties with `${event.propertyName}`.

```
{
  "enabled": true,
  "trigger": {"event": "order.ready"},
  "delaySeconds": 0,
```

```
  "message": {
    "schema": 3,
    "title": "Order update",
    "body": "Order ${event.orderId} is ready"
  }
}
```

Your server triggers enabled rules with `POST /api/v3/push/apps/{appId}/events` using a `push`-scoped API key:

```
{
  "name": "order.ready",
  "externalUserId": "customer-4815",
  "properties": {"orderId": "4815"}
}
```

Immediate matches enter the durable delivery queue. Delayed matches become scheduled campaigns, so restarts don't lose them. Automations are intentionally push rules, not a general-purpose customer journey or analytics product.

43.8. Widgets and live surfaces

A push can publish widget content or update a live activity before the regular listener receives the message. The `surface` object uses one of these operations:

```
{
  "schema": 3,
  "silent": true,
  "surface": {
    "operation": "widget",
    "kind": "order-status",
    "timeline": "{...serialized widget timeline...}"
  }
}
```

For a live activity use `live-update` or `live-end` with `id`, serialized `state`, and optional `dismissImmediately`. Unsupported surfaces are ignored. A surface command runs when the Codename One runtime receives the envelope. On platforms that don't start application code for a background push, the native queue preserves the command and applies it on the next start or resume. Applications must therefore refresh stale surface state during foreground activation too.

43.9. Use an application-owned push server

A custom server is a low-level alternative to managed push; it doesn't reuse the BuildCloud protocol at a different endpoint. A CN1Lib implements `PushTransport` around the private provider's native SDK. It obtains native registration material, reports token rotation, and forwards incoming data as

the canonical schema-3 JSON envelope. A matching `PushRegistrationSink` stores and removes the subscription on your server:

```
// companyTransport is supplied by the native CN1Lib.
// listener is the application's normal, non-null PushListener.
PushClient client = PushClient.builder("private-app-id")
        .transport(companyTransport)
        .registrationSink(new PushRegistrationSink() {
            @Override
            public void registered(PushSubscription value) {
                Log.p("Send the subscription to the company server");
            }

            @Override
            public void unregistered(PushSubscription value) {
                Log.p("Remove the subscription from the company server");
            }
        })
        .listener(listener)
        .build();
```

When a `PushTransport` implementation is present, `PushClient` never contacts BuildCloud. Your implementation owns these responsibilities:

- `getId()` returns a stable provider ID that your server understands.
- `isSupported()` checks whether the SDK can operate on this device.
- `register()` invokes `Callback.registered()` with a `PushSubscription`, then reports later token rotations through the same callback.
- `unregister()` removes the provider subscription and invokes `Callback.unregistered()` when complete.
- Incoming native messages call `Callback.message()` with one complete schema-3 envelope. Don't call the application listener directly and don't invent a second payload format.
- Registration or transport failures call `Callback.failed()`. The callback is thread-safe; `PushClient` moves application-facing callbacks to the EDT.

The registration sink should asynchronously insert or update the subscription by `installationId` on your server and replace its token when it rotates. The server then sends through your provider's native API. It must preserve the schema-3 envelope so the same `PushListener`, deep-link handling, and surface commands work in managed and custom-server modes.

The application still follows the normal lifecycle: retain one client, call `register()` from `start()`, and reserve `unregister()` for an actual opt-out. This custom integration doesn't introduce reflection or automatic listener discovery.

43.10. Testing delivery behavior

Hosted CI has no installed application, OS push daemon, permission state, or physical device, so it can't prove provider-to-device delivery. Codename One tests at the lowest controllable boundary instead. `PushClientTransportTest` injects a fake native transport and covers:

- foreground, background, force-stopped, and cold-start delivery;
- notification tap, action, dismissal, and deep-link routing;
- permission granted, denied, and later changed in system settings;
- token rotation, reinstall, expiry, logout, and multiple installations per user;
- silent delivery and simulated constrained-state replay;
- collapse keys, TTL expiry, Unicode, maximum payloads, images, and provider errors;
- queued widget and live-activity updates replayed at the native/runtime seam;
- retry idempotency and dead-letter behavior.

The JavaScript contract executes the real service-worker source with mocked browser clients and notification APIs. BuildCloud has a separate local suite using mocked provider HTTP responses for compatibility translation, credentials, durable queue behavior, retries, and automations. Physical-device sends are optional release diagnostics rather than a CI or nightly requirement. Provider acceptance isn't proof of device delivery.

Chapter 44. Notifications and Background Execution

Codename One exposes a cross-platform API for rich local notifications, runtime notification permission, notification channels, constraint-aware background work, foreground services, deferrable background processing, receiving content shared from other apps, and push topic subscriptions. Each capability degrades on platforms that can't support it — calls become documented no-ops rather than errors — and the JavaSE simulator implements them so you can exercise the flows without a device.

Because these features build on platform-specific behavior, every section below lists the support per platform. In the matrices: **Yes** = fully supported, **Partial** = supported with documented limitations, **No-op** = the call is accepted but does nothing, **Sim** = exercised in the simulator.

44.1. Notification permission

On iOS, and on Android 13+ (`POST_NOTIFICATIONS`), an app must obtain permission before its notifications are shown. Request it through `Display`:

```
NotificationPermissionRequest req = new NotificationPermissionRequest()
        .provisional(true)      // iOS: deliver quietly without an explicit prompt
        .timeSensitive(true);   // iOS: allow the time-sensitive interruption level

Display.getInstance().requestNotificationPermission(req, result -> {
    if (result.isGranted()) {
        // schedule notifications
    }
    AuthorizationLevel level = result.getAuthorizationLevel();
});
```

`NotificationPermissionResult.getAuthorizationLevel()` returns an `AuthorizationLevel` enum (`NOT_DETERMINED`, `DENIED`, `AUTHORIZED`, `PROVISIONAL`, `EPHEMERAL`). The values are ordered from least to most permissive, so `isGranted()` is "more permissive than `DENIED`." The callback is delivered on the EDT.

Request option	iOS	Android	Simulator
alert / sound / badge	Yes	Implied by POST_NOTIFICATIONS	Granted
provisional	Yes (iOS 12+)	No-op	Granted
timeSensitive	Yes (iOS 15+, needs entitlement)	No-op	Granted
critical	Yes (needs Apple entitlement)	No-op	Granted

Request option	iOS	Android	Simulator
carPlay / announcement	Yes	No-op	Granted

The `timeSensitive` and `critical` options require an Apple capability on the App ID. Enable injection of the matching entitlement with the `ios.timeSensitiveNotifications=true` and `ios.criticalAlerts=true` build hints (they're opt-in because an entitlement the provisioning profile doesn't carry breaks code signing).

44.2. Local notifications

`LocalNotification` is scheduled through `Display.scheduleLocalNotification(…)` as before; the bean has been extended with fluent, backward-compatible setters:

When the user taps an action, the selected action id (and any quick-reply text) are delivered through `com.codename1.push.PushContent` — `getActionId()`, `getActionTitle()` and `getTextResponse()` — before your `LocalNotificationCallback.localNotificationReceived(id)` runs, so push and local notification actions are handled identically.

Capability	iOS	Android	Simulator
title / body / badge / sound	Yes	Yes	Sim (panel)
image attachment	Yes (UNNotificationAttachment)	Yes (BigPicture)	Sim (thumbnail)
action buttons	Yes (UNNotificationAction)	Yes	Sim (buttons)
quick-reply input action	Yes (UNTextInputNotificationAction)	Yes (RemoteInput)	Sim (text field)
grouping	Yes (thread identifier)	Yes (group + summary)	Partial
progress (ongoing)	No-op	Yes	Sim (progress bar)
full-screen intent	No-op	Yes (needs permission, see hints)	No-op
time-sensitive	Yes (iOS 15+)	Approximated (category + priority)	Sim
messaging / conversation style	Partial	Yes (MessagingStyle)	Sim
custom view	Notification content extension	RemoteViews	Sim (render)

44.2.1. Resource names for sound, image and action icons

These string parameters are platform **resource names**, not numeric ids (Codename One has no `R.drawable` constants). They're resolved at runtime per platform:

Parameter	What to provide	Android resolution	iOS resolution
`alertSound` / channel sound	a file whose name starts with `notification_sound`, bundled in the app	`res/raw/<name>` via `android.resource://` URI	`UNNotificationSound` from the app bundle
`alertImage`	image path placed in the app root	loaded from `assets/`	`UNNotificationAttachment` from the bundle
`Action` icon	a drawable name bundled in `native/android`	`getResources().getIdentifier(name, "drawable", pkg)`	not displayed

The file extension is optional for the action icon and is ignored. iOS notification action buttons don't render icons, so the icon value is ignored there.

44.3. Notification channels (Android O+)

Channels let the user control importance, sound, vibration, lights, lock screen visibility and badge display for a group of notifications. Build and register one at startup, then reference it from a notification with `setChannelId(…)`:

Channels are an Android concept. On iOS and the simulator `register()` is a no-op (the simulator stores the channel so its name can be shown in the notification panel); the channel id you assign is still carried so the notification behaves consistently.

44.4. Constraint-aware background work

`BackgroundWork` schedules deferrable work that the OS runs when its constraints are met. Implement `BackgroundWorker` in a class with a public no-argument constructor (it's reconstructed after a cold launch — pass state through the input data, not fields):

	iOS	Android	Simulator
backend	BGTaskScheduler (iOS 13+)	JobScheduler	Timer
requiresNetwork / unmetered	Yes	Yes	Toggle in Simulate menu
requiresCharging	Yes (requiresExternalPower)	Yes	Toggle
requiresIdle / batteryNotLow	No-op	Yes	Toggle

	iOS	Android	Simulator
periodic	Approximated (resubmit)	Yes (>= 15 min)	Yes

`BackgroundTask.scheduleProcessing(id, earliestBeginDate, requiresNetwork, requiresPower, runnable)` is a one-shot variant for occasional maintenance work. On iOS the identifiers must be declared via the `ios.backgroundProcessingIds` build hint (comma separated, default `<packageName>.processing`); the build also adds the `processing` UIBackgroundMode and links `BackgroundTasks.framework` automatically.

44.5. Foreground service (Android)

`ForegroundService` runs a long task while a persistent notification is shown:

	Android	iOS	Simulator
persistent-notification service	Yes	No (best-effort background window + notification)	Sim (status indicator)

When the app references `ForegroundService` the build registers the service and injects `FOREGROUND_SERVICE` plus the per-type `FOREGROUND_SERVICE_<TYPE>` permission (Android 14). Override the type with the `android.foregroundServiceType` build hint (default `dataSync`).

44.6. Receiving shared content

Override `onReceivedSharedContent(SharedContent)` on your `Lifecycle` subclass to receive text, URLs, files or images shared into your app from other apps:

	Android	iOS	Simulator
receive shared content	Yes (SEND / SEND_MULTIPLE)	Yes (share extension + App Group)	Sim ("Send shared content")

On Android, declare the accepted MIME types with the `android.shareFilter` build hint (comma separated, for example `text/plain,image/*`); the build registers the share-receiver activity. On iOS this reuses the share-extension authoring pipeline; set the `ios.shareAppGroup` build hint to the App Group the extension writes to. File and image items are copied into Codename One `FileSystemStorage` so your code is platform neutral.

44.7. Push topics

`Push.subscribeToTopic("news")` / `unsubscribeFromTopic("news")` subscribe the device to a fan-out topic.

	Android	iOS
topic subscribe / unsubscribe	Yes (Firebase Cloud Messaging)	No-op (raw APNs has no topics; fan out server-side)

Server-side topic delivery is handled by the Codename One push server.

44.8. Simulator

The JavaSE simulator implements all the above so you can develop without a device. Use the **Simulate → Notifications and Background** menu to toggle the background constraints (network / charging / idle / battery), run scheduled work immediately, inspect registered channels, and inject shared content. Posted notifications appear in a panel at the top right; tapping an action (or submitting a quick-reply) routes back through your callback exactly as on device.

Chapter 45. Calendar integration

The `com.codename1.calendar` package provides scheduling rather than just an "add event" shortcut. It represents calendars, events, task/reminders, recurrence, attendees, alarms, free/busy intervals, attachments, conference links, paging, incremental synchronization, and explicit edit conflicts.

The API is capability based. Don't branch on the platform name. Ask the source what it can do:

```
LocalCalendarSource local = LocalCalendarSource.getInstance();
CalendarCapabilities capabilities = local.getCapabilities();

if (capabilities.supports(CalendarCapability.READ_EVENTS)) {
    local.requestAuthorization(CalendarAccess.EVENTS_READ_ONLY).ready(status -> {
        if (status == CalendarAuthorizationStatus.FULL) {
            local.queryEvents(new CalendarQuery()
                    .setCalendarId("primary")
                    .setStartTime(Instant.now()))
                 .ready(page -> page.getItems().forEach(System.out::println));
        }
    });
}
```

`LocalCalendarSource` is implemented with Android's Calendar Provider and Apple EventKit on iOS and Mac Catalyst. The simulator supplies an isolated, in-memory source so tests never modify the developer's real calendar. A port that has no local calendar service returns an empty capability set. Online sources and `.ics` import/export remain available on every port, including JavaScript and Linux.

Target	Built-in local source	Portable alternatives
Android	Calendar Provider events	Google, Microsoft, CalDAV, and `.ics`
iOS and Mac Catalyst	EventKit events and reminders	Google, Microsoft, CalDAV, and `.ics`
Codename One simulator	Isolated in-memory source	Google, Microsoft, CalDAV, and `.ics`
Win32, native macOS desktop, Linux, and JavaScript	No built-in local source; the capability set is empty	Google, Microsoft, CalDAV, and `.ics`

This API is a clean replacement for the old `CN1Calendar` integration. New code shouldn't mix objects or identifiers from the two APIs.

45.1. Creating an event

All-day values use `CalendarDateTime.allDay(LocalDate)`. Timed values use `ZonedDateTime` (or the

`Instant` and `ZoneId` convenience factory). Alarms use `Duration`, and query ranges and absolute timestamps use `Instant`. The API uses `java.time` instead of defining parallel date, time-zone, or duration types. Keeping all-day and timed values separate avoids shifting an all-day event when it crosses a time-zone boundary.

```
CalendarEvent event = new CalendarEvent()
        .setCalendarId(calendarId)
        .setTitle("Architecture review")
        .setStart(CalendarDateTime.instant(start, ZoneId.of("Europe/Paris")))
        .setEnd(CalendarDateTime.instant(end, ZoneId.of("Europe/Paris")))
        .setRecurrence(new CalendarRecurrenceRule()
                .setFrequency(CalendarRecurrenceRule.Frequency.WEEKLY)
                .addDayOfWeek(2))
        .addAttendee(new CalendarAttendee()
                .setName("Ari")
                .setEmail("ari@example.com"))
        .addAlarm(new CalendarAlarm().setTimeBefore(Duration.ofMinutes(15)));

local.saveEvent(event, CalendarMutationScope.ALL).ready(saved -> {
    Log.p("Created " + saved.getId());
});
```

The `version` returned by a source is an optimistic-concurrency token. Pass the returned object to `saveEvent()` or its version to a delete method. A stale token produces `CalendarError.CONFLICT` instead of overwriting a newer server edit.

45.2. Online sources

Online sources work on all ports and use the application's OAuth client. They never embed a client secret and never persist credentials implicitly.

```
OidcClient.discover("https://accounts.google.com").ready(oidc -> {
    oidc.setClientId(clientId)
        .setRedirectUri(redirectUri)
        .setScopes(GoogleCalendarSource.SCOPE_CALENDAR,
                   GoogleCalendarSource.SCOPE_TASKS)
        .authorize().ready(tokens -> {
            CalendarSource google = new GoogleCalendarSource(
                    new OidcCalendarTokenProvider(oidc, tokens));
            CalendarManager.getInstance().registerSource(google);
        });
});
```

`GoogleCalendarSource` covers Google Calendar and Google Tasks. `MicrosoftCalendarSource` covers Microsoft Graph calendars and Microsoft To Do. Both expose provider paging and delta tokens through `CalendarPage` and retry once with a refreshed token after HTTP 401.

For CalDAV, pass the account's calendar-home URL and an authentication strategy:

```
CalendarSource caldav = new CalDavCalendarSource(
        "work", "Work CalDAV", calendarHomeUrl,
        CalDavAuthentication.digest(username, password));
```

`CalDavAuthentication.basic()`, `bearer()`, and `digest()` are supported. Prefer Bearer or Digest over Basic unless Basic is protected by HTTPS. Calendar discovery filters collections by their advertised `VEVENT` and `VTODO` component support. Check the result of each asynchronous operation as servers can restrict individual collections further.

45.3. Import and export

`ICalendarCodec` reads and writes RFC 5545 `VEVENT`, `VTODO`, `VALARM`, attendees, recurrence rules, URI attachments, conference URIs, all-day values, and time zones. Unknown `X-` properties are retained in provider data.

```
String ics = ICalendarCodec.writeEvent(event);
CalendarEvent imported = ICalendarCodec.readEvent(ics);
```

45.4. Change notifications and offline synchronization

The cross-platform guarantee is intentionally pull based:

- Local sources can emit `CalendarChange` through `addChangeListener()` when the operating system reports a local store change.
- Online providers return a sync/delta token. Store it and pass it in the next `CalendarQuery`. The API doesn't register provider webhooks.

Listeners run on the Codename One EDT. Treat a `RESET` change as an instruction to query again; it doesn't promise one callback per underlying OS edit.

Offline mutation storage is opt-in. Supply a `CalendarCache` to `CalendarSyncEngine`, queue edits, and call `sync()` yourself or from the platform's background-work callback. `StorageCalendarCache` stores calendar data and pending mutations, never credentials. Conflicts remain paused until the application chooses `KEEP_LOCAL`, `KEEP_REMOTE`, or `MERGED`.

```
CalendarSyncEngine sync = new CalendarSyncEngine(
        google, new StorageCalendarCache("google-account-1"));
sync.queueEventSave(event, CalendarMutationScope.ALL);
sync.sync().ready(result -> {
    for (CalendarConflict conflict : result.getConflicts()) {
        // Present both versions, then call resolveConflict(...).
    }
});
```

45.5. Permissions and build hints

Calling `LocalCalendarSource.getInstance()`, `CalendarManager.getLocalSource()`, `CalendarManager.getSources()`, or `Display.getLocalCalendarSource()` is the builder signal for local calendar integration. Android builds add `READ_CALENDAR` and `WRITE_CALENDAR`; iOS and Mac Catalyst builds add EventKit and calendar/reminder privacy strings. You can replace the default user-facing strings with:

```
ios.NSCalendarsFullAccessUsageDescription=Explain why events are needed
ios.NSCalendarsWriteOnlyAccessUsageDescription=Explain why creating events is needed
ios.NSRemindersFullAccessUsageDescription=Explain why reminders are needed
```

The built-in Win32 source currently reports unavailable. An application-supplied native integration can explicitly opt into the standard MSIX `uap:appointments` capability with:

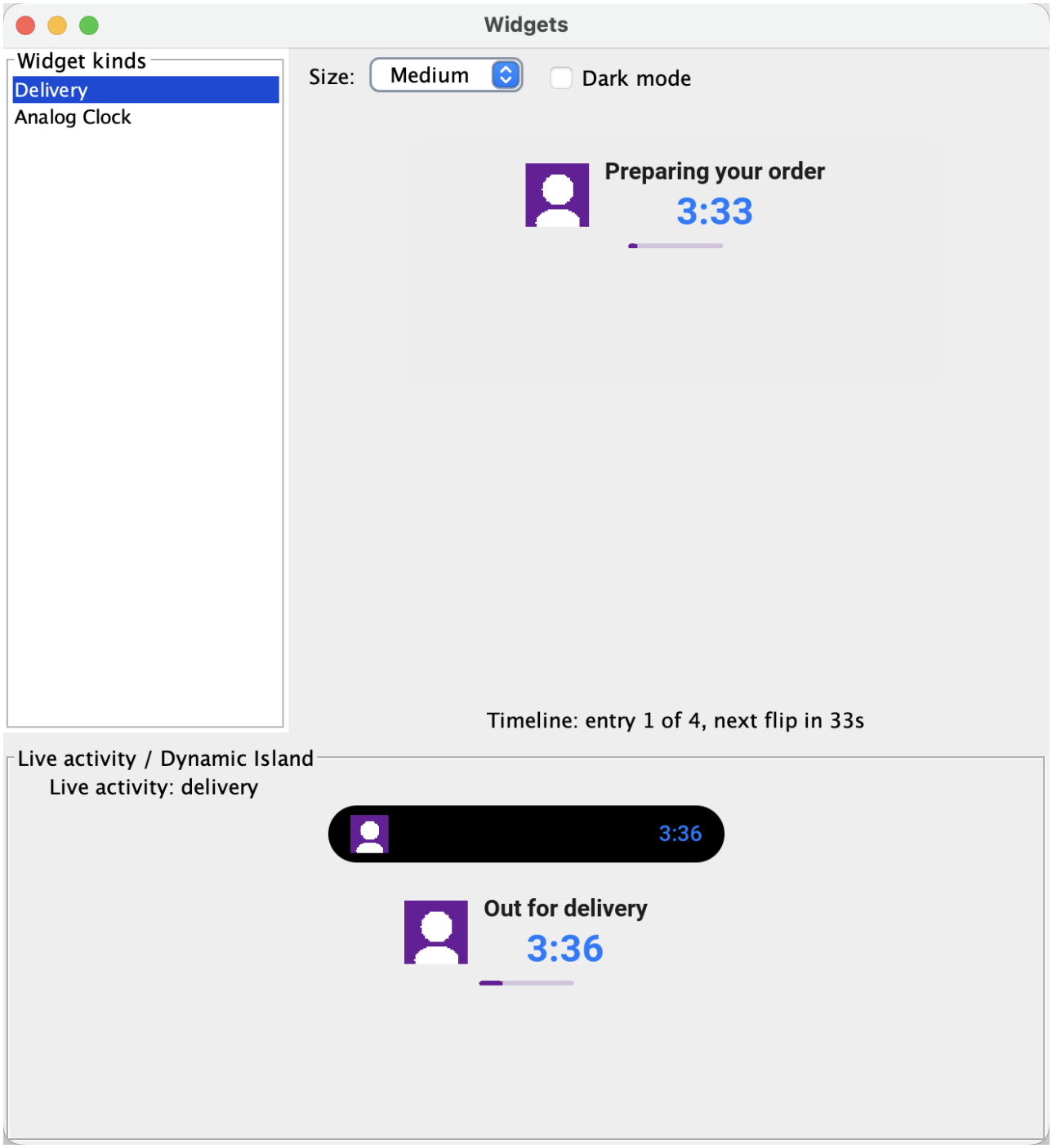
```
windows.msix=true
windows.calendar.restrictedCapability=true
```

Always check `getCapabilities()` and `getAuthorizationStatus()` at runtime. Manifest declarations make a request possible; they don't mean the user or provider granted it.

Chapter 46. External Surfaces: Widgets and Live Activities

Home-screen widgets and live activities answer the same developer question: how to keep a live piece of information outside the app. Codename One models both as one concept — an external surface — under the `com.codename1.surfaces` package. A **widget** is persistent, user-placed and content-driven: the weather, the next meeting, the delivery status on the home screen. A **live activity** is transient, app-started and progress-driven: the delivery that's out right now, a running timer, a live score on the iOS lock screen and Dynamic Island, an ongoing Android notification, a floating desktop pill. Both share the same layout model, serialization, state mechanism, and action model.

The simulator renders every surface you publish in a dedicated preview window, so the full loop runs on your desktop with no device:



46.1. The dead-process rule

Surfaces render while your app process may not be running. Everything you hand this API is therefore turned into plain data at publish time: layouts serialize to a compact JSON descriptor, images are encoded to named PNG blobs, and "callbacks" are string action ids that open the app and reach your handler on the EDT. Layout text embeds `${key}` placeholders resolved from a per-entry state map, so a content change is a cheap re-publish of data, not a new layout.

This rule explains everything unusual about the API. You can't attach a listener to a widget — the process that renders it may have nothing to call back into. You can't hand it a `Component` — there is no Codename One renderer on the other side. What you can do is describe the surface declaratively and let each platform's native renderer (SwiftUI on iOS, `RemoteViews` on Android, a rasterizer on desktop) draw it.

46.2. Getting started

The build injects the native plumbing (a WidgetKit extension and App Group on iOS, widget receivers on Android) when your bytecode references `com.codename1.surfaces`. Apps that never touch the package get none of it, and on ports without surface support every entry point is an inert no-op.

Platform widget galleries are compiled into the native app, so widget kinds must be known at build time. Declare them in a `surfaces.json` resource (in `src/main/resources` of a Maven project, next to your icons):

```
{
  "liveActivities": true,
  "kinds": [
    {
      "id": "delivery_status",
      "name": "Delivery",
      "description": "Track your order",
      "iosFamilies": ["systemSmall", "systemMedium"],
      "androidMinWidthDp": 180,
      "androidMinHeightDp": 60
    },
    {
      "id": "analog_clock",
      "name": "Analog Clock",
      "description": "A vector clock face",
      "iosFamilies": ["systemSmall"],
      "androidMinWidthDp": 110,
      "androidMinHeightDp": 110
    }
  ]
}
```

The `id` values must match `[a-z][a-z0-9_]*`. The `iosFamilies` list accepts both the portable names (`small`, `medium`, `large`, `lockscreen`) and the WidgetKit spellings (`systemSmall`, `systemMedium`, `systemLarge`, `accessoryRectangular`); when omitted, all three home-screen sizes are offered. The `androidMinWidthDp` / `androidMinHeightDp` / `androidResizeMode` fields fill the Android provider metadata. An optional top-level `appGroup` pins the iOS App Group id, and `"liveActivities": true` enables the live activity plumbing.

At runtime, mirror the manifest by registering each kind in your app's `init()`:

```
Surfaces.registerWidgetKind(new WidgetKind("delivery_status")
        .setDisplayName("Delivery")
        .setDescription("Track your order")
        .addSupportedSize(WidgetSize.SMALL)
        .addSupportedSize(WidgetSize.MEDIUM));
```

A widget renders a **timeline**: a shared layout plus dated entries whose state maps fill the layout's `${key}` placeholders. The OS flips entries on schedule without waking your app. This publishes a delivery tracker that advances through four states:

```
long now = System.currentTimeMillis();
long eta = now + 4 * 60000L;
WidgetTimeline timeline = new WidgetTimeline()
        .setContent(buildDeliveryLayout())
        .addEntry(new Date(now), deliveryState("Preparing your order", eta, 0.1f))
        .addEntry(new Date(now + 60000L), deliveryState("Out for delivery", eta, 0.4f))
        .addEntry(new Date(now + 3 * 60000L), deliveryState("Arriving now", eta, 0.9f))
        .addEntry(new Date(eta), deliveryState("Delivered", eta, 1f))
        .setReloadPolicy(WidgetTimeline.RELOAD_AT_END);
Surfaces.publish("delivery_status", timeline);
```

The layout itself is a small tree of nodes with shared styling (padding, background, corner radius, alignment, weight, size, action):

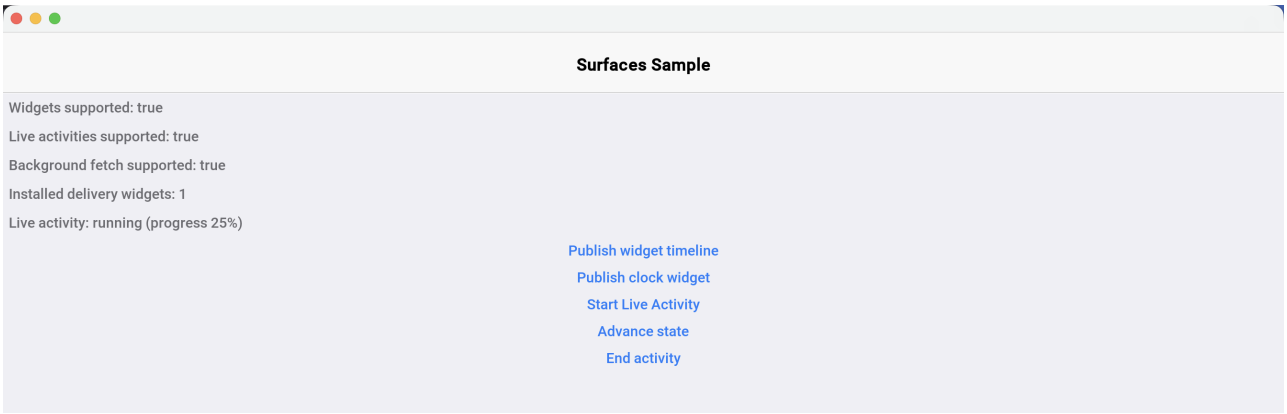
```
private SurfaceNode buildDeliveryLayout() {
    Map<String, Object> params = new HashMap<>();
    params.put("orderId", "CN1-12345");
    return new SurfaceColumn().setSpacing(6).setPadding(12)
            .add(new SurfaceRow().setSpacing(10)
                    .add(new SurfaceImage(courierAvatar)
                            .setSize(40, 40).setCornerRadius(20))
                    .add(new SurfaceColumn().setSpacing(2).setWeight(1)
                            .add(new SurfaceText("${status}")
                                    .setFontSize(15)
                                    .setFontWeight(SurfaceFontWeight.SEMIBOLD)
                                    .setMaxLines(1))
                            .add(new SurfaceDynamicText(
                                    SurfaceDynamicText.STYLE_TIMER_DOWN, "eta")
                                    .setFontSize(24)
                                    .setFontWeight(SurfaceFontWeight.BOLD)
                                    .setColor(SurfaceColor.ACCENT))))
            .add(new SurfaceProgress(SurfaceProgress.STYLE_LINEAR)
                    .setValueState("progress")
                    .setColor(SurfaceColor.rgb(ACCENT_COLOR)))
            .setAction("open_order", params);
}

private Map<String, Object> deliveryState(String status, long eta, float progress) {
    Map<String, Object> state = new HashMap<>();
    state.put("status", status);
    state.put("eta", eta);
    state.put("progress", progress);
    return state;
}
```

Three things in this layout carry the model's weight. `SurfaceText` renders `${status}` from the entry state, so updates ship a tiny state map instead of a layout. `SurfaceDynamicText` with `STYLE_TIMER_DOWN` is a countdown the OS animates on its own clock — the ETA ticks every second with zero app wake-ups. `SurfaceProgress` reads its value from the `progress` state key. The `setAction` call at the root makes a tap anywhere on the widget open the app and deliver the `open_order` action.

`Surfaces.publish(…)` is callable from any thread and never blocks on the EDT, which matters for background refresh later in this chapter.

A complete runnable example lives in `Samples/samples/SurfacesSample` together with its `surfaces.json`:



46.2.1. Previewing in the simulator

Open **Widgets** > **Widgets Preview** in the simulator. The window lists your registered kinds, renders the published timeline of the selected kind at any size in light or dark mode, flips timeline entries on schedule, and ticks countdowns exactly as a home-screen widget would. The mock Dynamic Island at the bottom renders running live activities. Clicks map through to your action handler, and a desktop (non-simulator) build renders the same publishes as floating widget windows pinned from a tray icon.

46.3. The node catalog

The catalog is kept small: a set of nodes every platform can render natively. Android app widgets (`RemoteViews`) are the constrained platform, so the catalog is designed to that floor and degrades as follows:

Node	iOS (SwiftUI)	Android (RemoteViews)	Notes
`SurfaceColumn` / `SurfaceRow`	`VStack` / `HStack`	`LinearLayout`	Weight maps to `layout_weight`
`SurfaceBox`	`ZStack`	`FrameLayout`	Child alignment via a 9-way enum

Node	iOS (SwiftUI)	Android (RemoteViews)	Notes
`SurfaceText`	`Text`	`TextView`	Android maps light/regular/medium weights to regular, semibold/bold to bold
`SurfaceDynamicText` timer	`Text(date, style:)`	`Chronometer`	Ticks natively on both
`SurfaceDynamicText` time	`Text(date, style: .time)`	`TextClock`	Native on both
`SurfaceDynamicText` date/relative	Native	Static text	Android refreshes it on the next update only
`SurfaceImage`	`Image`	`ImageView`	Named PNG blobs, content-hash dedup
`SurfaceProgress` linear	`ProgressView`	`ProgressBar`	Value 0..1 or a state key
`SurfaceProgress` circular	`Gauge`	Falls back to linear	Android widgets lack determinate circular progress
`SurfaceProgress` date interval	Native animation	Frozen at each refresh	iOS-only nicety
`SurfaceVector`	SwiftUI `Canvas`	Bitmap rendered in-process	Large vector nodes cost bitmap budget on Android
`SurfaceSpacer`	`Spacer`	Weighted `View`	Flexible gap
Corner radius	`clipShape`	Background drawable	May render square below Android 12
Node action	`Link` / `widgetURL`	`setOnClickPendingIntent`	Small iOS widgets honor only the root action

Descriptors are limited to 8 nesting levels. Keep payloads (JSON plus images) comfortably under 200kb — the iOS widget extension runs in about 30mb of memory, and Android parcels rendered widgets over a 1mb binder transaction.

46.4. Vector widgets

`SurfaceVector` covers the widgets the template catalog can't express — clocks, gauges, dials — with a retained list of vector drawing operations (fills, strokes, lines, arcs, text, rotation groups) replayed natively by every renderer. Rotation groups can read their angle from a state key, which turns a static drawing into a data-driven one.

The classic example is an analog clock. The face is resolution independent — the `200x200` view box scales to whatever size the widget gets:

```
private SurfaceNode buildClockFace() {
    SurfaceVector face = new SurfaceVector(200, 200)
            .fillEllipse(100, 100, 96, 96, SurfaceColor.rgb(0xfffafafa, 0xff1c1c1e))
            .strokeEllipse(100, 100, 96, 96, 4, SurfaceColor.rgb(ACCENT_COLOR));
    for (int hour = 0; hour < 12; hour++) {
        boolean quarter = hour % 3 == 0;
        face.beginRotation(hour * 30f, 100, 100)
                .line(100, 10, 100, quarter ? 24 : 18, quarter ? 5 : 3, SurfaceColor.LABEL)
                .endRotation();
    }
    face.beginRotation("hourAngle", 100, 100)
            .line(100, 100, 100, 52, 8, SurfaceColor.LABEL)
            .endRotation()
            .beginRotation("minuteAngle", 100, 100)
            .line(100, 100, 100, 26, 5, SurfaceColor.rgb(ACCENT_COLOR))
            .endRotation()
            .fillEllipse(100, 100, 6, 6, SurfaceColor.rgb(ACCENT_COLOR));
    return face;
}
```

Angles use the clock convention: degrees, 0 at 12 o'clock, clockwise positive. The face publishes once; 60 per-minute timeline entries carry only the two angle values, and the OS flips them on schedule. The clock stays correct for an hour with zero app wake-ups:

```
WidgetTimeline timeline = new WidgetTimeline()
        .setContent(buildClockFace())
        .setReloadPolicy(WidgetTimeline.RELOAD_AT_END);
Calendar c = Calendar.getInstance();
c.set(Calendar.SECOND, 0);
c.set(Calendar.MILLISECOND, 0);
long minuteStart = c.getTimeInMillis();
int baseMinutes = c.get(Calendar.HOUR_OF_DAY) * 60 + c.get(Calendar.MINUTE);
for (int m = 0; m < 60; m++) {
    int totalMinutes = baseMinutes + m;
    Map<String, Object> state = new HashMap<>();
    // clock convention: degrees, 0 = 12 o'clock, clockwise positive
    state.put("minuteAngle", totalMinutes % 60 * 6f);
    state.put("hourAngle", totalMinutes % 720 * 0.5f);
    timeline.addEntry(new Date(minuteStart + m * 60000L), state);
}
Surfaces.publish("analog_clock", timeline);
```



46.5. Live activities

A live activity is started explicitly from the app and updated by pushing fresh state. The descriptor carries the main content (the lock screen presentation on iOS, the notification layout on Android) plus the Dynamic Island regions: `setCompactLeading` / `setCompactTrailing` for the pill shown around the camera cutout, `setMinimal` for the detached circle when several activities run, and `setExpandedLeading` / `setExpandedTrailing` / `setExpandedCenter` / `setExpandedBottom` for the long-press expanded card.

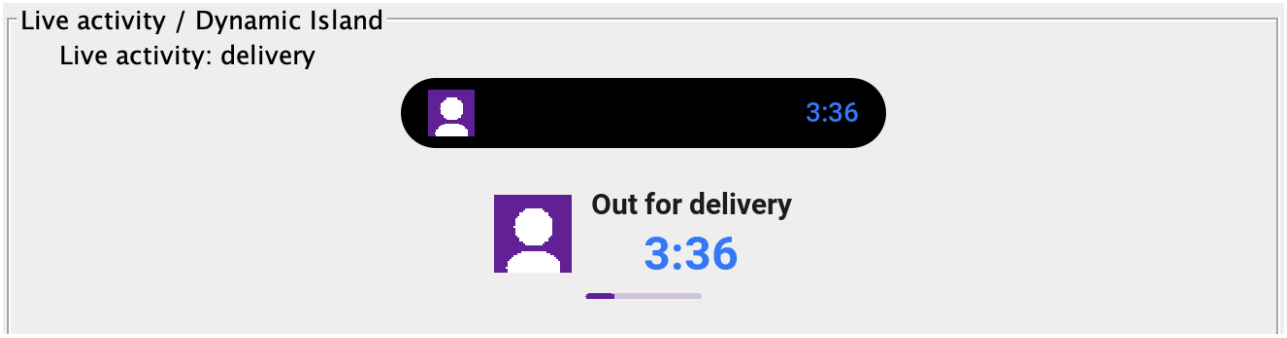
```
long eta = System.currentTimeMillis() + 4 * 60000L;
LiveActivityDescriptor descriptor = new LiveActivityDescriptor("delivery")
        .setContent(buildDeliveryLayout())
        .setCompactLeading(new SurfaceImage(courierAvatar)
                .setSize(24, 24).setCornerRadius(12))
        .setCompactTrailing(new SurfaceDynamicText(
                SurfaceDynamicText.STYLE_TIMER_DOWN, "eta")
                .setFontSize(14).setColor(SurfaceColor.ACCENT))
        .setExpandedCenter(new SurfaceText("${status}")
                .setFontSize(16).setFontWeight(SurfaceFontWeight.SEMIBOLD))
        .setExpandedBottom(new SurfaceProgress(SurfaceProgress.STYLE_LINEAR)
                .setValueState("progress")
                .setColor(SurfaceColor.rgb(ACCENT_COLOR)))
        .setTint(SurfaceColor.rgb(ACCENT_COLOR));
activity = LiveActivity.start(descriptor,
        deliveryState("Out for delivery", eta, 0.25f));
```

The returned handle is inert (`isActive()` returns `false`) on platforms without live activity support, so no platform checks are needed. Updates and the final state are plain state maps interpolated into the descriptor persisted at start time:

```
activity.update(deliveryState("Arriving now", eta, 0.9f));
// ... and when the delivery completes:
```

```
activity.end(deliveryState("Delivered", System.currentTimeMillis(), 1f));
```

The simulator preview renders the running activity as a mock Dynamic Island pill plus the expanded card:



On iOS the activity appears on the lock screen and, on supported devices, inside the Dynamic Island (ActivityKit requires iOS 16.1 or newer). On Android it lowers to an ongoing notification that renders the same content; on Android 13 and newer the first `start(…)` prompts for the notification permission, which the build declares for you, and `isSupported()` reports false once the user has turned that prompt down twice or has switched notifications off for the app. On desktop it appears in the simulator preview or as a floating window.

46.6. Actions and cold start

Every interactive node carries a string action id plus an optional parameter map, set with `setAction(id, params)`. Tapping the surface opens the app and delivers a `SurfaceActionEvent` to the single handler registered with `Surfaces.setActionHandler(…)`:

```
Surfaces.setActionHandler(evt -> {
    if ("open_order".equals(evt.getActionId())) {
        String orderId = (String) evt.getParams().get("orderId");
        showOrderForm(orderId);
    } else {
        Dialog.show("Surface Action", "Action " + evt.getActionId()
                + " from " + evt.getSource()
                + ", cold start: " + evt.isColdStart(), "OK", null);
    }
});
```

Actions are delivered on the EDT. When the tap launches a dead app, the event is queued across the cold start and delivered once your handler registers — which is why the registration belongs in `init()`. `SurfaceActionEvent.isColdStart()` tells you which path you are on, `getSource()` identifies the surface (widget kind or activity type), and `getParams()` returns the parameter map serialized with the action.

46.7. Updating from the background

Surfaces are updated from the running app in this release; push-driven updates are planned. Three

mechanisms keep content fresh without keeping the app open:

Timelines carry the future with them. Publish entries covering the hours ahead and the OS flips them on schedule — the clock above stays correct for an hour without a single wake-up. `WidgetTimeline.RELOAD_AT_END` (the default) asks the platform to request fresh content when the entries run out.

Background fetch re-publishes data. Implement `com.codename1.background.BackgroundFetch`, register with `Display.setPreferredBackgroundFetchInterval(int)`, and re-publish inside the callback — publishing is data-only file IO, safe from any thread and from a UI-less process:

```
public void performBackgroundFetch(long deadline, Callback<Boolean> onComplete) {
    // fetch fresh data, then re-publish the timeline
    publishDeliveryTimeline();
    onComplete.onSucess(Boolean.TRUE);
}
```

The completion callback must be invoked, or iOS stops granting fetch time. On iOS, background fetch also needs the `fetch` background mode:

```
codename1.arg.ios.background_modes=fetch
```

On Android, the widget pulls the app. When a provider renders an exhausted `RELOAD_AT_END` timeline (or no timeline at all) it starts the background fetch service directly — throttled to once per 15 minutes per kind, and only when the app declares background fetch. The iOS widget extension can't wake the app arbitrarily: an exhausted timeline makes WidgetKit re-read the persisted document, so publish timelines with enough future entries to bridge the gap between fetches. The simulator simulates background fetch with a timer that fires while the app is paused (**Simulate** > **Pause App**).

46.8. Per-platform notes

46.8.1. iOS

The build generates a `CN1Widgets` WidgetKit extension target into the Xcode project and shares published timelines with it through an App Group container. The App Group id defaults to `group.<your package name>`; override it with the `appGroup` field in `surfaces.json` or the `ios.surfaces.appGroup` build hint. The extension's deployment target defaults to 16.1 (ActivityKit's floor) without raising your app's own deployment target — on older iOS versions the extension never runs and `Surfaces.areWidgetsSupported()` returns `false`.

Signing follows the generic app extension rules: the extension needs its own App ID (`<your package>.CN1Widgets`) with the App Groups capability and, for cloud device builds with manual signing, its own provisioning profile. Either place the `.mobileprovision` under `common/src/main/resources` or supply a URL with the `ios.appext.CN1Widgets.provisioningURL` build hint. With automatic (managed) signing and in local `ios-source` builds no extra configuration is

needed — Xcode provisions the extension target as usual.

Like the app itself (`codename1.ios.debug.provision` / `codename1.ios.release.provision`), the extension needs a *development* profile for debug device builds and a *distribution* profile for release builds. Qualify any extension provisioning key with `debug` or `release` after the `ios.` segment and the build picks the variant matching the build target, falling back to the unqualified key when no matching variant is set:

- Build hints: `codename1.arg.ios.debug.appext.CN1Widgets.provisioningURL` and `codename1.arg.ios.release.appext.CN1Widgets.provisioningURL`
- Local profile paths in `codenameone_settings.properties`: `codename1.ios.debug.appext.CN1Widgets.provision` and `codename1.ios.release.appext.CN1Widgets.provision`

The Certificate Wizard populates both `provision` variants automatically when it sets up widget extension signing (the development profile requires at least one registered device). An unqualified key alone still works and is used for both build types.

Widget taps deep link back into the app through the `cn1surface://` URL scheme, which the build registers automatically.

46.8.2. Android

Widgets are rendered through `RemoteViews` by generated per-kind providers; no Android-specific build hints are needed, and the per-kind sizing metadata comes from `surfaces.json`. Timeline entry flips are scheduled with inexact alarms (a 30-second window) to avoid the exact-alarm permission by default; apps that need to-the-second flips can opt in with the `android.surfaces.exactAlarms` build hint. Second-precision countdowns still tick natively through `Chronometer`. Live activities lower to ongoing notifications, which on Android 13 and newer require the `POST_NOTIFICATIONS` runtime permission: the build declares it for you when `surfaces.json` sets `"liveActivities": true`, and the first `LiveActivity.start(…)` raises the system prompt, blocking the calling thread until the user answers. Codename One raises it at most twice across an install — Android stops showing the dialog after two refusals anyway — and spends an attempt only on a request it managed to issue. Android reports a dismissed dialog exactly as it reports a refusal, so dismissing one does cost an attempt, but one rather than the whole budget. Start the first activity while your app is in the foreground: there is no UI to prompt from in a background service or push handler, so a start from one before the permission is granted is refused without spending an attempt. `LiveActivity.isSupported()` is the programmatic signal once the answer has settled — a spent budget or, for an app that never declared the permission, a missing manifest entry — and `adb logcat -s CN1Surfaces` explains every refusal, reporting each settled reason once. A grant from anywhere counts — these prompts, push registration, `Display.requestNotificationPermission(…)` or the system settings — because the live permission state is always checked first. The approximations listed in the node catalog table apply: font weights collapse to regular/bold, circular progress falls back to linear, relative dates refresh only on entry flips, and vector nodes render as bitmaps.

46.8.3. Desktop, Windows, and Linux

In a desktop build the app shows a tray icon whose menu pins a floating widget per kind: a

frameless, always-on-top window rendering the published timeline, with clicks dispatched to your action handler. Window positions and the pinned set persist across runs, and a running live activity docks a pill window at the top of the primary screen. Desktop widgets are process-bound in this release — they exist while the app process runs. On Windows the plain signed executable ships these layered floating widgets with zero packaging; setting `windows.msix=true` additionally wraps the build in an MSIX package that declares a Windows 11 Widgets Board provider, so your kinds appear in the Win+W board rendered as Adaptive Cards. The MSIX channel is opt-in because it has real distribution prerequisites: a certificate the target machine trusts, the Windows App Runtime redistributable on the target machine, and Windows 11 for the board itself. On Linux the widgets are frameless GTK applet windows; on Wayland compositors that support the layer-shell protocol (KDE Plasma, Sway and the rest of the `wlroots` family) a runtime-loaded `gtk-layer-shell` places widgets above the wallpaper as real desktop applets with persistent positions and drag-to-move, and on GNOME Wayland they degrade to plain floating windows because the compositor controls global positioning and keep-above.

46.9. Build hints

Hint	Default	Purpose
`ios.surfaces.extension`	`true`	Set to `false` to skip the whole iOS lowering: no extension, no App Group, and the surfaces API reports unsupported at runtime
`ios.surfaces.appGroup`	`group.<package>`	The shared App Group id; the `appGroup` field in `surfaces.json` takes precedence
`ios.surfaces.deploymentTarget`	`16.1`	Deployment target of the widget extension (the host app is unaffected)
`ios.surfaces.frequentUpdates`	`false`	Adds `NSSupportsLiveActivitiesFrequentUpdates` for high-frequency live activity updates
`ios.appext.CN1Widgets.provisioningURL`		URL of the extension's provisioning profile for cloud manual-signing builds; used for both debug and release builds unless a qualified variant is set

Hint	Default	Purpose
`ios.debug.appext.CN1Widgets.provisioningURL` / `ios.release.appext.CN1Widgets.provisioningURL`		Build-type-specific variants: the URL of the development profile used by debug device builds and the URL of the distribution profile used by release builds. The variant matching the build target overrides the unqualified hint
`ios.background_modes`		Add `fetch` so background fetch can re-publish timelines on device
`android.surfaces.exactAlarms`	`false`	Schedule widget timeline entry flips with exact alarms; declares `SCHEDULE_EXACT_ALARM` and falls back to the inexact 30-second window when the user revokes the special app access
`windows.msix`	`false`	Wrap the Windows build in an MSIX package with a Widgets Board provider
`windows.msix.identityName`	package name	MSIX package identity name
`windows.msix.publisher`	`CN=<display name>`	MSIX identity publisher; must match the signing certificate subject
`windows.msix.version`	`1.0.0.0`	MSIX package version
`windows.msix.pfx` / `windows.msix.password`	executable signing configuration	Certificate used to sign the MSIX package

46.10. Current limitations

- Updates originate from the app (timelines, background fetch, live activity updates). Server-pushed widget content and ActivityKit push tokens are planned; the wire format already accommodates them.
- The node catalog is intentionally the lowest common denominator — there is no arbitrary per-pixel drawing beyond `SurfaceVector`, and no embedding of regular Codename One components.
- `WidgetSize.LOCKSCREEN` maps to the iOS `accessoryRectangular` family and is ignored on Android in this release.
- The Widgets Board provider requires the `windows.msix` opt-in and its distribution prerequisites; without it, Windows desktop widgets are floating windows.

Chapter 47. Maps

Codename One ships a modern mapping API in the `com.codename1.maps` package built around two components:

- `MapView` — a **pure-vector** map drawn entirely through the Codename One `Graphics` pipeline. It never embeds a native peer, so it composes cleanly with the rest of your UI (dialogs, lists and overlays draw over it) and behaves identically on every platform, including the simulator and the web.
- `NativeMap` — a map backed by a **native provider** (Apple MapKit, Google Maps, ...) when one is wired into the build, and which transparently **falls back to a `MapView`** when no provider is available (the simulator, devices without the selected provider, or builds that didn't opt in).

Both components implement the same `MapSurface` interface, so application code is identical regardless of which one — or which provider — is backing the map.



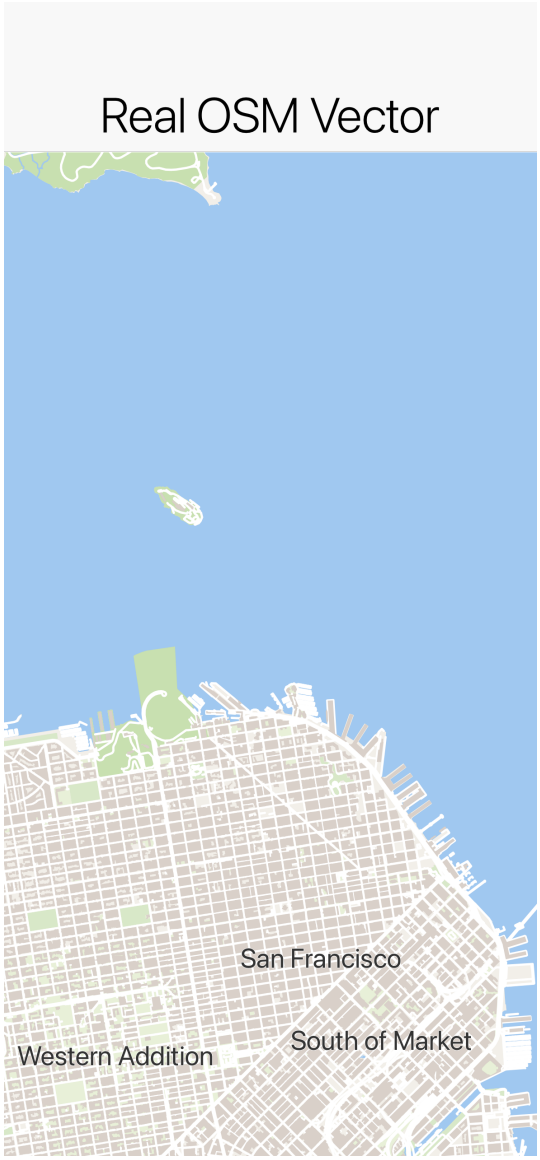
The legacy tile-based `MapComponent` and the external `codenameone-google-maps` cn1lib are deprecated in favor of this API.

47.1. A first map

The pure-vector `MapView` renders real maps with zero configuration and no API key — by default it shows the free, keyless OpenFreeMap [https://openfreemap.org] vector basemap (real OpenStreetMap data):

```
MapView map = new MapView();
map.moveCamera(new LatLng(37.7749, -122.4194), 12);
form.add(BorderLayout.CENTER, map);
```

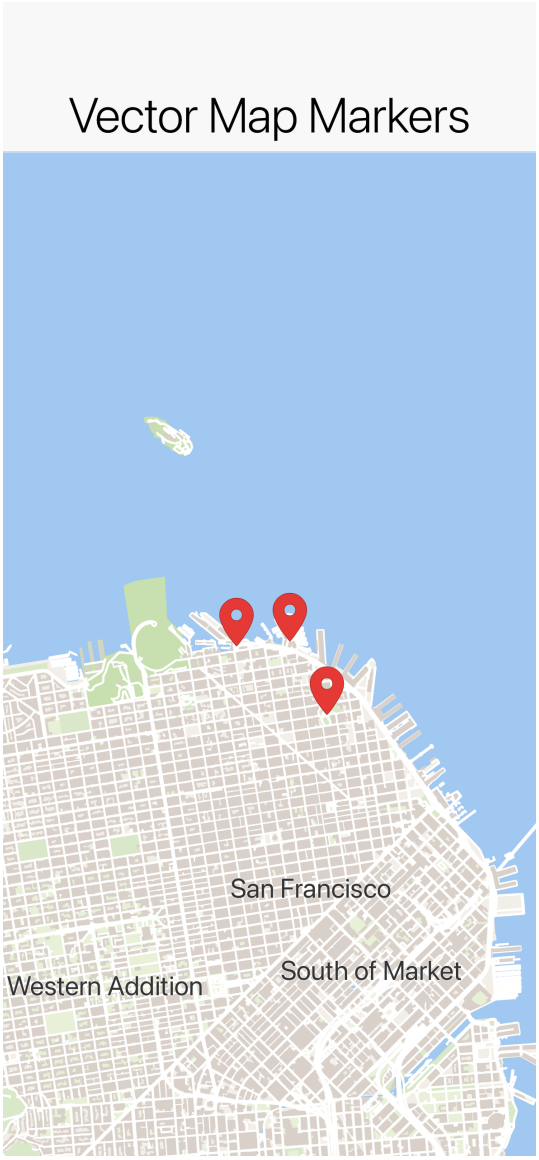
`LatLng` is the immutable WGS84 coordinate value type used throughout the API. The camera is described by a center `LatLng` and a fractional zoom level (the standard slippy-map scale where each whole increment doubles the scale).



47.2. The MapSurface API

Every map — vector or native — exposes the same operations through `MapSurface`:

Marker icons are supplied as `EncodedImage` and anchored in normalized `(u, v)` image space (`0.5, 1.0` puts the pin tip on the location). When no icon is given, a marker draws the standard Material Design map pin. Polygon and circle fills accept an `0xAARRGGBB` color so you can make them translucent.



47.3. Routes that follow the road

A polyline joins the coordinates you hand it with **straight** lines — it knows nothing about roads. Passing two endpoints to `addPolyline` therefore draws a single straight segment across the map, which is seldom what a navigation screen wants. Working out the road geometry between two points is a separate job, done by a routing service, and that's what the `com.codename1.maps.routing` package is for:

```
MapView map = new MapView();
form.add(BorderLayout.CENTER, map);

// Draws the driving route along the actual roads and frames it.
Routing.showRoute(map, origin, destination);
```

`Routing.showRoute(…)` fetches the route, draws it and moves the camera to frame it. Like the keyless OpenFreeMap basemap, it works with no API key and no signup: routing runs over OpenStreetMap data through the built-in `OsrmRouteService`.

Routing is asynchronous — the call returns immediately and the line appears when the service

answers. Use `Routing.findRoute(…)` when you want the result itself, for example to style the line or show the distance and the estimated time:

```
RouteRequest request = new RouteRequest(origin, destination)
        .addWaypoint(new LatLng(38.8899, -77.0091))
        .setTravelMode(TravelMode.DRIVING);

Routing.findRoute(request, new RouteCallback() {
    @Override
    public void routesFound(java.util.List routes) {
        Route best = (Route)routes.get(0);
        map.addPolyline(best.toPolyline().setStrokeColor(0xff5722).setStrokeWidth(6));
        map.fitBounds(best.getBounds(), CN.convertToPixels(4));
        label.setText((int)(best.getDistanceMeters() / 1000) + " km, "
                + (int)(best.getDurationSeconds() / 60) + " min");
    }

    @Override
    public void routeFailed(String message, Throwable error) {
        label.setText(message);
    }
});
```

A `Route` carries the drawable geometry (`toPolyline()`), the bounds to frame it with, the total distance and duration, and the `RouteLeg`/`RouteStep` breakdown behind a turn-by-turn list. A `RouteRequest` adds waypoints and a `TravelMode` (`DRIVING`, `WALKING`, `CYCLING`).



The default endpoint is OSRM's **public demo server**. It exists for development and demos: no SLA, rate limited, and it hosts the car profile, so walking and cycling requests are routed over driving data there. Before shipping, point the service at your own OSRM instance (or any OSRM-compatible endpoint) — nothing else in your code changes:

```
// Your own OSRM instance, or any OSRM-compatible endpoint.
Routing.setService(new OsrmRouteService("https://osrm.example.com"));
```

To route through a different provider entirely — Google Directions, Mapbox Directions, GraphHopper, your own server — implement `RouteService` and install it the same way. Every provider of that kind returns the route as an **encoded polyline**, which `Polyline.fromEncoded(geometry)` turns straight into something drawable (`PolylineCodec` also handles the `polyline6` variant used by Valhalla).

47.4. Tile sources and styles (MapView)

`MapView` pulls its tiles from a pluggable `com.codename1.maps.vector.TileSource`:

- `MvtTileSource` — networked Mapbox Vector Tiles (`.pbf`/`.mvt`). `MvtTileSource.openFreeMap()` is the

free, keyless OpenStreetMap-based default (its tile URLs are versioned, so it resolves them from OpenFreeMap's TileJSON automatically). Most other hosted vector basemaps require an API key supplied via `setApiKey(…)` and referenced as `{key}` in the URL template; a TileJSON endpoint (a URL with no `{z}` token) is resolved automatically.

- `RasterTileSource` — networked XYZ image tiles. `RasterTileSource.openStreetMap()` is a keyless raster alternative.
- `BundledTileSource` — tiles bundled into the app as resources, for fully offline maps. The screenshot tests in this repo use this against a small fixture set so the renders are deterministic and network-free; it's not how you ship a real map.
- `DemoTileSource` — a self-contained synthetic tile set (no network), handy for demos and deterministic tests.

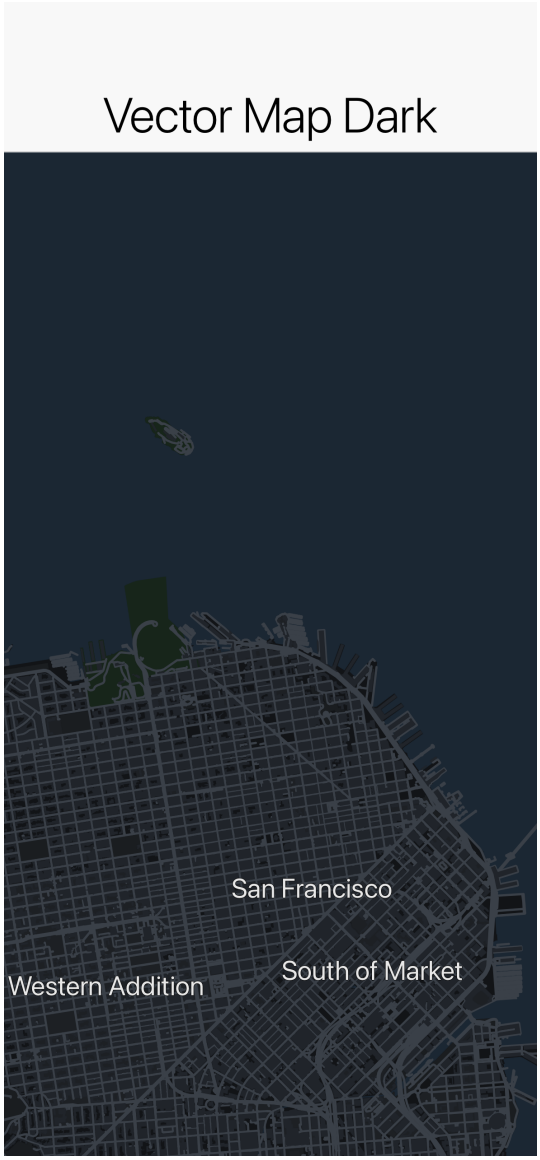
47.4.1. Connecting to real map data

For a real app you point `MapView` at a hosted tile service. The simplest keyless option is OpenFreeMap (OpenStreetMap-based vector tiles); for branded styles or higher quotas use a keyed provider such as MapTiler or your own self-hosted Protomaps/TileServer GL endpoint:

If the URL has no `{z}` token it's treated as a TileJSON document and the real tile-URL template (and min/max zoom) are resolved from it automatically, so you can paste a provider's TileJSON URL directly.

To bundle your own offline tiles instead, drop `.mvt`/`.pbf` files into the app's resources under a `{z}/{x}/{y}` layout and point `BundledTileSource` at them — that's exactly what the test fixtures do.

Vector tiles are painted according to a `MapStyle`. The built-in `MapStyle.light()` and `MapStyle.dark()` cover a usable basemap; `MapStyle.fromJson(json)` parses a subset of the MapLibre GL style specification (`background`/`fill`/`line`/`symbol` layers, zoom-stop interpolation, simple `["==", key, value]` filters).



47.5. Native maps and providers

`NativeMap` renders through a native map SDK when the build wires one in:

```
NativeMap map = new NativeMap(new LatLng(37.7749, -122.4194), 12);
map.addMarker(new MarkerOptions(new LatLng(37.7749, -122.4194)).title("San Francisco"));
form.add(BorderLayout.CENTER, map);

if (!map.isNativeMap()) {
    // Running on the simulator (or a build without a provider) -> vector fallback.
}
```



Which provider (if any) backs the map is decided entirely by **build hints** — the public API never names a provider, so unused providers add nothing to your app's size. Select one with the `maps.provider` build hint (or the per-platform `ios.maps.provider` / `android.maps.provider`):

```
# iOS uses Apple MapKit (a free system framework, no API key, no pod):
codename1.arg.ios.maps.provider=apple

# Android uses Google Maps (pulls Google Play Services Maps):
codename1.arg.android.maps.provider=google

# ...or Huawei Map Kit, for devices without Google Play Services (HMS Core).
# The build pulls com.huawei.hms:maps from Huawei's Maven repo automatically;
# your app still needs the usual HMS onboarding (agconnect-services.json / key):
codename1.arg.android.maps.provider=huawei
```

The built-in providers are Apple MapKit (iOS), Google Maps (Android) and Huawei Map Kit (Android). Huawei Map Kit runs on any Android device with HMS Core installed — not only Huawei hardware — so it's the natural choice for the Android market without Google Play Services. Each reports `isAvailable()` at runtime (Google checks Play Services, Huawei checks HMS Core), and when the selected provider is unavailable `NativeMap` falls back to the pure-vector `MapView`.

When a provider is selected the build server injects that provider's implementation into your app and wires it in; with no provider selected (or when the provider is unavailable at runtime, for example when Google Play Services is missing) `NativeMap` simply renders the vector `MapView` fallback. You can configure the fallback basemap explicitly:

47.5.1. Provider API keys

Apple MapKit needs no key. Google Maps requires the usual keys, supplied as build hints:

```
codename1.arg.android.xapplication=<meta-data android:name="com.google.android.maps.v2.API_KEY"
android:value="YOUR_ANDROID_API_KEY"/>
codename1.arg.ios.afterFinishLaunching=[GMSServices provideAPIKey:@"YOUR_IOS_API_KEY"];
```

47.5.2. Cross-platform web maps

A native provider only renders on the platforms that ship its SDK. To show a provider's map **everywhere** — including platforms with no native SDK for it — register a `WebMapProvider`, which

hosts the provider's JavaScript SDK inside a `BrowserComponent`:

Because it needs only a web view, the web provider (id `web`) is the natural last step before the pure-vector fallback in a provider chain. The order is fully customizable from code, so you can express per-platform preferences — for example "try the native Google SDK, then the web map, then the vector `MapView`":



47.6. Choosing between MapView and NativeMap

Use `MapView` when you want a lightweight, dependency-free map that looks the same everywhere and composes with CN1 components. Use `NativeMap` when you want the platform's native map look, gestures and performance, and are willing to opt a provider in through build hints — with the guarantee that it still works (as a vector map) where the provider is unavailable.

Chapter 48. In-Car Experiences (CarPlay & Android Auto)

Codename One can project a driver-safe UI onto Apple CarPlay and Google Android Auto from a single portable API under the `com.codename1.car` package. You describe the screens once with a small set of cross-platform templates; Codename One renders them as native `CPTemplate`s on CarPlay and `androidx.car.app` templates on Android Auto.



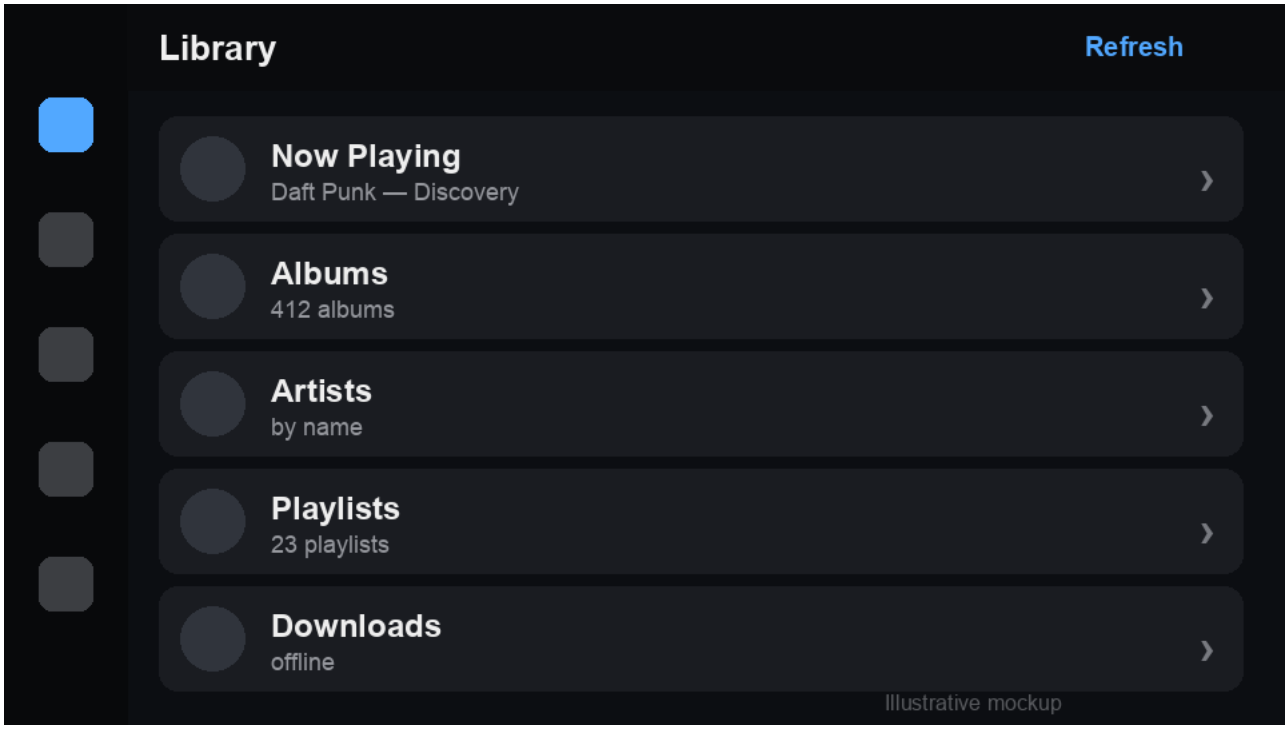
CarPlay and Android Auto are **template-based** UI systems. Unlike the phone, tablet, or even Apple TV/Android TV (which reuse the normal Codename One renderer), the car platforms don't let an app draw arbitrary pixels or show a regular Codename One `Form`. To minimise driver distraction the system only renders a fixed catalogue of templates — lists, grids, info panes, messages, navigation and now-playing — and grants access per **app category** (audio, communication, navigation, point-of-interest). The `com.codename1.car` API maps the common in-car use cases onto that catalogue. "Seamless" here means seamless integration, lifecycle and build wiring — not running your existing phone UI on the dashboard.

This API is **zero cost when unused**: simply referencing `com.codename1.car` is what tells the build to wire the native plumbing (the CarPlay scene + entitlement on iOS, the `androidx.car.app` dependency + `CarAppService` on Android). Apps that never touch the package get none of it.

48.1. Quick start

Register a single `CarApplication` from your app's `init()` — before a head unit connects — and return a root `CarScreen` that builds a template:

That single description renders natively on both platforms:



On the simulator and on any port without in-car projection there is no head unit, so the whole API is an inert no-op and `CN.isCarConnected()` returns `false` — your code never needs platform `if` statements.

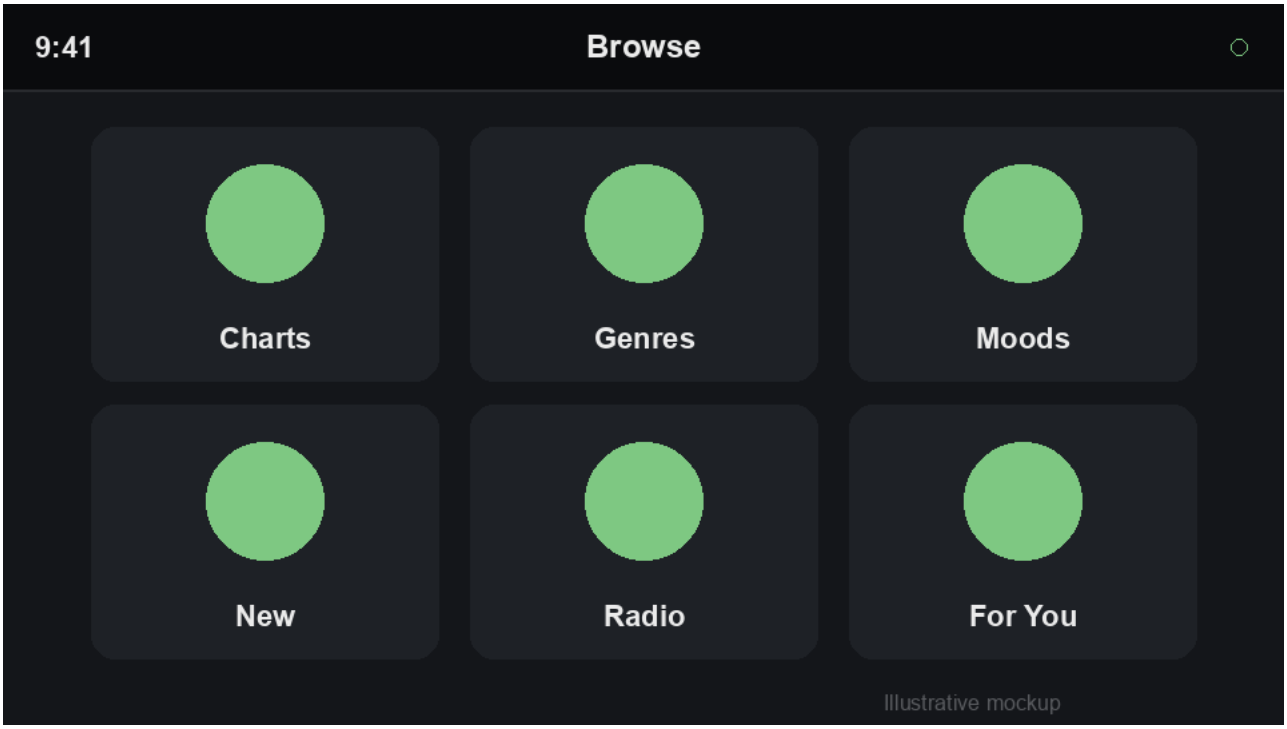
48.2. Templates

A `CarScreen` returns exactly one `CarTemplate`. The catalogue mirrors what the head units allow:

Template	Use for	CarPlay	Android Auto
`CarListTemplate`	Sectioned scrolling list of rows (browse, settings)	`CPListTemplate`	`ListTemplate`
`CarGridTemplate`	Image-forward grid (categories, presets, artwork)	`CPGridTemplate`	`GridTemplate`
`CarPaneTemplate`	Detail pane: label/value rows + up to two actions	`CPInformationTemplate`	`PaneTemplate`
`CarMessageTemplate`	Short message + actions (errors, prompts, messaging)	`CPInformationTemplate`	`MessageTemplate`
`CarNavigationTemplate`	Turn-by-turn surface with a drawable map	`CPMapTemplate`	`NavigationTemplate`
`CarNowPlayingTemplate`	System now-playing surface for audio apps	`CPNowPlayingTemplate`	media now-playing

Rows (`CarRow`), grid items (`CarGridItem`) and actions (`CarAction`) carry a title, optional secondary text,

an optional `com.codename1.ui.Image` icon and a `CarActionListener` invoked on the EDT when the element is selected. Mark a row `setBrowsable(true)` when selecting it drills into a deeper screen.



Head units enforce a hard cap on the number of rows/items they display (driver-distraction rules). Query it and trim accordingly:

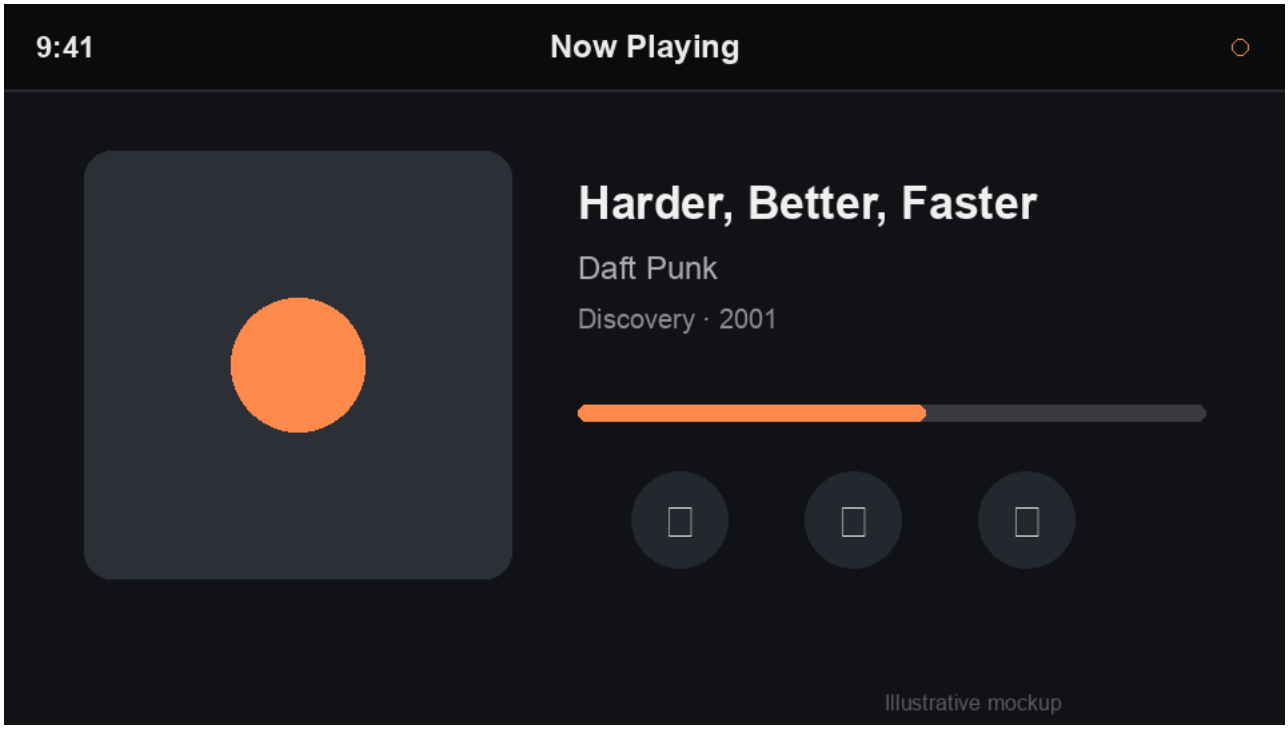
48.3. Screen stack & lifecycle

`CarContext` manages a back stack, mirroring `androidx.car.app`'s `ScreenManager` and CarPlay's `CPInterfaceController`:

`CarScreen` exposes optional lifecycle hooks — `onCreate()`, `onResume()`, `onPause()`, `onDestroy()` — and `CarApplication` is notified of connection changes via `onCarConnected(CarContext)` / `onCarDisconnected()`. You can also observe connection globally:

48.4. Now-playing (audio apps)

For audio apps the track metadata, artwork, scrubber and transport controls are driven by the platform media session (the Android background audio service / `MPNowPlayingInfoCenter` on iOS), so `CarNowPlayingTemplate` carries no metadata of its own — it routes the head unit to the now-playing surface and lets you add a couple of app-specific buttons (shuffle, like).



48.5. Build configuration

Native wiring is added automatically when your bytecode references `com.codename1.car`. Apple grants CarPlay entitlements — and Android Auto surfaces categories — **per app category**, so declare the categories you ship with these build hints:

Category	iOS hint	Android hint	Notes
Audio	`ios.carplay.audio=true`	(media browser service)	Default CarPlay entitlement when none specified
Communication	`ios.carplay.messaging=true`	`android.androidAuto.messaging=true`	Read-aloud / voice reply
Navigation	`ios.carplay.navigation=true`	`android.androidAuto.navigation=true`	Adds the drawable map surface + `MAP_TEMPLATES` permission
Point of interest	`ios.carplay.poi=true`	`android.androidAuto.poi=true` (default)	Browse/POI list apps

Optional: `android.carAppVersion` pins the `androidx.car.app` version; `android.androidAuto.minCarApiLevel` sets the minimum car API level.



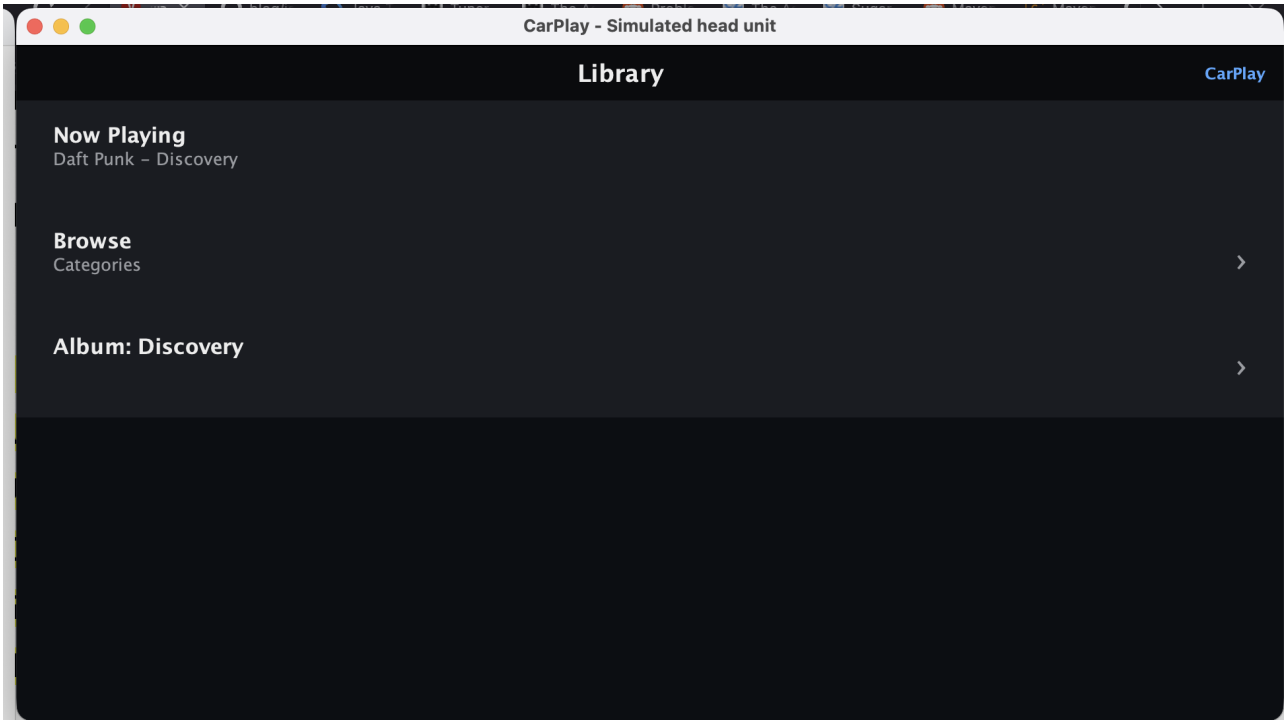
CarPlay and the restricted Android Auto categories require approval from Apple/Google before they will run outside the simulator/Desktop Head Unit. The build wiring is automatic, but shipping to real cars still needs the relevant CarPlay entitlement on your Apple App ID (request it at developer.apple.com) and acceptance into the Android Auto programme.

48.6. Debugging in the simulator

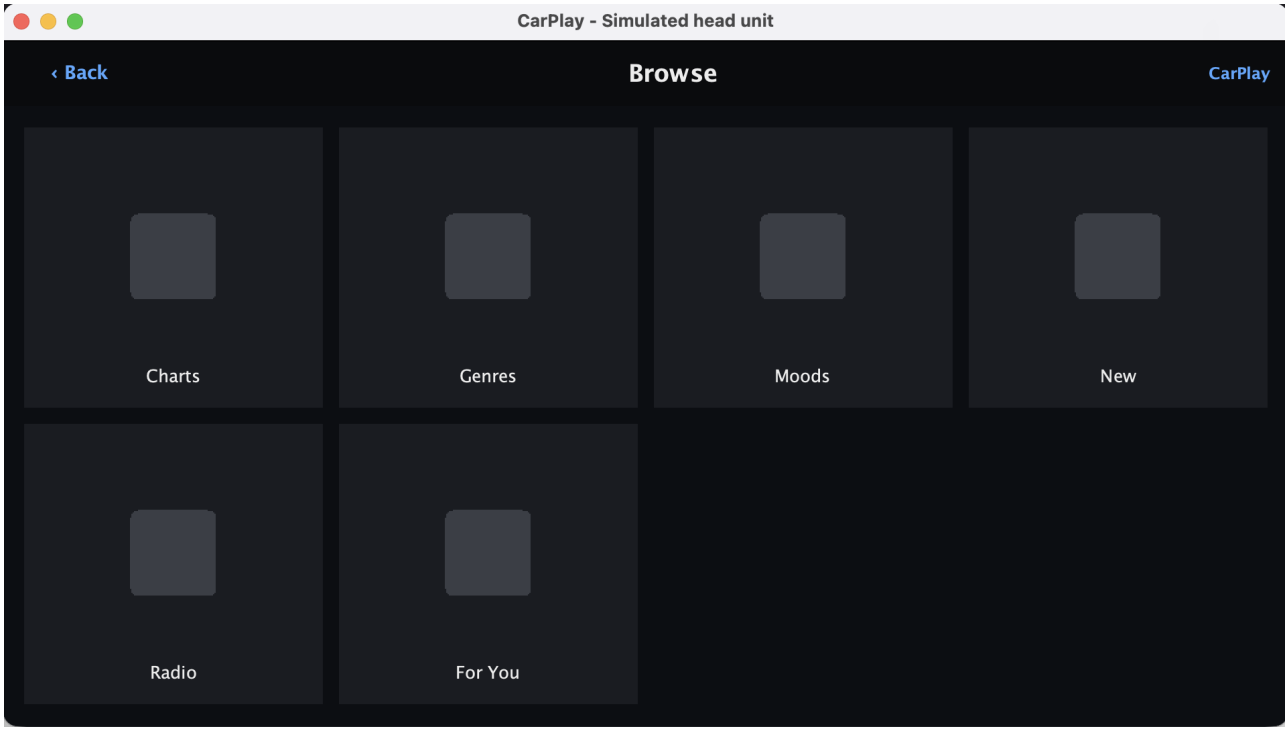
The fastest way to build and debug an in-car experience is the Codename One simulator. Run your app as usual, then use the **Car** menu:

- **Car** > **Connect CarPlay** or **Car** > **Connect Android Auto** opens a simulated head-unit window that renders your registered `CarApplication` in the matching style. Click rows, grid items and actions to drive navigation (`pushScreen`/`popScreen`) exactly as a real head unit would; the **Back** button pops the stack.
- **Car** > **Disconnect** (or closing the window) ends the session and fires `onCarDisconnected()`.

This needs no iOS/Android toolchain, no entitlement and no head unit — it exercises the same portable template tree and the same `CarActionListener` callbacks (on the EDT) that ship to the device, so you can iterate on your screens in seconds. If you haven't called `Car.setApplication(…)` the menu tells you so. The simulated head unit approximates the two platform styles and the driver-distraction row caps; it's a debugging aid, not a pixel-accurate emulation.



Selecting a browsable row pushes the next screen, just as it would in the car — here a `CarGridTemplate`, reachable via the **Back** button:



48.7. Testing on the real platforms

- **iOS**: run the app in the iOS Simulator, then **I/O** > **External Displays** > **CarPlay** to open the CarPlay window.
- **Android**: install the **Desktop Head Unit (DHU)** from the Android SDK and connect a device/emulator running Android Auto.

A complete runnable example lives in `Samples/samples/CarSample`.

48.8. Limitations

- The navigation map **surface** (`CarSurfaceCallback`) is scaffolded; live map drawing onto the head-unit surface is a work in progress — the navigation template's controls, header and ETA strip are wired.
- Android Auto media browsing is driven by the standard media session; `CarNowPlayingTemplate` routes to it.
- You can't show arbitrary Codename One forms on the head unit — use the template catalogue above.

Chapter 49. Motion Sensors

The `com.codename1.sensors` package exposes the device motion hardware through a single entry point, `MotionSensorManager`, and layers high-level gesture detection on top of the raw sensor stream. The raw sensors are the accelerometer, the gyroscope and the magnetometer. From those readings the framework derives the gravity vector, the linear acceleration and the device orientation, and it recognizes a set of common gestures: shake, flip, tilt, pick up and free fall.

Gesture detection lives in the core rather than in each platform port. Any platform that reports an accelerometer therefore gets the full gesture set, with no per-device work.

49.1. Reading a sensor

`MotionSensorManager.getInstance()` always returns a manager. On a device without motion hardware every sensor reports as unsupported, so application code can call the API without a null check on the manager itself. `getSensor(int)` returns `null` when the requested sensor isn't available on the current device:

Callbacks arrive on the EDT, so the listener can update the UI directly. Sampling is reference counted: the hardware sensor draws power only while at least one listener is registered. Remove the listener once the screen is no longer visible, typically from the form's `removeNotify`, so the sensor powers down:

49.2. Sensor types

Each sensor is identified by a `TYPE_*` constant on `MotionSensorManager`. The three hardware sensors map directly to the device. The gravity, linear acceleration and orientation values are derived in the core from the accelerometer (and the magnetometer, for the compass azimuth) when the platform doesn't report them on its own.

Constant	Reading
`TYPE_ACCELEROMETER`	Acceleration including gravity, in meters per second squared.
`TYPE_LINEAR_ACCELERATION`	Acceleration with gravity removed, in meters per second squared.
`TYPE_GRAVITY`	The gravity vector on its own, in meters per second squared.
`TYPE_GYROSCOPE`	Rate of rotation around each axis, in radians per second.
`TYPE_MAGNETOMETER`	The ambient magnetic field, in microtesla.
`TYPE_ORIENTATION`	Azimuth, pitch and roll, in radians.

Readings use a single convention across every port: the x-axis points to the right, the y-axis points up, and the z-axis points out of the screen toward the user, with the device held upright in portrait. At rest, face up, the accelerometer reports about `+9.81` on the z-axis.

49.3. Detecting gestures

Register a `GestureListener` for one of the `GestureEvent.TYPE_*` gestures. The accelerometer powers on for the duration that a gesture listener stays registered:

The recognized gestures are:

- `TYPE_SHAKE` — the device was shaken back and forth.
- `TYPE_FLIP_FACE_DOWN` and `TYPE_FLIP_FACE_UP` — the device was turned screen down or screen up.
- `TYPE_TILT_LEFT`, `TYPE_TILT_RIGHT`, `TYPE_TILT_FORWARD` and `TYPE_TILT_BACKWARD` — the device crossed a tilt angle on the roll or pitch axis.
- `TYPE_PICK_UP` — the device was lifted from a resting position.
- `TYPE_FREE_FALL` — the device is falling (near weightlessness was detected).

The detection thresholds suit most apps as shipped, and you can tune them when a gesture needs to be more or less sensitive:

A smaller sampling interval makes gestures more responsive at a higher battery cost. The value is clamped to the range 10ms through 1000ms.

49.4. Device orientation

`TYPE_ORIENTATION` is a derived sensor. Its event carries the azimuth, pitch and roll in radians, available through `getAzimuth()`, `getPitch()` and `getRoll()` on `MotionEvent`. Pitch and roll come from the gravity vector and need only the accelerometer. The azimuth is a compass heading and needs the magnetometer; on a device without one the azimuth stays at zero.

49.5. Platform notes

- **Android** — backed by `android.hardware.SensorManager`. Android already reports its readings in the units and axis convention this API documents, so no conversion takes place.
- **iOS** — backed by CoreMotion (`CMMotionManager`). Add the build hint `ios.NSMotionUsageDescription` with a short sentence describing why the app reads motion data; iOS denies access to the motion hardware without it. The iOS Simulator doesn't expose an accelerometer, so the sensors report as unsupported there. Test motion features on a real device.
- **Other ports** — a port that doesn't implement the sensors returns an unsupported manager, so the API stays safe to call everywhere.

49.6. Testing in the simulator

The simulator can stand in for real hardware. Open **Simulate** > **Motion** / **Gesture Simulation** to show a small control window. The two sliders set the simulated device orientation, which the accelerometer reflects, and the buttons inject the motion needed to fire the shake, flip, pick up and free fall gestures. This lets you wire up and debug a gesture handler on the desktop before deploying to a device.

Chapter 50. Miscellaneous features

50.1. Phone functions

Most of the low-level phone functionality is accessible in the Display [https://www.codenameone.com/javadoc/com/codename1/ui/Display.html] class. Think of it as a global class that covers access to the "system."

50.1.1. SMS

Codename One supports sending SMS messages but not receiving them as this functionality isn't portable. You can send an SMS using either the `Display` singleton or the static helper methods on `CN`. Both APIs throw `IOException`, which you should handle in case the native layer reports a failure:

```
try {
    Display.getInstance().sendSMS("+999999999", "My SMS Message");
    // Or: CN.sendSMS("+999999999", "My SMS Message");
} catch(IOException err) {
    Log.e(err);
    Dialog.show("SMS Failed", "Unable to send the SMS", "OK", null);
}
```

Android supports sending SMS messages in the background without any UI. iOS doesn't provide that ability, so the best it can offer is to launch the native SMS app with your message composed for the user to send. Android supports that interactive flow as well (launching the OS native SMS app).

The default `sendSMS` API ignores that difference and works interactively on iOS while sending in the background for Android when the platform allows it.

The `getSMSSupport` API returns one of the following options:

- SMS_NOT_SUPPORTED - for desktop, tablet etc.
- SMS_SEAMLESS - `sendSMS` won't show a UI and will send in the background
- SMS_INTERACTIVE - `sendSMS` will show an SMS sending UI
- SMS_BOTH - `sendSMS` can support both seamless and interactive mode, this works on Android

The `sendSMS` can accept an interactive argument: `sendSMS(String phoneNumber, String message, boolean interactive)`

The last argument will be ignored unless `SMS_BOTH` is returned from `getSMSSupport` at which point you would be able to choose one way or the other. The default behavior (when not using that flag) is the background sending which is the current behavior on Android.

A typical use of this API would be something like this:

50.1.2. Dialing

Dialing the phone is pretty trivial. This should open the dialer UI without physically dialing the phone, as that's discouraged by device vendors.

You can dial the phone by using:

```
Display.getInstance().dial("+999999999");
```

50.1.3. Call detection

Codename One includes a generic call detection API through `Display.isCallDetectionSupported()` and `Display.isInCall()`.

This API is intentionally **best effort** and should be used as a UX hint (for example, to defer a non-critical animation while the app is interrupted).

On iOS, `isInCall()` is inferred from app interruption lifecycle events, which means it can report `true` for non-call interruptions (for example: Control Center, app switching, permission sheets), and some call flows may still be missed.

Because iOS toggles this flag during app lifecycle transitions, polling it with a `UITimer` can miss the interruption window entirely (timers are paused while the app is inactive, and the flag is reset when the app becomes active again). If you need to react, check it from lifecycle callbacks such as your app's `stop()`/`start()` flow instead of periodic polling.

On Android, call detection is unsupported because robust detection would require invasive telephony permissions that are intentionally avoided:

```
Display display = Display.getInstance();
if (display.isCallDetectionSupported() && display.isInCall()) {
    Log.p("Call interruption is currently active (heuristic)");
}
```

50.1.4. E-Mail

You can send an email through the platforms native email client with code such as this:

```
Message m = new Message("Body of message");
Display.getInstance().sendMessage(new String[] {"someone@gmail.com"}, "Subject of message", m);
```

You can add one attachment by using `setAttachment` and `setAttachmentMimeType`.



You need to use files from `FileSystemStorage` and **NOT** `Storage` files!

You can add more than one attachment by putting them directly into the attachment map for example:



Some features such as attachments etc. Don't work in the simulator but should work on iOS/Android

50.2. Larger text accessibility

Codename One can adapt to the operating system's larger text accessibility setting. The Display [https://www.codenameone.com/javadoc/com/codename1/ui/Display.html] APIs expose two signals: `isLargerTextEnabled()` indicates whether the user has enabled larger text, and `getLargerTextScale()` returns the scale factor that should be applied to fonts.

If you want the theme system to automatically scale its fonts when larger text is enabled, enable the UIManager [https://www.codenameone.com/javadoc/com/codename1/ui/plaf/UIManager.html] setting:

```
UIManager.getInstance().setUseLargerTextScale(true);
```

You can also enable this behavior in the theme by setting the `useLargerTextScaleBool` theme constant to `true`.

If you need to apply the scale manually for custom fonts or layout calculations, read the values directly:

50.2.1. How the scale is computed per platform

- **iOS:** the scale is the ratio between the body text size at the user's current Dynamic Type setting and the iOS standard 17pt body size, that is, `[UIFont preferredFontForTextStyle:UIFontTextStyleBody].pointSize / [UIFont systemFontSize]`. Scales for the standard slider stops range from `0.82×` (Extra Small) through `1.00×` (Large, default) up to `1.35×` (Extra Extra Large). When the user enables *Larger Accessibility Sizes* in Settings → Accessibility → Display & Text Size → Larger Text, the slider extends with five more stops whose scales reach as high as `3.12×`. These values match what every native iOS app sees from `UIFontMetrics`.
- **Android:** the scale is `Configuration.fontScale`, which is bucketed by Android (typically `0.85`, `1.0`, `1.15`, `1.30`).



Because the scale is multiplicative, a base font size that's already larger than the platform default will compound. For example, a `defaultFontSizeInt: 18` in your CSS is 6% larger than iOS's 17pt body, so at iOS Dynamic Type *Extra Extra Large* (`1.35×`) your text will render at `24.3pt` versus a native iOS app's `23pt`. If you intentionally use a base size larger than the platform default and you also enable `useLargerTextScaleBool`, expect the absolute size at high accessibility settings to be visibly larger than typical native apps. Many native iOS apps cap Dynamic Type via `UIFontMetrics maximumPointSize:` for non-text-heavy UIs to avoid layout breakage.

50.2.2. Previewing in the simulator

The JavaSE simulator includes a **Simulate → Larger Text** menu that lets you preview every iOS Dynamic Type stop (Extra Small through Accessibility 5) without leaving your desktop. The simulator returns the same scale factor that the corresponding iOS device would report, so this is the recommended way to verify how your UI behaves at each setting before testing on hardware. The selected stop is persisted across simulator restarts.



If your app looks correct in the simulator at *Large (default)* but fonts appear oversized on your iPhone with the Display & Brightness slider near the middle, check whether *Settings → Accessibility → Display & Text Size → Larger Text → Larger Accessibility Sizes* is enabled on the device. When that toggle is on, the system reports the larger *Accessibility* size categories whose scales begin at `1.65x` and exceed what the standard slider can reach.

50.3. Contacts API

The contacts API provides you with the means to query the phone's address book, delete elements from it and create new entries into it. To get the platform specific list of contacts you can use `String[] contacts = ContactsManager.getAllContacts();`

Notice that on some platforms this will prompt the user for permissions and the user might choose not to grant that permission. To detect whether this is the case you can invoke `isContactsPermissionGranted()` after invoking `getAllContacts()`. This can help you adapt your error message to the user.

Once you have a Contact [https://www.codenameone.com/javadoc/com/codename1/contacts/Contact.html] you can use the `getContactById` method, but the default method is a bit slow if you want to pull a large batch of contacts. The solution for this is to extract the data that you need through

Here you can specify true for the attributes that actually matter to you.

Another capability of the contacts API is the ability to extract all the contacts. This isn't supported on all platforms but platforms such as Android can get a boost from this API as extracting the contacts one by one is slow on Android.

You can check if a platform supports the extraction of all the contacts through `ContactsManager.isAllContactsFast()`.



When retrieving all the contacts, notice that you should probably not retrieve all the data and should set some fields to false to perform a more efficient query

You can then extract all the contacts using code that looks a bit like this, notice that you use a thread so the UI won't be blocked!

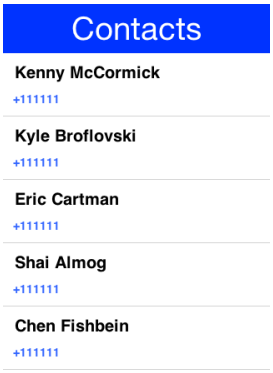


Figure 280. List of contacts

Notice that you didn't fetch the image of the contact as the performance of loading these images might be prohibitive. You can enhance the code above to include images by using slightly more complex code such as this:



The `scheduleBackgroundTask` method is like `new Thread()` in some regards. It places elements in a queue instead of opening too many threads so it can be good for non-urgent tasks

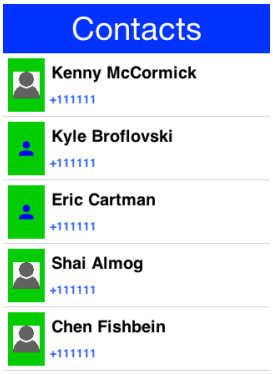


Figure 281. Contacts with the default photos on the simulator, on device these will use actual user photos when available



Notice that the code above uses `callSerially` & `scheduleBackgroundTask` in a liberal nested way. This is important to avoid an EDT violation

You can use `createContact(String firstName, String familyName, String officePhone, String homePhone, String cellPhone, String email)` to add a new contact and deleteContact(String id) to delete a contact.

50.4. Localization & internationalization (L10N & I18N)

Localization (l10n) means adapting to a locale which is more than translating to a specific language but also to a specific language within environment for example: `en_US!= en_UK`. Internationalization (i18n) is the process of creating one application that adapts to all locales and regional requirements.

Codename One supports automatic localization and seamless internationalization of an application using Java properties bundles.

Place locale-specific properties files inside an `l10n` directory directly under your module's `main` directory (for example: `common/src/main/l10n`). Each file inside this directory is treated as a resource bundle and will be packaged automatically with your app, allowing you to version translations alongside the rest of your source code.

When you're iterating on translations you can also use the simulator to capture bundles automatically. Open the Simulator menu and enable **Auto Update Default Bundle** so that running your app in the Codename One simulator will create any missing resource bundles on the fly as you interact with the UI, which makes it simple to populate keys without manually editing files during development. You can install the bundle using code like this:

The device language (as an ISO 639 two-letter code) could be retrieved with this:

```
String local = L10NManager.getInstance().getLanguage();
```

Once installed a resource bundle takes over the UI and every string set to a label (and label like components) will be automatically localized based on the bundle. You can also use the localize method of UIManager [https://www.codenameone.com/javadoc/com/codename1/ui/plaf/UIManager.html] to perform localization on your own:

```
UIManager.getInstance().localize( "KeyInBundle", "DefaultValue");
```

The list of available languages in the resource bundle could be retrieved like this. Notice that this a list that was set by you and doesn't need to confirm to the ISO language code standards:

An exception for localization is the `TextField`/`TextArea` components both of which contain user data, in those cases the text won't be localized to avoid accidental localization of user input.



You can export and import resource bundles as standard Java properties files, CSV and XML. The formats are pretty standard for most localization shops, the XML format Codename One supports is the one used by Android's string bundles which means most localization specialists should localize it

The resource bundle is a map between keys and values for example: the code below displays "`This Label is localized`" on the `Label` with the hardcoded resource bundle. It would work the same with a resource bundle loaded from a resource file:

```
Form hi = new Form("L10N", new BoxLayout(BoxLayout.Y_AXIS));
HashMap<String, String> resourceBudle = new HashMap<String, String>();
resourceBudle.put("Localize", "This Label is localized");
UIManager.getInstance().setBundle(resourceBudle);
hi.add(new Label("Localize"));
hi.show();
```



Figure 282. Localized label

50.4.1. Localization manager

The L10NManager [https://www.codenameone.com/javadoc/com/codename1/l10n/L10NManager.html] class includes a multitude of features useful for common localization tasks.

It allows formatting numbers/dates & time based on platform locale. It also provides a great deal of the information you need such as the language/locale information you need to pick the proper resource bundle:



Figure 283. Localization formatting/parsing and information

50.4.2. RTL/Bidi

RTL stands for right to left, in the world of internationalization it refers to languages that are written from right to left (Arabic, Hebrew, Syriac, Thaana).

Most western languages are written from left to right (LTR), but some languages are written from right to left (RTL) speakers of these languages expect the UI to flow in the opposite direction otherwise it seems weird like reading this word would be to most English speakers: "drieW."

The problem posed by RTL languages is known as BiDi (Bi-directional) and not as RTL since the "true" problem isn't the reversal of the writing/UI but rather the mixing of RTL and LTR together. For example: numbers are always written from left to right (like in English) so in an RTL language the direction is from right to left and once you reach a number or English text embedded in the middle of the sentence (such as a name) the direction switches for a duration and is later restored.

The main issue in the Codename One world is in the layouts, which need to reverse on the fly. Codename One supports this through an RTL flag on all components that's derived from the global `RTL` flag in UIManager [https://www.codenameone.com/javadoc/com/codename1/ui/plaf/UIManager.html].

Resource bundles can also include special case constant @rtl, which indicates if a language is written from right to left. This allows everything to automatically reverse.

When in `RTL` mode the UI will be the exact mirror so `WEST` will become `EAST`, `RIGHT` will become `LEFT` and this would be true for paddings/margins as well.

If you have a special case where you don't want this behavior you will need to wrap it with an `isRTL` check. You can also use `setRTL` on a per `Component` basis to disable RTL behavior for a specific `Component`.



Most UI APIs have special cases for BiDi instead of applying it globally for example: AWT introduced constants such as `LEADING` instead of making `WEST` mean the opposite direction. You think that was a mistake since the cases where you wouldn't want the behavior of automatic reversal are rare.

Codename One's support for bidi includes the following components:

- **Bidi algorithm** - allows converting between logical to visual representation for rendering
- **Global RTL flag** - default flag for the entire application indicating the UI should flow from right to left
- **Individual RTL flag** - flag indicating that the specific component/container should be presented as an RTL/LTR component (for example: for displaying English elements within an RTL UI).
- **RTL text field input**

Most of Codename One's RTL support is under the hood, the LookAndFeel [https://www.codenameone.com/javadoc/com/codename1/ui/plaf/LookAndFeel.html] global RTL flag can be enabled using:

```
UIManager.getInstance().getLookAndFeel().setRTL(true);
```

Once RTL is activated all positions in Codename One become reversed and the UI becomes a mirror of itself. For example: Adding a `Toolbar` command to the left will actually make it appear on the right. Padding on the left becomes padding on the right. The scroll moves to the left etc.

This applies to the layout managers (except for group layout) and most components. Bidi is seamless in Codename One but a developer still needs to be aware that his UI might be mirrored for these cases.

50.4.3. Localizing native iOS strings

Some strings in iOS need to be localized using iOS's native mechanisms - namely providing **.lproj* directories with *.strings* files. For example, if you want the app to have a different bundle display name for each language, or you want to translate the "UsageDescription" strings of your Info.plist into many languages, you would need to use iOS' native localization facilities [https://developer.apple.com/localization/].

Example: Localizing the app name

The app name, as it's displayed to the user, is defined in using the *CFBundleDisplayName* key of the app's Info.plist file., this will be automatically set to your app's display name, as defined in your *codenameone_settings.properties* file. This works fine if your app will have the same name in every locale, but suppose you want your app to take on a different name in French than in English. For example: You want your app to be called "Hello App" for English-speaking users, and "Bonjour App" for French-speaking users.

In this case, you need to add iOS localization bundles "en.lproj" and `fr.lproj`, each with a file named `InfoPlist.strings`. If you're using Maven, then you can add these directly inside the *ios/src/main/strings* directory of your project.



You will need to create the *strings* directory manually, if it doesn't exist yet.

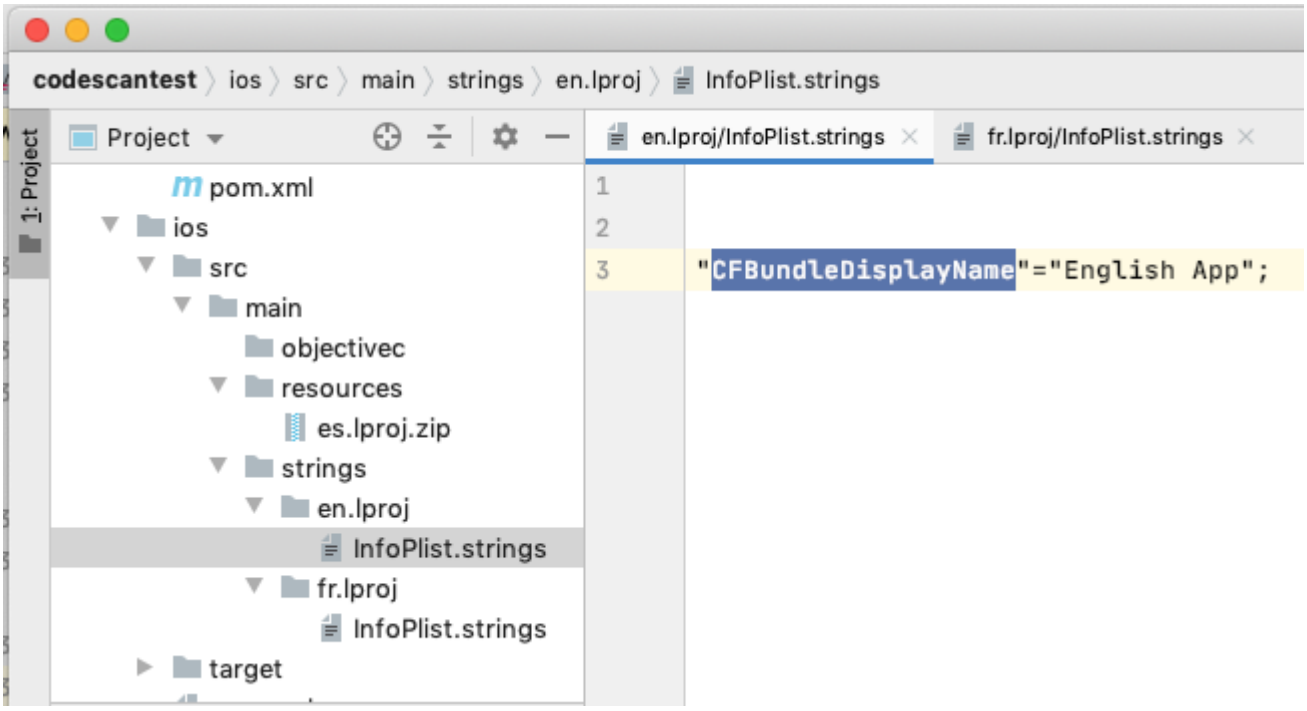


Figure 284. Maven project with English, French, and Spanish localizations for Info.plist. English and French language bundles are contained in the ios/src/main/strings directory. The Spanish bundle is included as a zip file in ios/src/main/resources. Both methods are supported (zipped in resources and unzipped in strings).

Listing 15. ios/src/main/strings/en.lproj/InfoPlist.strings

```
"CFBundleDisplayName"="Hello App";
```

Listing 16. ios/src/main/strings/en.lproj/InfoPlist.strings

```
"CFBundleDisplayName"="Bonjour App";
```



The *strings* format is like the *properties* file format, except that both the "key" and the "value" must be wrapped in quotes. And if there are many strings, then they must be delimited by a semi-colon `;`.

Legacy Ant Projects

Legacy Ant projects have a different directory structure. They have no equivalent location to the Maven *ios/src/main/strings* directory, but the legacy `ios/src/main/resources` content can be replicated under *native/ios*. To include native iOS localizations in those projects, place zipped versions of your .lproj directories inside the `native/ios` directory. For example, *en.lproj.zip*, *fr.lproj.zip*, etc.



Whenever possible, migrate legacy builds to Maven to take advantage of the modern workflow. See Chapter 1, *Maven Project Workflow* for automated migration options.

Example: Localization app usage description strings

iOS requires you to supply usage descriptions for many features that will be displayed to the user when the app requests permission to use the feature. For example, the NSCameraUsageDescription [https://developer.apple.com/documentation/bundleresources/information_property_list/nscamerausagedescription?language=objc] string must be provided if your app needs to use the camera. You can specify these values as build hints using the pattern `ios.NSXXXUsageDescription=This feature is needed blah blah`. In the `NSCameraUsageDescription` case, you might include the build hint:

```
ios.NSCameraUsageDescription=This app needs to use your camera to scan bar codes
```

These descriptions are embedded in your app's Info.plist file, so they can be localized the same way you localize other Info.plist values - in the localized `InfoPlist.strings` file.

See the above example for instructions on localizing values in the Info.plist file. Then add translations to the `InfoPlist.strings` file for your usage descriptions.

Listing 17. ios/src/main/strings/en.lproj/InfoPlist.strings

```
"CFBundleDisplayName"="Hello App";
"NSCameraUsageDescription"="This app needs to use your camera to scan bar codes";
```

Listing 18. ios/src/main/strings/en.lproj/InfoPlist.strings

```
"CFBundleDisplayName"="Bonjour App";
"NSCameraUsageDescription"="Cette application doit utiliser votre appareil photo pour scanner les codes à barres";
```

50.4.4. Localizing the app icon

Codename One can ship a different launcher icon per locale on iOS and Android by detecting specially named PNG files during the build. Drop your per-locale artwork into `common/src/main/resources` using the convention:

```
cn1_icon_<lang>[_<country>].png
```

`<lang>` is a two-letter ISO 639-1 language code and the optional `<country>` is a two-letter ISO 3166-1 country code. Language is treated as case-insensitive; the country component is normalized to uppercase. A few valid examples:

- `cn1_icon_fr.png`: French (any region)
- `cn1_icon_en_GB.png`: British English

- `cn1_icon_es_MX.png`: Mexican Spanish

Supply a square source image at least 432×432 pixels (the largest size emitted for Android adaptive icons; 1024×1024 is recommended, matching the main app icon); the build resizes it to every target density. A smaller source would be upscaled and look blurry, so the build now fails with a clear error when it detects an undersized localized icon rather than shipping a soft one. The default app icon continues to be controlled by your `codenameone_settings.properties` file and is used whenever the device locale doesn't match any of the localized variants.



These icons live under `src/main/resources` and Maven copies them into `target/classes` incrementally — it **doesn't** delete the copy when you remove or rename the source file. If you replace a localized icon and still see the old (often blurry) one, run `mvn clean` to clear the stale resource from `target/classes` before rebuilding. The build also fails when it finds an undersized `cn1_icon_*.png` in the compiled output that's about to be bundled and sent to the build server.

At runtime the builders look for a `<lang>_<COUNTRY>` match first, then fall back to a bare `<lang>` match. Providing both (for example `cn1_icon_en.png` plus `cn1_icon_en_GB.png`) lets you give British users a country-specific icon while every other English locale still receives the generic English icon.

Android behavior

On Android the build generates locale-qualified drawable resources at every density so the platform picks the right icon automatically based on the device's current locale:

- `drawable-<lang>[-r<COUNTRY>][-<dpi>]/icon.png` at 36, 48, 72, 96, 128, 144 and 192 pixels for legacy launchers.
- When `android.enableAdaptiveIcons=true` the same variants are also written as `mipmap-<lang>[-r<COUNTRY>]-<dpi>/ic_launcher.png` and `ic_launcher_foreground.png`. The default adaptive background and the `mipmap-anydpi-v26/ic_launcher.xml` definition are reused as-is.

No code changes are required—Android's resource framework switches icons when the locale changes.

When you supply a region-qualified icon (such as `cn1_icon_ar_AE.png`) without a matching language-only variant, the build also emits the **default** (non-localized) icon into `drawable-<lang>/` and the matching `mipmap-*-<lang>/` directories. This barrier is required because Android's resource resolver (API 24+) walks every child of the parent locale when it can't find an exact or parent-language match, and would otherwise pick `ar-rAE` for, say, an `ar-PK` device. The barrier short-circuits that lookup so only devices whose region matches the supplied variant receive the localized icon. If you also ship a language-only file (for example `cn1_icon_ar.png`) it's used as the barrier instead, so you keep full control of the fallback icon for Arabic speakers outside AE.

iOS behavior

iOS doesn't localize launcher icons natively, so Codename One wires up alternate app icons [https://developer.apple.com/documentation/uikit/uiapplication/2806818-setalternateiconname] for you:

- For each detected locale the build generates an `AppIcon_<lang>_<COUNTRY>.appiconset` inside

`Images.xcassets` containing the 120×120, 180×180, 152×152 and 167×167 (iPad Pro) PNGs along with a matching `Contents.json`.

- The build adds the matching `ASSETCATALOG_COMPILER_ALTERNATE_APPICON_NAMES` Xcode build setting so `actool` emits a coherent partial `Info.plist` covering both `CFBundlePrimaryIcon` and `CFBundleAlternateIcons`. The user's `Info.plist` is left untouched; injecting `CFBundleIcons` manually would be dropped during the actool merge and is therefore avoided.
- The `CodenameOne_GLAppDelegate` is patched to call `-[UIApplication setAlternateIconName:completionHandler:]` at launch. The delegate reads `[NSLocale preferredLanguages]`, tries the full `<lang>_<COUNTRY>` key first, then falls back to the language-only key, and clears the alternate icon (reverting to the default) if no variant matches.
- The injection is idempotent and runs before the `ios.afterFinishLaunching` hook, so any custom code you supply via that build hint is unaffected.



IOS displays a system alert the first time an app switches to an alternate icon. This is platform-standard behavior—Codename One can't suppress it.

Tips and troubleshooting

- File names are matched case-insensitively. `cn1_icon_EN_uk.png` is treated the same as `cn1_icon_en_UK.png`.
- Names that don't fit the `<2-letter lang>[_<2-letter country>]` pattern are logged and skipped; the rest of the build continues.
- The original `cn1_icon_*.png` files are removed after processing so they aren't shipped as stray bundle resources.
- To override the default (non-localized) icon, keep using the `Icon` setting in `codenameone_settings.properties` or the `icon` image in `CN1Resource.res`. This feature only adds **alternate** icons for specific locales.

50.5. Location - GPS

The Location [https://www.codenameone.com/javadoc/com/codename1/location/Location.html] API allows you to track changes in device location or the current user position.



The Simulator includes a Location Simulation tool that you can launch to determine the current position of the simulator and debug location events

The most basic usage for the API allows you to fetch a device Location, notice that this API is blocking and can take a while to return:

```
Location position = LocationManager.getLocationManager().getCurrentLocationSync();
```



In order for location to work on iOS you **MUST** define the build hint `ios.locationUsageDescription` and describe why your application needs access to location. Otherwise you won't get location updates!

The `getCurrentLocationSync()` method is good for cases where you need to fetch a current location once and not repeatedly query location. It activates the GPS then turns it off to avoid excessive battery usage. For example, if an application needs to track motion or position over time it should use the location listener API to track location as such:



Notice that there is a method called `getCurrentLocation()` which will return the current state and might not be right for some cases:

```
class MyListener implements LocationListener {
    public void locationUpdated(Location location) {
        // update UI etc.
    }

    public void providerStateChanged(int newState) {
        // handle status changes/errors appropriately
    }
}
LocationManager.getLocationManager().setLocationListener(new MyListener());
```



On Android location maps to low-level APIs if you disable the usage of Google Play Services. By default location should perform well if you leave the Google Play Services on

50.5.1. Location in the background - geofencing

Polling location is expensive and requires a special permission on iOS. Its also implemented rather differently both in iOS and Android. Both platforms place restrictions on the location API usage in the background.

Because of the nature of background location the API is non-trivial. It starts with the venerable `LocationManager` but instead of using the standard API you need to use `setBackgroundLocationListener`.

Instead of passing a `LocationListener` instance you need to pass a `Class` object instance. This is important because background location might be invoked when the app isn't running and an object would need to be allocated.

Notice that you **shouldn't** perform long operations in the background listener callback. IOS wake-up time is limited to about 10 seconds and the app could get killed if it exceeds that time slice.

Notice that the listener can also send events when the app is in the foreground, so it's recommended to check the app state before deciding how to process this event. You can use `Display.isMinimized()` to determine if the app is running or in the background.

When implementing this make sure that:

- The class passed to the API is a public class in the global scope. Not an inner class or anything like that!

- The class has a public no-argument constructor
- You need to pass it as a class literal for example: `MyClassName.class`. Don't use `Class.forName("my.package.MyClassName")`!
  Class names are problematic since device builds are obfuscated, you should use literals which the obfuscator detects and handles.

The following code demonstrates usage of the GeoFence API:

Android background location permissions (API 30+)

On Android 11 (API level 30) and higher, requesting background location permission requires a two-step process. First, foreground location permissions must be granted. Then, the app must request background location access, which will direct the user to the system settings to select `Allow all the time`. Codename One handles this flow automatically when you use `LocationManager`.

For Android 11+ (API 30+), Codename One detects if background location is needed and presents a dialog explaining the need before redirecting the user to the app settings. You can customize the permission prompt message using the localization key `android.permission.ACCESS_BACKGROUND_LOCATION`:

50.6. Background music playback

Codename One supports playing music in the background (for example: when the app is minimized) which is useful for developers building a music player style application.

This support isn't totally portable since the Android and iOS approaches for background music playback differ a great deal. To get this to work on Android you need to use the API: `MediaManager.createBackgroundMedia()`.

You should use that API when you want to create a media stream that will work even when your app is minimized.

For iOS you will need to use a special build hint: `ios.background_modes=music`.

Which should allow background playback of music on iOS and would work with the `createBackgroundMedia()` method.

50.7. Capture - photos, video, audio

The capture API allows you to use the camera to capture photographs or the microphone to capture audio. It even includes an API for video capture.
The API itself couldn't be simpler:

```
String filePath = Capture.capturePhoto();
```

Just captures and returns a path to a photo you can either open it using the Image [https://www.codenameone.com/javadoc/com/codename1/ui/Image.html] class or save it somewhere.



The returned file is a temporary file, you shouldn't store a reference to it and instead copy it locally or work with the `Image` object

For example: you can copy the `Image` to `Storage` using:



When running on the simulator the `Capture` API opens a file chooser API instead of physically capturing the data. This makes debugging device or situation specific issues simpler

You can capture an image from the camera using an API like this:

```
Form hi = new Form("Capture", new BorderLayout());
hi.setToolbar(new Toolbar());
Style s = UIManager.getInstance().getComponentStyle("Title");
FontImage icon = FontImage.createMaterial(FontImage.MATERIAL_CAMERA, s);

ImageViewer iv = new ImageViewer(icon);

hi.getToolbar().addCommandToRightBar("", icon, (ev) -> {
    String filePath = Capture.capturePhoto();
    if(filePath != null) {
        try {
            DefaultListModel<Image> m = (DefaultListModel<Image>)iv.getImageList();
            Image img = Image.createImage(filePath);
            if(m == null) {
                m = new DefaultListModel<>(img);
                iv.setImageList(m);
                iv.setImage(img);
            } else {
                m.addItem(img);
            }
            m.setSelectedIndex(m.getSize() - 1);
        } catch(IOException err) {
            Log.e(err);
        }
    }
});

hi.add(BorderLayout.CENTER, iv);
hi.show();
```



Figure 285. Captured photos previewed in the ImageViewer

You show video capture in the MediaManager section [https://www.codenameone.com/manual/components.html#mediamanager-section].

The sample below captures audio recordings (using the 'Capture' API) and copies them locally under unique names. It also demonstrates the storage and organization of captured audio:

```
Form hi = new Form("Capture", BoxLayout.y());
hi.setToolbar(new Toolbar());
Style s = UIManager.getInstance().getComponentStyle("Title");
FontImage icon = FontImage.createMaterial(FontImage.MATERIAL_MIC, s);

FileSystemStorage fs = FileSystemStorage.getInstance();
String recordingsDir = fs.getAppHomePath() + "recordings/";
fs.mkdir(recordingsDir);
try {
    for(String file : fs.listFiles(recordingsDir)) {
        MultiButton mb = new MultiButton(file.substring(file.lastIndexOf("/") + 1));
        mb.addActionListener((e) -> {
            try {
                Media m = MediaManager.createMedia(recordingsDir + file, false);
                m.play();
            } catch(IOException err) {
                Log.e(err);
            }
        });
        hi.add(mb);
    }

    hi.getToolbar().addCommandToRightBar("", icon, (ev) -> {
        try {
            String file = Capture.captureAudio();
            if(file != null) {
                SimpleDateFormat sd = new SimpleDateFormat("yyyy-MMM-dd-kk-mm");
                String fileName =sd.format(new Date());
                String filePath = recordingsDir + fileName;
                Util.copy(fs.openInputStream(file), fs.openOutputStream(filePath));
                MultiButton mb = new MultiButton(fileName);
                mb.addActionListener((e) -> {
```

```
                    try {
                        Media m = MediaManager.createMedia(filePath, false);
                        m.play();
                    } catch(IOException err) {
                        Log.e(err);
                    }
                });
                hi.add(mb);
                hi.revalidate();
            }
        } catch(IOException err) {
            Log.e(err);
        }
    });
} catch(IOException err) {
    Log.e(err);
}
hi.show();
```

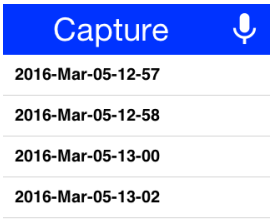
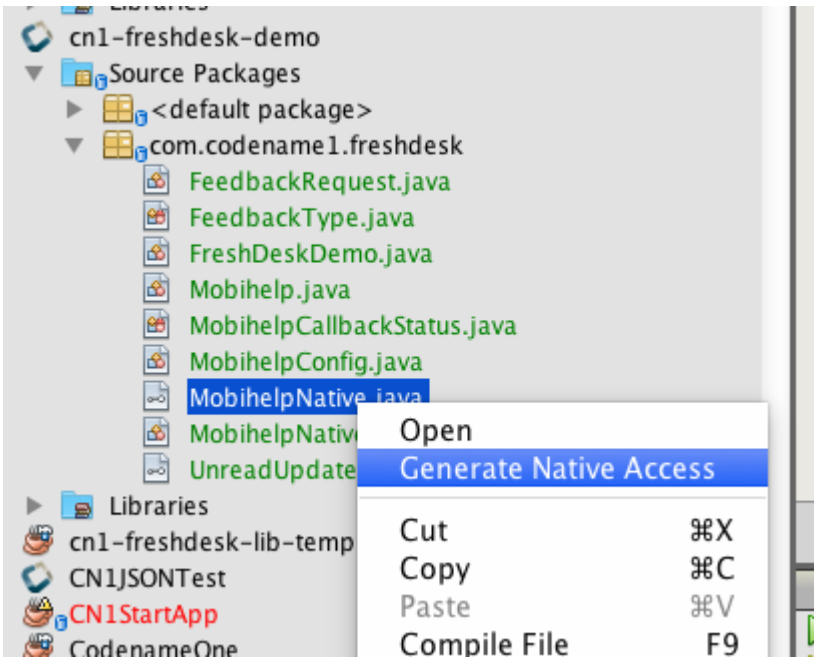


Figure 286. Captured recordings in the demo

Or, you can use the `Media`, `MediaManager` and `MediaRecorderBuilder` APIs to capture audio, as a more customizable approach than using the Capture API:



50.7.1. Capture asynchronous API

The `Capture` API also includes a callback based API that uses the `ActionListener` interface to implement capture. For example: you can adapt the previous sample to use this API as such:

50.8. Gallery

The gallery API allows picking an image and/or video from the cameras gallery (camera roll).



Like the `Capture` API the image returned is a temporary image that should be copied locally, this is due to device restrictions that don't allow direct modifications of the gallery

You can adapt the `Capture` sample above to use the gallery as such:

```
Form hi = new Form("Capture", new BorderLayout());
hi.setToolbar(new Toolbar());
Style s = UIManager.getInstance().getComponentStyle("Title");
FontImage icon = FontImage.createMaterial(FontImage.MATERIAL_CAMERA, s);

ImageViewer iv = new ImageViewer(icon);

hi.getToolbar().addCommandToRightBar("", icon, (ev) -> {
    Display.getInstance().openGallery((e) -> {
        if(e != null && e.getSource() != null) {
            try {
                DefaultListModel<Image> m = (DefaultListModel<Image>)iv.getImageList();
                Image img = Image.createImage((String)e.getSource());
                if(m == null) {
                    m = new DefaultListModel<>(img);
                    iv.setImageList(m);
                    iv.setImage(img);
                } else {
                    m.addItem(img);
                }
                m.setSelectedIndex(m.getSize() - 1);
            } catch(IOException err) {
                Log.e(err);
            }
        }
    }, Display.GALLERY_IMAGE);
});

hi.add(BorderLayout.CENTER, iv);
```



No need for a screenshot as it will look identical to the capture image screenshot above

The last value is the type of content picked which can be one of: `Display.GALLERY_ALL`, `Display.GALLERY_VIDEO` or `Display.GALLERY_IMAGE`.

50.9. Analytics integration

Analytics now has its own chapter. It covers the provider SPI, consent and privacy handling, the built-in providers (the Codename One first-party service, Google Analytics 4, Matomo, Firebase and a logging provider), and how to write your own. See the Analytics chapter.

The `AnalyticsService` class documented here previously is deprecated but still works — it delegates to the new API. The migration table at the end of the Analytics chapter maps each old call to its replacement.

50.10. Social sign-in (Facebook, Google, …)

The legacy Facebook and Google OAuth2 sign-in flows that used to live here drove an embedded `WebBrowser`, which the providers no longer permit. The full sign-in stack — system-browser-based OpenID Connect for any provider, plus Facebook / Google / Microsoft / Apple / Auth0 / Firebase helpers — now lives in the Authentication and Identity chapter.

50.11. Lead Component

Codename One has two basic ways to create new components:

1. Subclass a `Component` override `paint`, implement event callbacks etc.
2. Compose many components into a new component, by subclassing a `Container`.

Components such as Tabs [https://www.codenameone.com/javadoc/com/codename1/ui/Tabs.html] subclass `Container` which make a lot of sense for that component since it's physically a `Container`.

For example, components like MultiButton [https://www.codenameone.com/javadoc/com/codename1/components/MultiButton.html], SpanButton [https://www.codenameone.com/javadoc/com/codename1/components/SpanButton.html] & SpanLabel [https://www.codenameone.com/javadoc/com/codename1/components/SpanLabel.html] don't necessarily seem like the right candidate for compositing but they're all `Container` subclasses.

Using a `Container` provides you a lot of flexibility in layout & functionality for a specific component. `MultiButton` is a great example of that. It's a `Container` internally that's composed of 5 labels and a `Button`.

Codename One makes the `MultiButton` "feel" like a single button through the use of `setLeadComponent(Component)` which turns the button into the "leader" of the component.

When a `Container` hierarchy is placed under a leader all events within the hierarchy are sent to the leader, so if a label within the lead component receives a pointer pressed event this event will be sent to the leader.

For example: for the `MultiButton` the internal button will receive that event and send the action

performed event, change the state etc.

This creates some potential issues for instance in `MultiButton`:

```
myMultiButton.addActionListener((e) -> {
    if(e.getComponent() == myMultiButton) {
        // this won't occur since the source component is really a button!
    }
    if(e.getActualComponent() == myMultiButton) {
        // this will happennull
    }
});
```

The leader also determines the style state, so all the elements being lead are in the same state. For example: if the button is pressed all elements will display their pressed states, notice that they will do so with their own styles but they will each pick the pressed version of that style so a `Label` UIID within a lead component in the pressed state would return the Pressed state for a `Label` not for the `Button`.

This is convenient when you need to construct more elaborate UI's and the cool thing about it's that you can do this entirely in the designer which allows assembling containers and defining the lead component inside the hierarchy.

For example: the `SpanButton` class is like this code:

```
public class SpanButton extends Container {
    private Button actualButton;
    private TextArea text;

    public SpanButton(String txt) {
        setUIID("Button");
        setLayout(new BorderLayout());
        text = new TextArea(getUIManager().localize(txt, txt));
        text.setUIID("Button");
        text.setEditable(false);
        text.setFocusable(false);
        actualButton = new Button();
        addComponent(BorderLayout.WEST, actualButton);
        addComponent(BorderLayout.CENTER, text);
        setLeadComponent(actualButton);
    }


    public void setText(String t) {
        text.setText(getUIManager().localize(t, t));
    }

    public void setIcon(Image i) {
        actualButton.setIcon(i);
```

```
    }

    public String getText() {
        return text.getText();
    }

    public Image getIcon() {
        return actualButton.getIcon();
    }

    public void addActionListener(ActionListener l) {
        actualButton.addActionListener(l);
    }

    public void removeActionListener(ActionListener l) {
        actualButton.removeActionListener(l);
    }

}
```

50.11.1. Blocking lead behavior

The `Component` class has two methods that allow you to exclude a component from lead behavior: `setBlockLead(boolean)` & `isBlockLead()`.

Effectively when you have a `Component` within the lead hierarchy that you would like to treat differently from the rest you can use this method to exclude it from the lead component behavior while keeping the rest in line...

This should have no effect if the component isn't a part of a lead component.

The sample below is based on the `Accordion` component which uses a lead component internally:

```
f.getToolbar().addMaterialCommandToRightBar("", FontImage.MATERIAL_ADD, e -> addEntry(accr));
addEntry(accr);
f.add(BorderLayout.CENTER, accr);
f.show();
```

This allows you to add/edit entries but it also allows the delete button above to actually work. Without a call to `setBlockLead(true)` the delete button would cat as the rest of the accordion title.

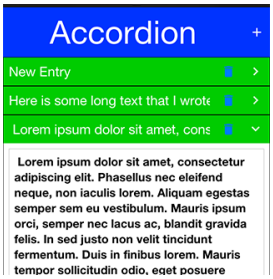


Figure 287. Accordion with delete button entries that work despite the surrounding lead

50.12. Pull to refresh

Pull to refresh is the common UI paradigm that Twitter popularized where the user can pull down the form/container to receive an update. Adding this to Codename One couldn't be simpler!

Just invoke `addPullToRefresh(Runnable)` on a scrollable container (or form) and the runnable method will be invoked when the refresh operation occurs.



Pull to refresh is implicitly implements in the `InifiniteContainer`

```
Form hi = new Form("Pull To Refresh", BoxLayout.y());
hi.getContentPane().addPullToRefresh(() -> {
    hi.add("Pulled at " + L10NManager.getInstance().formatDateTimeShort(new Date()));
});
hi.show();
```

`setPullToRefresh(Runnable)` is provided as a more discoverable alias for the same single-task slot — both names route to the same field, and a second call replaces whatever runnable was previously registered.

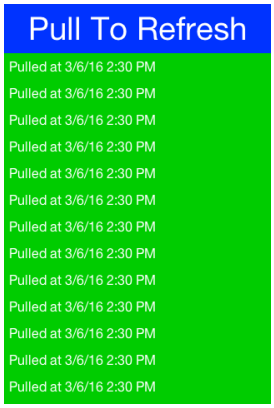


Figure 288. Pull to refresh demo

50.12.1. Modern arc-spinner pull-to-refresh

The legacy pull-to-refresh visual is the classic "rotating arrow + text" stack. Modern Material 3 / iOS apps instead show a thin circular arc that grows as the user pulls and spins once the task fires. This is shipped as an opt-in path via the `pullToRefreshModernBool` theme constant; the iOS Modern and Android Material themes enable it by default so apps shipping against those themes get the spec-accurate look without any extra wiring.

To enable in a custom theme:

```
#Constants {
    pullToRefreshModernBool: true;
    pullToRefreshIndicatorDiameterMm: 8;       /* 8mm circle           */
    pullToRefreshIndicatorStrokeMm: "0.6";     /* 0.6mm stroke thickness */
}
```

```
TabIndicator {
    /* The pull-to-refresh arc shares its color with the animated tab
       indicator -- both follow the brand's accent. */
    color: #007aff;
    background-color: transparent;
    padding: 0;
    margin: 0;
}
```

Behavior:

- During the pull gesture the arc sweep grows from 0° to ~330° proportional to how far the user has pulled past the threshold.
- When the refresh task fires the arc rotates continuously at ~360°/sec until the task completes.
- All sizing is in millimetres so the indicator stays physical-size-accurate across device densities; no rasterized images involved.

The legacy `addPullToRefresh` API is unchanged — the same `Runnable` is invoked at the same moment; only the visual rendering differs.

50.13. Running 3rd party apps using display's execute

The Display [https://www.codenameone.com/javadoc/com/codename1/ui/Display.html] class's `execute` method allows you to invoke a URL which is bound to a particular application.

This works rather well assuming the application is installed. For example: this list [http://wiki.akosma.com/IPhone_URL_Schemes] contains a set of valid URLs that can be used on iOS to run common applications and use built-in functionality.

Some URLs might not be supported if an app isn't installed, on Android there isn't much that can be done but iOS has a `canOpenURL` method for Objective-C.

On iOS you can use the `Display.canExecute()` method which returns a `Boolean` instead of a `boolean` which allows you to support 3 result states:

1. `Boolean.TRUE` - the URL can be executed
2. `Boolean.FALSE` - the URL isn't supported or the app is missing
3. `null` - you've no idea whether the URL will work on this platform.

The sample below launches a "godfather" search on IMDB when this is sure to work (on iOS currently). You can actually try to search for null as well but this sample plays it safe by using the http link which is sure to work:

```
Boolean can = Display.getInstance().canExecute("imdb:///find?q=godfather");
if(can != null && can) {
    Display.getInstance().execute("imdb:///find?q=godfather");
} else {
```

```
    Display.getInstance().execute("http://www.imdb.com");
}
```

50.14. Automatic build hint configuration

You try to make Codename One "seamless," this expresses itself in small details such as the automatic detection of permissions on Android etc. The build servers go a long way in setting up the environment as intuitive. However, it isn't enough, build hints are often confusing and obscure. It's hard to abstract the mess that's native mobile OSes and the odd policies from Apple/Google...

A good example for a common problem developers face is location code that doesn't work in iOS. This is due to the `ios.locationUsageDescription` build hint that's required. The reason that build hint was added is a need by Apple to provide a description for every app that uses the location service.

To solve this sort of used case you have two APIs in `Display`:

Both of these allow you to detect if a build hint is set and if not (or if it's set incorrectly) set its value...

If you use the location API from the simulator and you didn't define `ios.locationUsageDescription` Codename One will implicitly define a string there. The cool thing is that you will now see that string in your settings and you would be able to customize it.

For example, this gets way better than that trivial example!

The real value is for 3rd party cn1lib authors. For example: Google Maps or Parse. They can inspect the build hints in the simulator and show an error in case of a misconfiguration. They can even show a setup UI. Demos that need special keys in place can force the developer to set them up before continuing.

50.15. Easy thread

Working with threads is ranked as one of the least intuitive and painful tasks in programming. This is such an error prone task that some platforms/languages took the route of avoiding threads entirely. A typical scenario: you need to convert some code to work on a separate thread but you still want the ability to communicate and transfer data from that thread.

This is possible in Java but non-trivial, the thing is that this is easy to do in Codename One with tools such as `callSerially` that let arbitrary code run on the EDT. Why not offer that to any random thread?

That's why `EasyThread` exists: it takes some concepts of Codename One's threading and makes them more accessible to an arbitrary thread. This way you can move things like resource loading into a separate thread and synchronize the data back into the EDT as needed...

Easy thread can be created like this:

You can send a task to the thread using:

However, it gets better, say you want to return a value:

Lets break that down... You ran the thread with the success callback on the new thread then the callback got invoked on the EDT as a result. This code `(success) → success.onSuccess(doThisOnTheThread())` ran off the EDT in the thread and when you invoked the `onSuccess` callback it sent it asynchronously to the EDT here: `(myResult) → onEDTGotResult(myRsult)`.

These asynchronous calls make things a bit painful to wade through, so the API wraps them in a simplified synchronous version:

There are a few other variants like `runAndWait` and there is a `kill()` method which stops a thread and releases its resources.

50.16. Mouse cursor

Codename one can change the mouse cursor when hovering over specific areas to show resizability, movability etc. For obvious reasons this feature is available in the desktop and JavaScript ports as the other ports rely on touch interaction. The feature is off by default and needs to be enabled on a `Form` by using `Form.setEnableCursors(true);`. If you're writing a custom component that can use cursors such as `SplitPane` you can use:

```
@Override
protected void initComponent() {
    super.initComponent();
    getComponentForm().setEnableCursors(true);
}
```

Once this is enabled you can set the cursor over a specific region using `cmp.setCursor()` which accepts one of the cursor constants defined in `Component`.

50.17. Working with GIT

Working with GIT for storing Codename One projects isn't a feature but since it's so ubiquitous you think it's important to have a common guideline.

When you first started committing to git you used something like this for NetBeans projects:

```
*.jar
nbproject/private/
build/
dist/
lib/CodenameOne_SRC.zip
```

Removing the jars, build, private folder etc. makes a lot of sense but there are a few nuances that are missing here...

50.17.1. cn1lib's

You will notice you excluded the jars which are stored under lib and you exclude the Codename One source zip. However, cn1libs aren't excluded... That was an omission since the original project you committed didn't have cn1libs. However, should you commit a binary file to git?

It's a fair question. Git isn't good with binaries but committing cn1libs can still make sense. In another project that did have a cn1lib the approach was this:

```
*.jar
nbproject/private/
build/
dist/
lib/CodenameOne_SRC.zip
lib/impl/
native/internal_tmp/
```

The important lines are `lib/impl/` and `native/internal_tmp/`. Technically cn1libs are zips. When you run a refresh-libs operation they unzip into the right directories under `lib/impl` and `native/internal_tmp`. By excluding these directories you can remove duplicates that can result in conflicts.

50.17.2. Resource files

Committing the res file is a matter of personal choice. It's committed in the git ignore files above but you can remove it. The res file is at risk of corruption and in that case having a history you can refer to, matters a lot.

However, the resource file is a bit of a problematic file. As a binary file if you have a team working with it the conflicts can be a major blocker. This was far worse with the old GUI builder, that was one of the big motivations of moving into the new GUI builder which works better for teams.

Still, if you want to keep an eye of every change in the resource file you can switch on the File → XML Team Mode which should be on by default. This mode creates a file hierarchy under the `res` directory to match the res file you opened. For example: if you have a file named `src/theme.res` it will create a matching `res/theme.xml` and also nest all the images and resources you use in the res directory.

That's useful as you can edit the files directly and keep track of every file in git. For example, this has two big drawbacks:

- It's flaky - while this mode works it never reached the stability of the regular res file mode
- It conflicts - the simulator/device are oblivious to this mode. If you fetch an update you also need to update the res file and you might still have conflicts related to that file

Both of these issues shouldn't be a deal-breaker. Even though this mode is a bit flaky it's better than the alternative as you can "see" the content of the resource file. You can revert and reapply your changes to the res file when merging from git, it's tedious but again not a deal-breaker.

50.17.3. Eclipse version

Building on the gitignore you've for NetBeans the eclipse version should look like this:

```
.DS_Store
*.jar
build/
dist/
lib/impl/
native/internal_tmp/
.metadata
bin/
tmp/
*.tmp
*.bak
*.swp
*.zip
*~.nib
local.properties
.settings/
.loadpath
.recommenders
.externalToolBuilders/
*.launch
*.pydevproject
.cproject
.factorypath
.buildpath
.project
.classpath
```

50.17.4. IntelliJ/IDEA

```
.DS_Store
*.jar
build/
dist/
lib/impl/
native/internal_tmp/
*.zip
.idea/**/workspace.xml
.idea/**/tasks.xml
.idea/dictionaries
.idea/**/dataSources/
.idea/**/dataSources.ids
.idea/**/dataSources.xml
.idea/**/dataSources.local.xml
.idea/**/sqlDataSources.xml
.idea/**/dynamic.xml
```

```
.idea/**/uiDesigner.xml
.idea/**/gradle.xml
.idea/**/libraries
*.iws
/out/
atlassian-ide-plugin.xml
```

Chapter 51. Video encoding & decoding (VideoIO)

The VideoIO [https://www.codenameone.com/javadoc/com/codename1/media/VideoIO.html] API is to video what the ImageIO [https://www.codenameone.com/javadoc/com/codename1/ui/util/ImageIO.html] API is to still images, but it goes deeper. It uses the platform's **native** codecs to:

1. **Encode** application rendered frames and audio into a standard video file, with deep control over dimensions, frame rate, codec, bitrate, key frame interval and the audio track.
2. **Decode** an existing clip into **frame accurate** RGBA frames and PCM audio.



The decode side is something the Media [https://www.codenameone.com/javadoc/com/codename1/media/Media.html] player API doesn't provide. `Media` is a player: its `setTime()` snaps to key frames and gives no exact frame guarantee. `VideoReader`, by contrast, decodes the precise frame at a requested time, which is what you need when you want to read an imported clip (for example a screen recording) frame by frame.

`VideoIO` is supported on iOS, macOS, Android, Windows, Linux, JavaScript and the desktop simulator. It's **not** available on the TV, Watch or Car targets.

51.1. Checking for support

As with other optional platform features, always gate usage with `isSupported()`:

```
if (!VideoIO.isSupported()) {
    // Video encoding / decoding is not available on this platform
    // (for example TV, Watch or Car), so fall back gracefully.
    return;
}
VideoIO io = VideoIO.getVideoIO();
```



On the desktop simulator `VideoIO` is backed by an `ffmpeg`/`ffprobe` binary. The simulator bundles one automatically; if `isSupported()` returns `false` on the desktop, make sure `ffmpeg` is on your `PATH` or set the `ffmpeg.dir` system property.

51.2. Encoding a video

Build a `VideoWriter` with a VideoWriterBuilder [https://www.codenameone.com/javadoc/com/codename1/media/VideoWriterBuilder.html], then push frames and (optionally) audio. Because every frame is an ordinary Image [https://www.codenameone.com/javadoc/com/codename1/ui/Image.html] that you draw into, you have complete control over the pixels - render charts, overlays, transformed camera frames, generated animation, anything.

```
String out = FileSystemStorage.getInstance().getAppHomePath() + "/generated.mp4";
int w = 640, h = 480;
float fps = 30;

VideoWriter writer = new VideoWriterBuilder()
        .path(out)
        .width(w).height(h).frameRate(fps)
        .videoCodec(VideoIO.CODEC_H264).videoBitRate(4_000_000)
        .build();

// Each frame is just an Image you fully control: draw whatever you like.
for (int i = 0; i < 90; i++) {                  // 3 seconds at 30fps
    Image frame = Image.createImage(w, h, 0xff000000);
    Graphics g = frame.getGraphics();
    g.setColor(0xffffff);
    g.fillRect((i * 8) % w, h / 2 - 20, 60, 40);
    writer.writeFrame(frame, Math.round(i * 1000f / fps));
}
writer.close();
```

To add an audio track, enable it on the builder (`.hasAudio(true)`) and push interleaved 16 bit PCM with `VideoWriter.writeAudio(…)`, or hand it an AudioBuffer [https://www.codenameone.com/javadoc/com/codename1/media/AudioBuffer.html].

51.3. Decoding a video frame by frame

`VideoReader` exposes the clip's duration, dimensions and frame rate, a frame accurate `frameAt(long)` and a variable-to-constant frame rate resampler `readFrames(fps, callback)` that walks the whole clip emitting evenly spaced frames:

```
VideoReader reader = VideoIO.getVideoIO().openReader(videoPath);
System.out.println(reader.getWidth() + "x" + reader.getHeight()
        + " " + reader.getFrameRate() + "fps, "
        + reader.getDurationMillis() + "ms");

List<Image> thumbnails = new ArrayList<>();

// Frame accurate single frame (unlike Media.setTime which snaps to key frames):
VideoFrame oneSecond = reader.frameAt(1000);
if (oneSecond != null) {
    thumbnails.add(oneSecond.toImage());
}

// Resample the (possibly variable frame rate) clip to a constant 10fps stream:
reader.readFrames(10, f -> {
    // f.getARGB() / f.toImage() give you the decoded RGBA pixels
    thumbnails.add(f.toImage());
    return thumbnails.size() < 50;   // stop after 50 frames
```

```
});
reader.close();
```

The audio track, when present, is decoded to interleaved PCM:

```
VideoReader reader = VideoIO.getVideoIO().openReader(videoPath);
if (reader.hasAudio()) {
    AudioBuffer pcm = reader.readAudio();
    System.out.println("Decoded " + pcm.getSize() + " PCM samples at "
            + reader.getAudioSampleRate() + "Hz, "
            + reader.getAudioChannels() + " channels");
}
reader.close();
```

51.4. Discovering available codecs

The set of codecs is device dependent. Enumerate it to pick the best one (and to detect hardware accelerated encoders):

```
VideoIO io = VideoIO.getVideoIO();
for (VideoCodec codec : io.getAvailableEncoders()) {
    System.out.println(codec.getId()
            + " (" + codec.getName() + ")"
            + (codec.isHardwareAccelerated() ? " [hardware]" : ""));
}
boolean canEncodeH264 = io.isEncoderSupported(VideoIO.CODEC_H264);
```

51.5. Platform support

Platform	Backend
iOS / macOS	AVFoundation (`AVAssetReader` / `AVAssetWriter`, hardware H.264/HEVC)
Android	`MediaCodec` + `MediaMuxer` / `MediaExtractor` + `MediaMetadataRetriever`
JavaScript	HTML5 `<video>` + `<canvas>` decode + WebCodecs encode
Windows	Media Foundation (`IMFSourceReader` / `IMFSinkWriter`)
Linux	GStreamer (`appsrc` / `appsink`)
Simulator	`ffmpeg` / `ffprobe`
TV / Watch / Car	Not supported (`isSupported()` returns `false`)

Chapter 52. Performance, size & debugging

52.1. Reducing resource file size

It's easy to lose track of size/performance when you're working within the comforts of a visual tool like the Codename One Designer. When optimizing resource files you need to keep in mind one thing: it's all about image sizes.



Images will take up 95-99% of the resource file size; everything else pales in comparison.

Like every optimization the first rule is to reduce the size of the biggest images which will provide your biggest improvements, for this purpose Codename One provides the ability to see image sizes in kilobytes. To launch that feature use the menu item Images → Image Sizes (KB) in the designer.

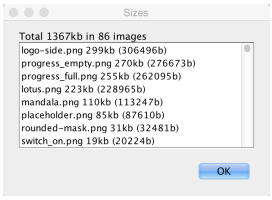


Figure 289. Image sizes window that allows you to find the biggest impact on your RAM/Storage

This produces a list of images sorted by size with their sizes. Often the top entries will be multi-images, which include HD resolution values that can be pretty large. These high-resolution images take up a significant amount of space!

Just going to the multi-images, selecting the unnecessary resolutions & deleting these images can save significant amounts of space:

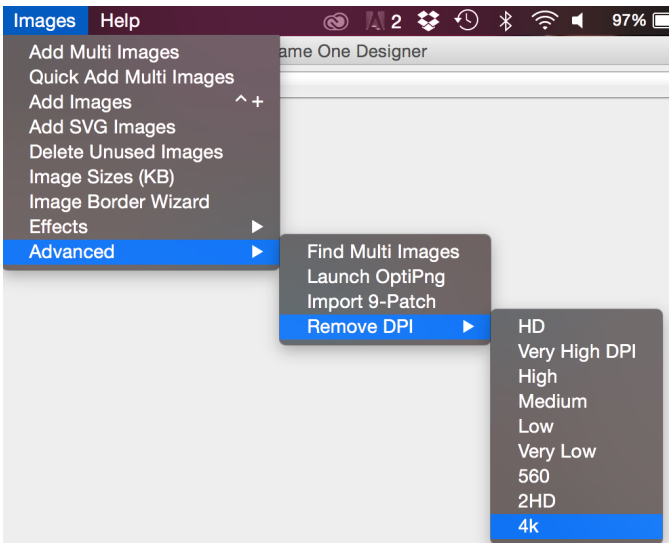


Figure 290. Removing unused DPI's



You can see the size in KB at the top right side in the designers image viewer

Applications using the old GUI builder can use the Images → Delete Unused Images menu option (it's also under the Images menu). This tool allows detecting and deleting images that aren't used

within the theme/GUI.

If you have a large image that's opaque you might want to consider converting it to JPEG and replacing the built in PNG's. Notice that JPEGs work on all supported devices and are typically smaller.

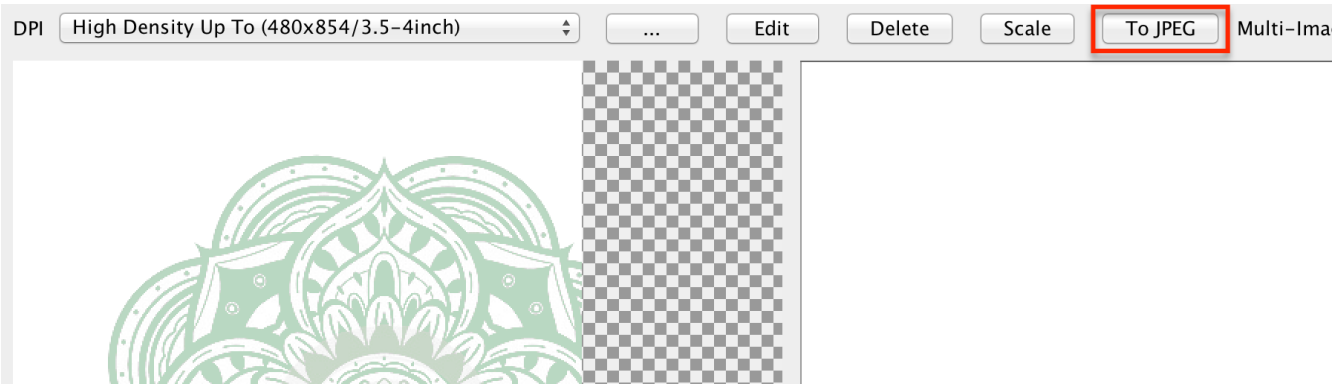


Figure 291. Convert a MultiImage to use JPEGs instead of PNGs

You can use the excellent OptiPng [http://optipng.sourceforge.net/] tool to optimize image files right from the Codename One designer. To use this feature you need to install OptiPng then select Images → Launch OptiPng from the menu. Once you do that the tool will automatically optimize all your PNG's.

When faced with size issues make sure to check the size of your res file, if your JAR file is large open it with a tool such as 7-zip and sort elements by size. Start reviewing which element justifies the size overhead.

52.2. Improving performance

As a developer you can do quite a few things to improve the performance and memory footprint of a Codename One application. This sometimes depends on specific device behaviors but some tips here are true for all devices.

The simulator contains some tools to measure performance overhead of a specific component and also detect EDT blocking logic. Other than that follow these guidelines to create more performance code:

- **Avoid round rect borders** - they have a huge overhead on all platforms. Use image borders instead (counter intuitively they're MUCH faster)
- **Avoid Gradients** - they perform poorly on most OSes. Use a background image instead
- **Use larger images** when tiling or building image borders, using a 1 pixel (or event a few pixels) wide or high image and tiling it repeatedly can be expensive
- **Shrink resource file sizes** - Otherwise data might get collected by the garbage collector and reloading data might be expensive
- **Check that you don't have too many image lock misses** - this is discussed in the graphics section
- **On some platforms mutable images are slow** - mutable images are images you can draw on (using `getGraphics()`). On some platforms they perform quite badly (for example, iOS) and

should be avoided. You can check if mutable images are fast in a platform using `Display.areMutableImagesFast()`

- * Make components either transparent or opaque * - a translucent component must paint it's parent every time. This can be expensive. An opaque component might have margins that would require that you paint the parent so there is often overdraw in such cases (overdraw means the same pixel being painted twice).

52.2.1. ParparVM native translation performance hints

For ParparVM-generated native code, Codename One now supports method-level optimization hints via annotations. These can provide good wins in hot code paths, but they come with tradeoffs and should be applied surgically.

Method-level codegen hints

- `@DisableDebugInfo`
  Suppresses generated line/debug metadata for the annotated method. This can reduce generated C size and remove some per-instruction debug overhead.
- `@DisableNullChecksAndArrayBoundsChecks`
  Suppresses generated null and array-bounds checks for the annotated method. This can reduce branch-heavy code in tight loops.



Use these on methods that are both performance-critical and well-covered by tests. These annotations intentionally trade runtime safety diagnostics for speed.

Class-level concrete implementation hints

For native ParparVM output (C/Objective-C), you can also provide a class-level hint that a base type always maps to a known concrete subclass at runtime:

- `@Concrete(name="fully.qualified.ConcreteClassName")`
  Allows the translator to bypass virtual table lookup for `invokevirtual` calls on the annotated base class. The translator first attempts a direct call on the concrete class; if the method isn't implemented there, it falls back to the annotated base class implementation.

This is useful for platform abstraction classes where one implementation is guaranteed in the native pipeline.

For example, Codename One annotates:



This hint is intended for ParparVM native translation and doesn't apply to the JavaScript back end.

Fused object layout: `@Fused`

A class annotated `@com.codename1.annotations.Fused` tells the translator that primitive arrays created in its constructors are **fully encapsulated**: they're implementation details the class never hands out. The translator then lays the owner object and those arrays out in a **single allocation**

block — one allocation instead of several, one GC visit instead of several, and no pointer indirection between the object and its buffer.

```
@com.codename1.annotations.Fused
class RgbImage {
    private final int[] pixels;   // laid out INSIDE the RgbImage allocation
    private final byte[] flags;
    RgbImage(int w, int h) {
        pixels = new int[w * h];  // computed sizes (w * h) fuse too
        flags = new byte[16];
    }
}
```

The contract you accept by annotating:

- The fused arrays must never be **stored** into another object's field, a static, or an array — their lifetime is the owner's lifetime. Reading them, iterating them, and passing them as transient call arguments is fine.
- Keep the fields private and never return them from a getter. If the class hands its buffer out, don't annotate it.

Everything else stays automatic and safe: constructors that compute non-constant sizes, reflective instantiation, delegating constructors and oversized arrays (bigger than the small-object allocator handles) all fall back to ordinary separate allocations with identical semantics.

`java.lang.String` and `java.lang.StringBuilder` are fused this way internally — a typical string is one allocation, not two.

Optimizations you get without annotations

Several ParparVM optimizations are applied automatically when the translator can **prove** they're safe; knowing they exist helps you write code that benefits from them:

- **Stack-allocated string building.** Every string concatenation the compiler lowers to a `StringBuilder` chain — and any builder the translator can prove never escapes its method — is allocated on the C stack, including its initial character buffer. A typical `"a" + x + "b"` allocates **nothing** on the heap except the resulting `String`. This is proven per call site by escape analysis; passing the builder to another method or storing it in a field simply keeps that site on the ordinary heap path.
- **Tagged integers.** On 64-bit targets, `Integer.valueOf` returns a tagged immediate pointer instead of allocating — boxing an `int` for a map key is free. 32-bit-pointer targets (including Apple Watch) fall back to heap boxing automatically.
- **Closed-world devirtualization.** A virtual call with no reachable override compiles to a direct C call, which link-time optimization can then inline. `final` classes and methods make this trivially provable.
- **Compact collections.** `HashMap`/`LinkedHashMap` use open addressing over parallel arrays — no entry objects, no allocation on `put`, `clear()` is three array wipes.

- **Bounds-check elimination.** The canonical `for (int i = 0; i < arr.length; i++)` loop shape is proven safe and compiled without per-access checks — prefer it over manual index juggling.

The benchmark and correctness suite that gates these optimizations lives in `vm/benchmarks` in the repository, with run instructions in its README; every optimization must produce output bit-identical to a host JVM there before it ships.

Fast method-stack path

The translator can emit a fast method-stack prologue/epilogue (`DEFINE_METHOD_STACK_FAST_*` and `CN1_FAST_RETURN_RELEASE`) for methods that meet strict safety criteria. Methods that qualify for the stricter **frameless** form (no try/catch, no synchronization) skip the VM frame entirely — their locals become plain C locals the optimizer keeps in registers.

In practice, this tends to help for:

- Small, hot methods.
- Methods without monitor usage / exception-heavy flow.
- Methods with straightforward control flow and low instruction complexity.

Tradeoffs:

- Overly broad fast-path eligibility can regress performance if extra branches or memory writes are introduced.
- Primitive- fast-frame variants may not always outperform a straightforward full clear on all targets/compilers.



Benchmark representative workloads after enabling fast-stack behavior. Keep eligibility conservative and expand where measurement shows consistent gains.

Base64-style hot-loop guidelines

For low-level loops (for example, Base64 encode/decode):

- Prefer simple loop bodies with predictable branches.
- Cache decode/lookup tables in primitive arrays (`int[]` lookup tables can reduce per-iteration conversion overhead).
- Avoid adding “defensive” branches in the inner-most loop unless they’re required for correctness in production inputs.

Build configuration matters

When benchmarking translator output, ensure native projects are compiled with optimization enabled (for example, CMake `Release` builds). Debug/default builds can hide improvements or produce misleading regressions.

If you’re using the integration test harness, make sure CMake is configured with:

```
-DCMAKE_BUILD_TYPE=Release
```

Without this setting, comparison between Java and ParparVM native output is often noisy and can lead to wrong optimization conclusions.

52.3. SIMD: Data-Parallel primitives

SIMD stands for **Single Instruction, Multiple Data**. It's a family of CPU instructions that apply the same arithmetic or logical operation to several values at once, packed together into a single wide register. On a 128-bit NEON or SSE register you can hold sixteen bytes, eight 16-bit integers, four 32-bit integers, or four 32-bit floats, and a single instruction such as `VADD` or `PADDD` then operates on every lane in parallel. For data-parallel workloads—Base64 encode/decode, pixel blending, alpha-mask compositing, color-channel manipulation, table lookups—SIMD typically delivers a 3× to 10× speedup over scalar code.

Modern mobile and desktop CPUs expose several SIMD instruction sets:

- x86/x64—MMX, SSE, SSE2/3/4, AVX, AVX2, AVX-512
- ARM/ARM64—NEON (Advanced SIMD) and SVE

Codename One exposes those primitives as portable Java method calls on `com.codename1.util.Simd` [https://www.codenameone.com/javadoc/com/codename1/util/Simd.html]. On ParparVM (iOS) the translator lowers them to NEON intrinsics; on Android and the JavaSE simulator a pure-Java fallback in `JavaSESimd` provides the same semantics so code written against `Simd` runs everywhere and simply performs better where native SIMD is available.

52.3.1. When to use SIMD

SIMD is worth reaching for when **all** the following hold:

- You are processing a buffer of at least dozens of elements at a time.
- The operation per element is simple (add, min, blend, table lookup, interleave/unpack) and identical across elements.
- Control flow is regular—no per-element early exit, no data-dependent branches that differ across lanes.
- Data is contiguous in a primitive array (`byte[]`, `int[]`, `float[]`).

Classic fits are image pixel passes, codecs (Base64, UTF-8 validation, hex encoding), checksums, audio mixing, and vector arithmetic. Poor fits are tight loops with pointer chasing, per-element branching on complex state, or buffers smaller than a single SIMD register.

If you aren't sure whether your platform has a vectorized implementation available, call `Simd.get().isSupported()`. The fallback path still returns correct results; `isSupported()` only tells you whether you will see the speedup.

52.3.2. Getting an instance

`Simd` is accessed as a singleton:

The returned object is safe to cache in a field as long as the cache is refreshed across simulator restarts (the simulator may swap the backing implementation).

52.3.3. Allocation: Alignment and provenance

SIMD load/store instructions prefer—and on some architectures require— that the memory address they read from or write to be a multiple of the register width (16 bytes on NEON and SSE). Misaligned access either throws a hardware fault on strict platforms or costs extra cycles because the CPU has to split the transaction across two cache lines. A JVM offers no guarantee that `new byte[64]` is aligned to anything beyond the platform's object-header convention, so the framework forbids passing arbitrary `new`-allocated arrays to `Simd` primitives.

Every buffer used with a `Simd` primitive must so come from one of the allocation helpers on `Simd`:

All three methods enforce a minimum size of 16 elements and throw `IllegalArgumentException` for anything smaller; the minimum keeps each buffer large enough to hold a full SIMD register and leaves room for alignment padding.



Don't construct SIMD buffers with `new byte[64]` and pass them to `Simd`. The simulator enforces this at runtime (see *Simulator Tracking* below) and the iOS native path assumes alignment. Passing an unregistered array yields either an `IllegalArgumentException` in the simulator or undefined results on device.

52.3.4. A minimal End-to-End example

All `Simd` ops take explicit `offset` / `length` pairs so callers can chunk a large buffer through a small scratch array without reallocating. Many methods have interleaved variants—`unpackBytesInterleaved3/4`, `packBytesInterleaved3/4`—designed for packed pixel formats (RGB, RGBA) and fused operations like `blendByMaskTestNonzero` and `replaceTopByteFromUnsignedBytes` that collapse common three- or four-pass pixel pipelines into a single vector pass.

52.3.5. Scratch allocations: `alloca*`

A SIMD primitive often needs a short-lived working buffer: somewhere to stage intermediate lanes, hold a temporary mask, or accumulate partial results. Allocating a new `byte[]` per call via `allocByte` would dominate the runtime of the primitive it was trying to accelerate—heap allocation, zero-fill, and eventual GC reclamation all add up. The natural solution is to let the compiler place these buffers on the call stack where they cost nothing to create and are reclaimed automatically when the method returns. This is what the `alloca*` family exposes:

On the simulator (and on Android) these behave like heap allocations with registration. On ParparVM the translator intercepts each `alloca*` call and rewrites it into a C-level `__builtin_alloca` that carves a faux `JavaArrayPrototype` out of the current C stack frame. The resulting pointer masquerades as an ordinary Java array for the lifetime of the enclosing method and is reclaimed on return.

The distinction between `allocByte` and `allocaByte` mirrors the distinction between `new byte[N]` and C's `alloca(N)`:

	`alloc*` **(heap)**	`alloca*` **(stack on ParparVM)**
Cost to assign	object header + zero fill + GC bookkeeping	register adjust, essentially free
Lifetime	until last reference dropped	until enclosing method returns
Can be stored in a field	yes	**no—use-after-free on device**
Can be returned from the method	yes	**no**
Can be passed to non-`Simd` methods	yes	**no**
Initial contents	zero	undefined (use `*Zeroed` / `*Filled`)
Size limit	heap	bounded by remaining stack



`alloca*` memory, like any stack memory, starts out containing whatever the previous frame left behind. Callers that require predictable initial contents should use `allocaByteZeroed` / `allocaIntZeroed` / `allocaFloatZeroed`, or the `*Filled` variants that accept an explicit initial value, rather than rolling their own loop.

52.3.6. Rules for `alloca*` scratch arrays

Because `alloca*` memory can't outlive its defining method on ParparVM, the array it returns must be treated as method-local. The framework enforces four rules at build and run time:

1. Don't **return** an `alloca*` array from the method that allocated it.
2. Don't **store** an `alloca*` array in an instance or static **field**, or into an **object array**.
3. Don't pass an `alloca*` array to a method whose owner isn't `Simd`, `IOSSimd`, or `JavaSESimd`. Helper methods outside the SIMD package might themselves violate one of the other rules.
4. Don't pass an `alloca*` array through `invokedynamic`; dynamic dispatch can't be analyzed statically.

Inside the defining method you are free to pass the array to any number of `Simd` primitives, copy values into and out of it, combine it with other buffers, etc.

52.3.7. Build-Time verification

The Maven plugin ships a bytecode-compliance goal (`compliance-check`) that walks every method in the application's compiled classes and performs a dataflow pass whose only job is to track whether each value on the operand stack or in a local variable was produced by an `alloca*` call. The pass is an ASM `BasicInterpreter` subclass that taints every value returned from a method on `Simd`, `IOSSimd`, or `JavaSESimd` whose name begins with `alloca` followed by an upper-case letter. The taint propagates through `DUP`, `ASTORE`/`ALOAD`, `CHECKCAST`, and every control-flow merge, so a value keeps its provenance through any amount of local plumbing.

The verifier fails the build if a tainted value reaches any of:

- `ARETURN`—"SIMD alloca value returned from method"
- `PUTFIELD` / `PUTSTATIC`—"SIMD alloca value stored into instance/static field"
- `AASTORE`—"SIMD alloca value stored into object array"
- `INVOKE*` whose owner isn't `Simd`, `IOSSimd`, or `JavaSESimd`
- `INVOKEDYNAMIC`: refused unconditionally

Every violation is reported with the offending class, method, and the hint "Keep SIMD alloca scratch arrays method-local and only pass them to SIMD methods." Because this is wired into the Maven build, code that would dereference freed stack memory on device can't ship.

A complementary guard lives inside the translator itself: `CustomInvoke.appendSimdAllocaExpression` only performs stack lowering when the length argument is a **compile-time constant**. A call like `simd.allocaByte(n)` with a variable `n` falls through to the default codegen and becomes a normal heap allocation, which avoids both unbounded stack growth and an alloca whose size the translator can't inspect.

52.3.8. Simulator tracking of aligned arrays

The JavaSE simulator runs on the desktop where there is no NEON, no `__builtin_alloca`, and no guarantee that a given `byte[]` is 16-byte aligned. To keep developer bugs from hiding on the desktop and reappearing only on device, the simulator takes a strict stance: **only arrays obtained from `Simd.alloc` / `Simd.alloca*` are accepted as inputs to SIMD primitives.***

Internally this is an identity-based registry. Every allocation helper routes its result through a registration call that stores the array's `System.identityHashCode`—the JVM's per-object identity hash, stable for the object's lifetime regardless of GC movement. Every SIMD entry point validates its array arguments against the same set and throws `IllegalArgumentException` if a caller hands it a plain `new byte[64]`:

```
java.lang.IllegalArgumentException: SIMD array argument was not
allocated using Simd.alloc*(). objectId=…
```

Combined with the build-time verifier, both static analysis and dynamic execution refuse to accept arrays that would not be 16-byte aligned on device. On the device itself no such validation occurs—by that point the build has already proved the invariant.

52.3.9. Built-In SIMD paths

You gain from SIMD without touching the API directly. `com.codename1.util.Base64` detects `Simd.isSupported()` at encode/decode entry points and routes the hot loop through `unpackLookupBytesInterleaved4`, `lookupBytes`, `or`, and `shl` with cached per-call constant tables; the bodies of `Image.applyMask`, `Image.modifyAlpha`, and `Image.removeColor` use fused primitives like `replaceTopByteFromUnsignedBytes` and `blendByMaskTestNonzero` to collapse multi-pass pixel pipelines into a single vector loop.

If you are writing your own hot loop on a primitive buffer and the data is already contiguous and aligned in a `Simd.alloc*` array, reach for the `Simd` API before optimizing further with the ParparVM annotations described above—vectorization and annotation hints compose, but vectorization is the larger win.

52.4. Performance monitor

The Performance Monitor tool can be accessible via the Simulator → Performance Monitor menu option in the simulator. This launches the following UI that can help you improve application performance:

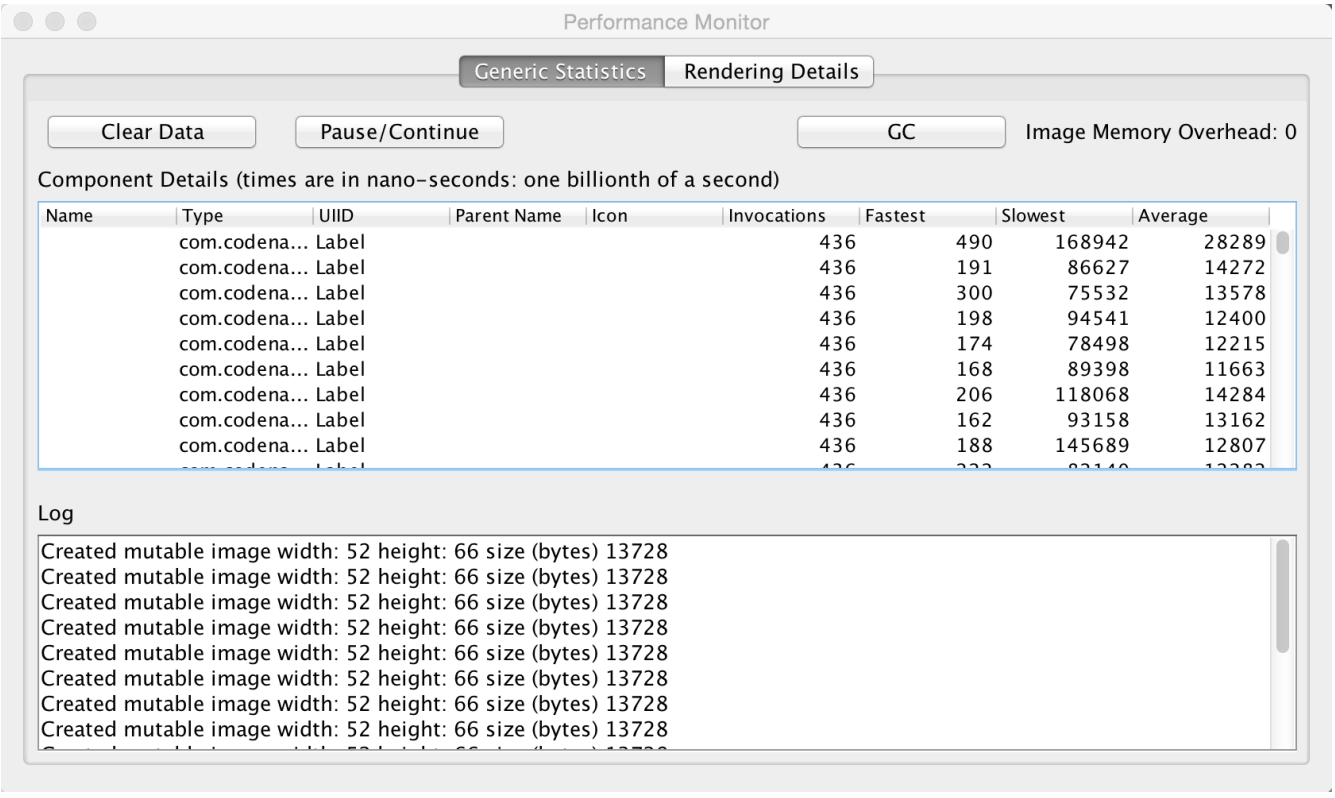


Figure 292. Main tab of the performance monitor: Logs and timings

The first tab of the performance monitor includes a table of the drawn components. Each entry includes the number of times it was drawn and the slowest/fastest and average drawing time. The toolbar across the top includes Pause/Continue buttons so you can freeze the counters while you inspect the current snapshot, a "Clear Data" action to reset the tables, and a "GC" button that invokes the simulator's garbage collector so you can see how memory usage changes.

This is useful if a `Form` is slow. You might be able to pinpoint it to a specific component using this tool.

The Log on the bottom includes debug related information. For example, it warns about the usage of mutable images which might be slow on some platforms. This also displays warnings when an unlocked image is drawn etc. A live "Image Memory Overhead" meter summarizes how much native image memory the current form consumes so you can correlate spikes with your drawing code.

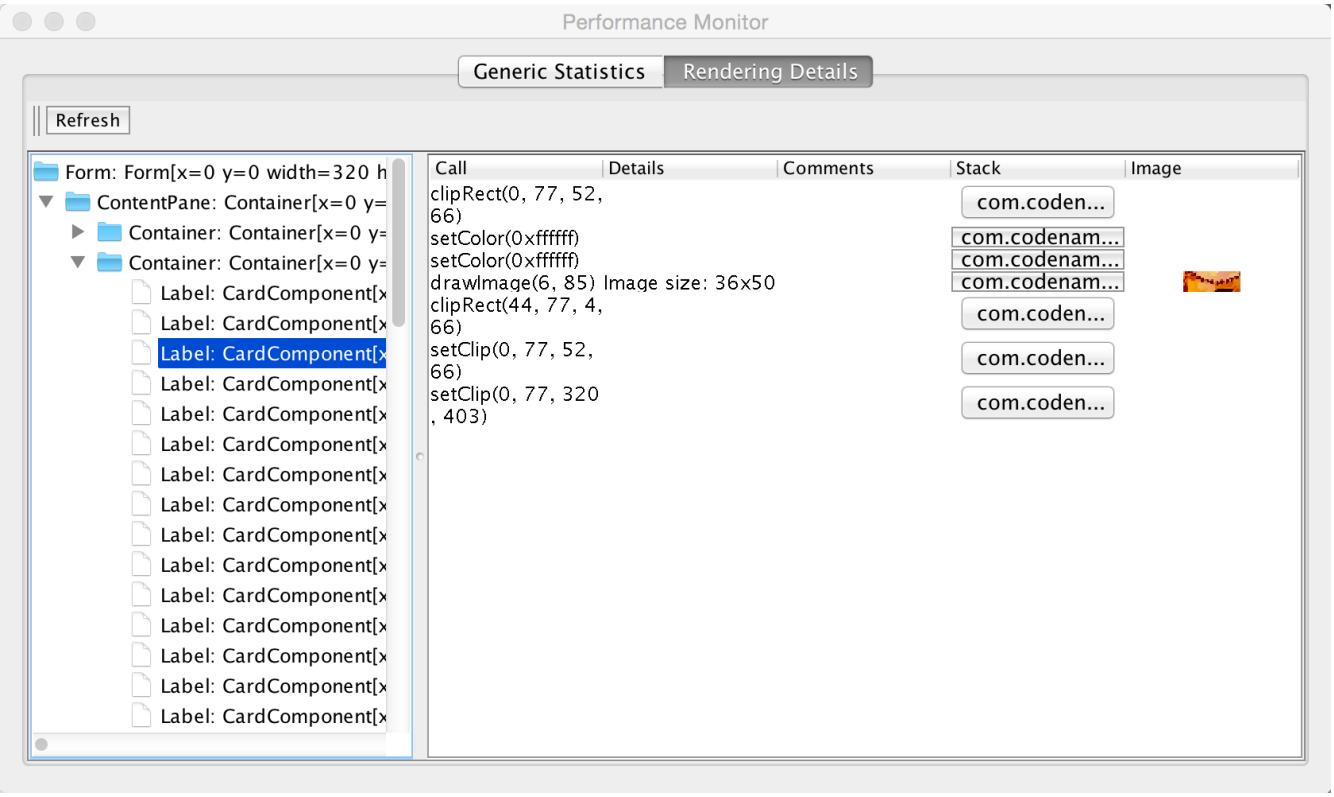


Figure 293. Rendering tree

The rendering tree view allows you to inspect the hierarchy painting. You can press the refresh button which will trigger the painting of the current `Form`. Every graphics operation is logged and so is the stack to it.

You can then inspect the hierarchy and see what was drawn by the various components. You can click the "stack" buttons to see the specific stack trace that lead to that specific drawing operation.

This is a powerful debugging tool as you can see "overdraw" within this tool. E.g if you see `fillRect` or similar APIs invoked in the parent and then again and again in the children this could show a problem.



Android devices have a nice overdraw debugging tool

52.5. Network speed

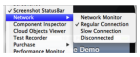


Figure 294. Network speed tool

This feature is actually more useful for general debugging but it’s sometimes useful to simulate a slow/disconnected network to see how this affects performance.

For this purpose the Codename One simulator allows you to slow down networking or even fake a disconnected network to see how your application handles such cases.

52.6. Debugging Codename One sources

When you debug your app with your source code you can place breakpoints deep within Codename One and gain unique insight. You can also use the profilers and profile into Codename One to gain similar performance specific insight.

When you run into a bug or a missing feature you can push that feature/fix back to Codename One [https://www.codenameone.com/] using a pull request. GitHub makes that process trivial and in this new video and slides below you show you how. The steps to use the code are:

1. Signup for GitHub
2. Fork http://github.com/codenameone/CodenameOne and http://github.com/codenameone/codenameone-skins (also star and watch the projects for good measure).
3. Clone the git URLs from the projects into the IDE using the `Team` → `Git` → `Clone` menu option. Notice that you must deselect projects in the IDE for the menu to appear.
4. Download the cn1-binaries project from GitHub here [https://github.com/codenameone/cn1-binaries/archive/master.zip].
5. Unzip the cn1-binaries project and make sure the directory has the name cn1-binaries. Verify that cn1-binaries, CodenameOne and codenameone-skins are within the same parent directory. .In your own project remove the jars both in the build & run libraries section. Replace the build libraries with the `CodenameOne/CodenameOne` project. Replace the runtime libraries with the `CodenameOne/Ports/JavaSEPort` project.

This allows you to run the existing Codename One project with the Codename One source code and debug into Codename One. You can now also commit, push and send a pull request with the changes.

52.7. Device testing Framework/Unit testing

Codename One includes a built in testing framework and test recorder tool as part of the simulator. This allows developers to build both functional and unit test execution on top of Codename One. It even enables sending tests for execution on the device (pro-feature).

To get started with the testing framework, launch the application and open the test recorder in the simulator menu.

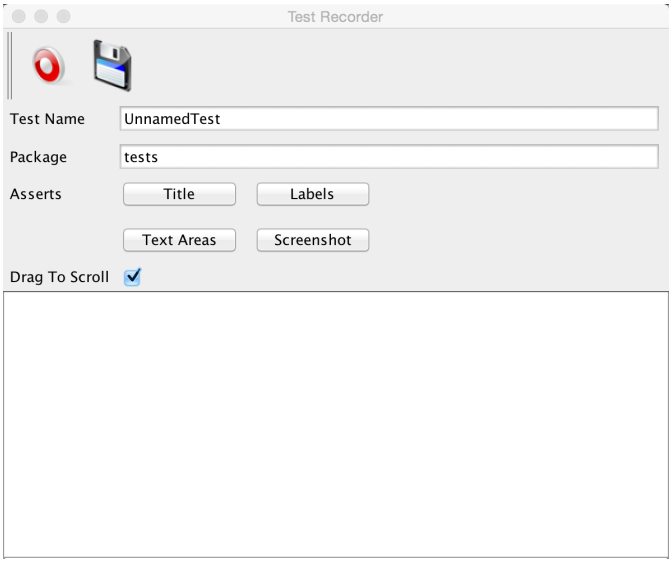


Figure 295. The test recorder tool in the simulator

Once you press record a test will be generated for you as you use the application.

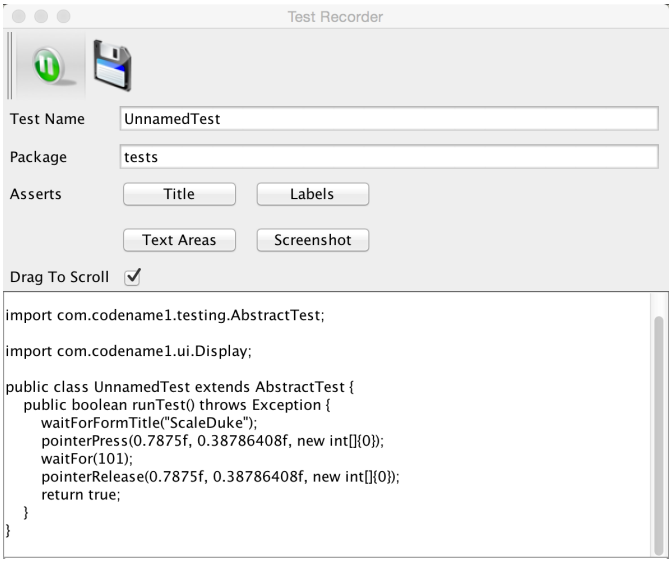


Figure 296. Test recording in progress, when done press the save icon

You can build tests using the Codename One testing package to manipulate the Codename One UI programmatically and perform various assertions.

Unlike frameworks such as JUnit which assign a method per test, the Codename One test framework uses a class per test. This allows the framework to avoid reflection and thus allows it to work on the device.

52.8. EDT error handler and sendlog

Handling errors or exceptions in a deployed product is pretty difficult, most users would throw away your app and some would give it a negative rating without providing you with the opportunity to actually fix the bug that might have happened.



Figure 297. Default error dialog

Google improved on this a bit by allowing users to submit stack traces for failures on Android devices but this requires the users approval for sending personal data which you might not need if you want to receive the stack trace and maybe some basic application state (without violating user privacy).

For quite some time Codename One had a powerful feature that allows you to both catch and report such errors, the error reporting feature uses the Codename One cloud which is exclusive for pro/enterprise users. In Codename One you catch all exceptions on the EDT (which is where most exceptions occur) and display an error to the user as you can see in the picture. This isn't helpful to you as developers who want to see the stack; furthermore you might prefer the user doesn't see an error message at all!

Codename One allows you to grab all exceptions that occur on the EDT and handle them using the method `addEdtErrorHandler` in the Display [https://www.codenameone.com/javadoc/com/codename1/ui/Display.html] class. Adding this to the Log's ability to report errors directly to you and you can get a powerful tool that will send you an email with information when a crash occurs!

This can be accomplished with a single line of code:

```
Log.bindCrashProtection(true);
```

You place this in the `init(Object)` method so all future on-device errors are emailed to you. Internally this method uses the `Display.getInstance().addEdtErrorHandler()` API to bind error listeners to the EDT. When an exception is thrown there it's swallowed (using `ActionEvent.consume()`). The `Log` data is then sent using `Log.sendLog()`.

If your crash handler runs while networking is unavailable or you want to avoid blocking the EDT, use `Log.sendLogAsync()` instead. It performs the upload in a background thread and is what Codename One's lifecycle helper falls back to when regular error reporting fails.

You can also plug in your own crash reporting pipeline by calling `Display.getInstance().setCrashReporter(CrashReport)`. The `CrashReport` [https://www.codenameone.com/javadoc/com/codename1/system/CrashReport.html] callback will receive the exception, device information, and log payload so you can forward it to services like Firebase Crashlytics or your in-house tools.

To truly gain from this feature you need to use the `Log` class for all logging and exception handling instead of APIs such as `System.out`.

To log standard printouts you can use the `Log.p(String)` method and to log exceptions with their stack trace you can use `Log.e(Throwable)`.

52.9. Kitchen sink case study

Performance is one of those vague subjects that's often taught by example.

While debugging the contacts demo (part of the new kitchen sink demo), its performance appeared sub-par. The initial assumption was that this was due to the implementation of `getAllContacts` and that there was nothing to do. Later, while debugging an unrelated issue, an anomaly was noticed

during the loading of the contacts.

This led to the discovery that you're loading the same resource file over and over again for every single contact in the list!

In the new Contacts demo you have a share button for each contact, the code for constructing a `ShareButton` looks like this:

This seems reasonable until you realize that the constructors for `SMSShare`, `EmailShare` & `FacebookShare` load the icons for each of those...

These icons are in a shared resource file that you load and don't cache. The initial workaround was to cache this resource but a better solution was to convert this code:

Into this code:

```
@Override
public Image getIcon() {
    Image i = super.getIcon();
    if(i == null) {
        i = Resources.getSystemResource().getImage("sms.png");
        setIcon(i);
    }
    return i;
}
```

This way the resource uses lazy loading as needed.

This small change boosted the loading performance and probably the general performance due to less memory fragmentation.

The lesson that you should learn every day is to never assume about performance...

52.9.1. Scroll performance - threads aren't magic

Another performance pitfall in this same demo came during scrolling. Scrolling was janky (uneven/unsmooth) right after loading finished would recover after a couple of minutes.

This relates to the images of the contacts.

To hasten the loading of contacts you load them all without images. You then launch a thread that iterates the contacts and loads an individual image for a contact. Then sets that image to the contact and replaces the placeholder image.

This performed well in the simulator but didn't do too well even on powerful mobile phones. You assumed this wouldn't be a problem because you used `Util.sleep()` to yield CPU time but that wasn't enough.

Often when you see performance penalty the response is: "move it to a separate thread." The problem is that this separate thread needs to compete for the same system resources and merge its

changes back into the EDT. When you perform something intensive you need to make sure that the CPU isn't needed right now...

In this and past cases you solved this using a class member indicating the last time a user interacted with the UI.

Here you defined:

```
private long lastScroll;
```

Then you did this within the background loading thread:

This effectively sleeps when the user interacts with the UI and loads the images if the user hasn't touched the UI in a while.

Notice that you also check if the scroll changes, this allows you to notice cases like the animation of scroll winding down.

All you need to do now is update the `lastScroll` variable whenever user interaction is in place. This works for user touches:

This works for general scrolling:



Due to technical constraints you can't use a lambda in this specific case...

Chapter 53. Testing with JUnit 5

Codename One ships two compatible test frameworks. `com.codename1.testing.AbstractTest`, driven by the `cn1:test` Maven goal, is the only way to run tests on a physical iOS or Android device — JUnit Jupiter isn't available on ParparVM. The JavaSE simulator also exposes a standard **JUnit 5** integration so you can write tests that run through Surefire and your IDE's native test runner; those tests run only in the simulator JVM but in exchange give you reflection, mocking libraries, and the rest of the standard Java testing ecosystem.

Both styles coexist in the same project under `common/src/test/java`. You pick per test class. This chapter explains when to reach for which, walks through the JUnit annotations, and shows how the two frameworks compare side by side.

53.1. When to use JUnit versus AbstractTest

Framework	Use it for	Don't use it for
`AbstractTest` + `cn1:test`	Tests that must execute on a real device (`mvn cn1:test -Dtarget=ios` / `-Dtarget=android`). Tests that must compile under the strict Codename One device subset (no reflection, no `java.net.http.*`, no `java.nio.file.*`).	Tests that need reflection, Mockito, AssertJ, parameterized data sets, `assertThrows`, or anything else that lives outside the device subset.
`@CodenameOneTest` (JUnit 5)	Simulator-only tests. Anything that wants a full JVM at test time — reflection, mocking libraries, parameterized tests, IDE green-bar integration, `mvn test -Dtest=Foo#bar` filtering, `@BeforeEach`/`@AfterEach` lifecycle methods.	Anything that must also run on a device — JUnit Jupiter isn't available on ParparVM.

The two runners discover disjoint sets of test classes (cn1:test looks for `com.codename1.testing.UnitTest` implementers; Surefire looks for `@Test`-annotated methods), so a project can mix both. `mvn install` runs Surefire during the `test` phase and `cn1:test` from the javase module's `test` profile in the same phase — you don't lose either test pass.

53.2. Dependencies

The cn1app archetype generates a `common/pom.xml` and `javase/pom.xml` that already pull in `junit-jupiter` and `codenameone-javase` at test scope. If your project predates that wiring, add these dependency blocks yourself:

Listing 19. common/pom.xml

```
<dependencies>
    <dependency>
        <groupId>com.codenameone</groupId>
        <artifactId>codenameone-javase</artifactId>
```

```
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.9.3</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Listing 20. javase/pom.xml

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.9.3</version>
    <scope>test</scope>
</dependency>
```

Tests live in `common/src/test/java`. Surefire only runs from the `javase` module — the `common` module keeps Surefire skipped (`<skipTests>true</skipTests>`) to avoid running each test twice, since `javase/pom.xml` mounts the same sources via `<testSourceDirectory>${project.basedir}/../common/src/test/java</testSourceDirectory>`.

53.3. A minimal JUnit test

Run with:

```
mvn -pl javase test                                      # all JUnit + cn1:test
mvn -pl javase test -Dtest=GreetingFormTest              # one class
mvn -pl javase test -Dtest=GreetingFormTest#formShowsExpectedTitle   # one method
```

53.4. `@CodenameOneTest`

A class-level meta-annotation equivalent to `@ExtendWith(CodenameOneExtension.class)`. The extension does four things:

1. If `GraphicsEnvironment.isHeadless()` is true, throw `TestAbortedException` from `@BeforeAll`. JUnit reports the class as **skipped** rather than failed, and the Display singleton is never left half-initialized — so subsequent test classes in the same JVM aren't poisoned. This makes JUnit-style Codename One tests safe to run on a headless CI runner; they simply skip.
2. Boot `Display` once per JVM via `Display.init(null)`. The call is idempotent, so multiple `@CodenameOneTest` classes share one Display instance for the rest of the test run.
3. Apply any visual-config annotations (`@Theme`, `@DarkMode`, `@LargerText`, `@Orientation`, `@RTL`, `@SimulatorProperty`) on the EDT in one batch before each test, followed by a single theme refresh.

4. Dispatch `@RunOnEdt` test methods through `CN.callSerially` and rethrow their exceptions on the JUnit thread so stack traces stay clickable in IDEs.

53.5. @RunOnEdt

Place on a method (or on the class for all tests) when the body mutates UI state.

Class-level `@RunOnEdt` also routes `@BeforeEach`/`@AfterEach` methods through the EDT. Method-level `@RunOnEdt` is scoped to that one `@Test`.

Tests that only exercise pure model or utility code can omit `@RunOnEdt` and run on the JUnit worker thread — they're faster because nothing pumps the EDT.

53.6. @SimulatorProperty / @SimulatorProperties

Set a property visible to the simulator before the test runs.

The `scope` field selects where the value lands:

Scope	Where	When applied
`DISPLAY` (default)	`Display.getInstance().setProperty(name, value)`	After Display init. Use for properties your app reads via `Display.getProperty(…)`.
`SYSTEM`	`System.setProperty(name, value)`	Before Display init (class-level only). Use for things the simulator reads at startup, for example `java.awt.headless`. Method-level SYSTEM properties are ignored since Display is already up by the time `@BeforeEach` runs.

The JavaSE port compiles at source 1.7, which predates `@Repeatable`. To set more than one property on the same target, wrap multiple `@SimulatorProperty` entries inside `@SimulatorProperties({…})` rather than repeating the annotation directly.

53.7. @Theme

Loads a base theme resource and installs it through `UIManager.setThemeProps`, then triggers a refresh. The annotation accepts either of two mutually exclusive inputs.

By native theme. Use the `NativeTheme` enum for the themes bundled into the simulator jar — the same set the simulator's **Simulate** > **Native Theme** menu offers:

The enum carries the resource path (`NativeTheme.IOS_MODERN.resourcePath()` returns `/iOSModernTheme.res`) and the simulator-menu label (`displayName()`), so test reports can render the same name a user sees in the simulator UI.

By resource path. For app themes shipped under `src/main/resources` or `src/test/resources`, point `value` at the `.res` file with a leading slash:

If both `nativeTheme` and `value` are set, `nativeTheme` wins. If neither is set, the annotation is a no-op.

53.8. @DarkMode

Toggles dark/light mode via `Display.setDarkMode(Boolean)` and refreshes the active form.

53.9. @LargerText

Sets the accessibility text-scale multiplier — the same knob exposed by the simulator's **Simulate → Larger Text** submenu. Useful for catching layout regressions at accessibility font sizes.

`scale = 1.0f` restores the default size; common values mirror the menu's `1.3f` / `1.6f` / `2.0f` presets.

53.10. @Orientation

Forces the simulator into portrait or landscape for the test.

This calls a non-persisting setter on `JavaSEPort` and sets an explicit-portrait flag honored by `Display.isPortrait()` — so unit-test JVMs (where the canvas inherits the host window's full size and would otherwise read landscape on every wide screen) get the expected orientation back.

53.11. @RTL

Flips the look-and-feel into right-to-left mode (Arabic, Hebrew) via `UIManager.getInstance().getLookAndFeel().setRTL(…)`. The active form is revalidated so existing layouts reflow before the test body asserts.

53.12. Annotation resolution

When the same annotation appears at both the class and the method level, **method wins**. Annotations the method doesn't override are inherited from the class. Annotations that appear on neither leave Display state alone — the extension never resets a knob the caller didn't ask for. Use `@AfterEach` for cross-test cleanup if a class must leave the simulator pristine for the next one.

53.13. EDT semantics

`@RunOnEdt` dispatches the test body through `CN.callSerially(…)` and uses a latch with `wait/notify` to block the JUnit worker thread until the EDT-side runnable finishes or `timeoutMillis` elapses. Throwables from inside the EDT runnable are captured and rethrown on the JUnit thread so the assertion stack trace lands in the IDE as if the test had been invoked directly.

Without `@RunOnEdt`, the test body runs on the Surefire worker thread — fine for pure model assertions, broken for UI mutation. Forms, components, and `UIManager` are EDT-confined; touch them off the EDT and you get sporadic deadlocks and rendering glitches.

53.14. Headless behavior

`Display.init(null)` eventually calls `JavaSEPort.init(null)`, which constructs a `javax.swing.JFrame` to host the simulator canvas. JFrame construction in a headless JVM throws `HeadlessException` immediately. Therefore:

- **Local dev (macOS / Windows / Linux desktop)**: works out of the box.
- **CI with an X server or Xvfb**: works — wrap your build with `xvfb-run mvn test` if the runner is otherwise headless.
- **CI without DISPLAY, or with explicit `-Djava.awt.headless=true`**: every `@CodenameOneTest` class aborts via `TestAbortedException` from `@BeforeAll`. JUnit reports the class as **skipped**, not **failed**, and crucially the Display singleton is never left half-initialized — so subsequent test classes in the same JVM aren't poisoned.

This is why the framework's internal `CodenameOneExtensionTest` (in the Codename One sources) carries `@DisabledIfSystemProperty(named = "java.awt.headless", matches = "true")`. The extension catches the headless case AWT detects automatically (Linux without DISPLAY); the annotation catches the explicit `-Djava.awt.headless=true` case. Together they let the test class skip cleanly on a headless runner.

Pure-logic tests (no UI, no `@CodenameOneTest`) can run on any headless runner without configuration.

53.15. Coexistence with `cn1:test`

The two runners discover disjoint sets of classes:

- `cn1:test` looks for classes that `implements com.codename1.testing.UnitTest` (which `AbstractTest` extends).
- Surefire (JUnit Jupiter) looks for `@Test`-annotated methods.

They don't trip over each other. `mvn install` runs Surefire during the `test` phase, then the `cn1:test` execution bound in the `javase/pom.xml` `test` profile runs in the same phase. To target just one runner:

```
mvn -pl javase test                          # both runners
mvn -pl javase test -DskipTests              # skip Surefire, cn1:test still runs
mvn -pl javase test -Dtest=NoMatch           # filter Surefire to nothing,
                                             # cn1:test still runs
```

53.16. Side-by-side example

The same "sign in, assert Home is shown" check, written both ways:

JUnit 5 / Surefire

The UI driving (`TestUtils.*`) is identical — `TestUtils` is independent of the runner. What changes is the lifecycle plumbing: `shouldExecuteOnEDT()` + `runTest() returns boolean` becomes `@RunOnEdt` + `@Test`

`void` + a standard assertion library.

53.17. Cross-reference

- For the `AbstractTest` framework's full API surface (`TestUtils` helpers, `screenshotTest` tolerance algorithm, baseline management), see Screenshot Testing in the performance chapter.
- For the `cn1:test` Maven goal, see the test goal appendix.

Chapter 54. Advertising

Codename One provides a pluggable advertising API under `com.codename1.ads`. The framework defines the formats, the consent flow and the app/UI lifecycle integration; an ad network is supplied by a separate library that implements the provider interface. The reference implementation is the Google AdMob library (`cn1-admob`), but the design is network agnostic - AdMob, AppLovin MAX, Unity LevelPlay or a custom mediation layer all plug in the same way, with no changes to the framework or the build.

The supported formats are:

- **Banner** - a small inline ad that lives in the component hierarchy (`com.codename1.ads.BannerAd`).
- **Interstitial** - a full screen ad shown at a natural break, such as between game levels (`com.codename1.ads.InterstitialAd`).
- **Rewarded** - an opt-in full screen ad that grants a reward for watching (`com.codename1.ads.RewardedAd`).
- **Rewarded interstitial** - an incentivized transition ad (`com.codename1.ads.RewardedInterstitialAd`).
- **App open** - shown when the app is brought to the foreground (`com.codename1.ads.AppOpenAd`).
- **Native** - assets you render with your own components (`com.codename1.ads.NativeAd`).

The screenshot below is the advertising sample running in the simulator against the deterministic mock provider (`cn1-ads-mock`): a news feed with an in-feed native ad (the "Sponsored" row, rendered with the app's own components) and an anchored banner at the bottom.

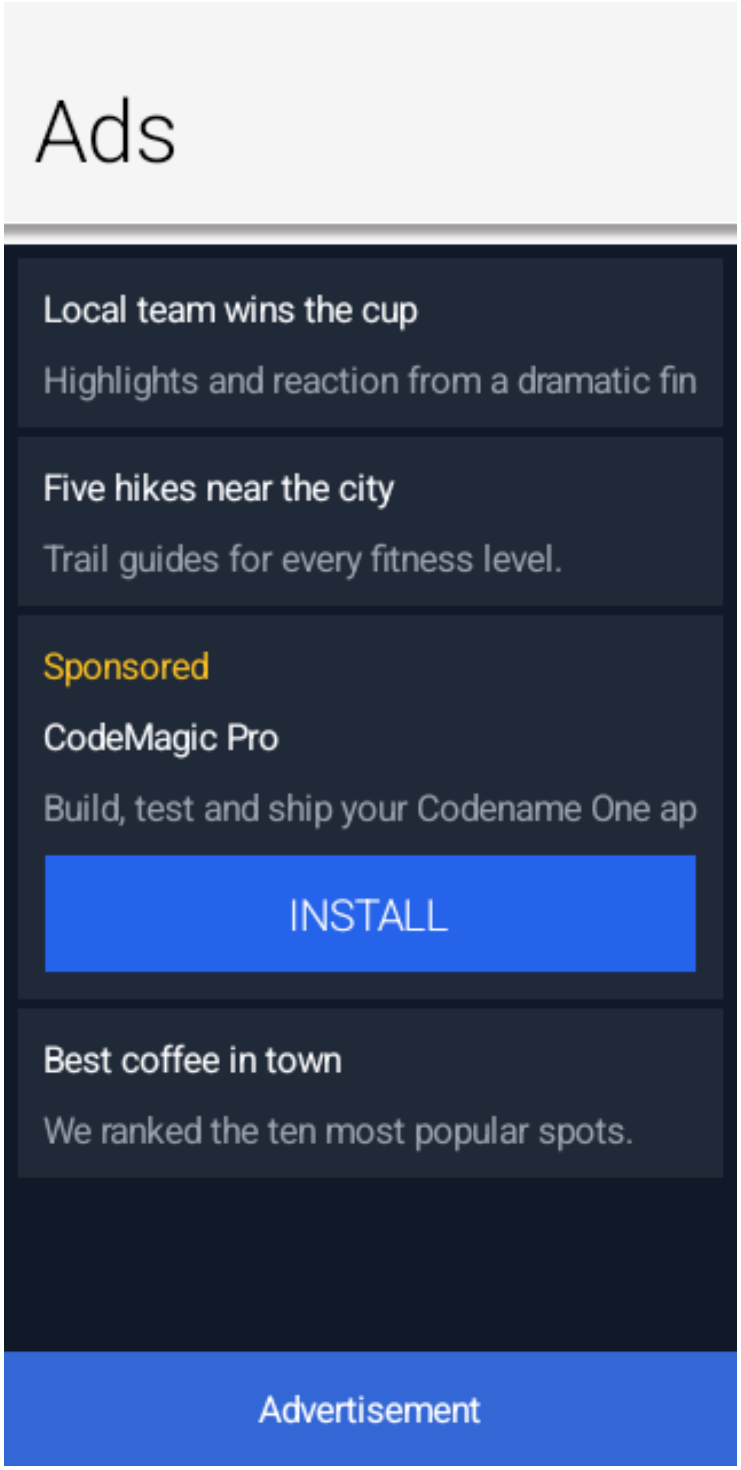


Figure 298. The advertising sample in the simulator: an in-feed native ad and a banner

54.1. Ad provider cn1libs

Several provider libraries ship with the framework. They're released to Maven Central together with the core and the AI cn1libs, sharing the same version (`${cn1.version}`), so you never mix versions. Each declares its own native SDK dependencies (Pods / Gradle dependencies / permissions), applied automatically when the library is on the classpath; you don't edit build hints by hand, except for the per-app id described under Section 54.9, "Configuring the AdMob application id".

cn1lib	Provider class	Provides
cn1-admob	com.codename1.ads.admob.AdMobProvider	Google AdMob on the Google Mobile Ads SDK (banner, interstitial, rewarded, rewarded interstitial, app open), with AdMob mediation behind it.
cn1-applovin	com.codename1.ads.applovin.AppLovinProvider	AppLovin MAX mediation (banner, interstitial, rewarded, app open).
cn1-unity-levelplay	com.codename1.ads.levelplay.LevelPlayProvider	Unity LevelPlay (ironSource) mediation (banner, interstitial, rewarded).
cn1-ads-mock	com.codename1.ads.mock.MockAdProvider	A deterministic, network-free provider that renders fixed labelled ads - for unit tests, screenshot tests and trying the API in the simulator.

Add the matching dependency to your project's `common/pom.xml`. The provider libraries use a `-lib` aggregator with `<type>pom</type>` so Maven pulls in the per-platform classifier jars:

```
<dependency>
    <groupId>com.codenameone</groupId>
    <artifactId>cn1-admob-lib</artifactId>
    <version>${cn1.version}</version>
    <type>pom</type>
</dependency>
```

`cn1-ads-mock` is a single cross-platform jar, so it needs no classifier:

```
<dependency>
    <groupId>com.codenameone</groupId>
    <artifactId>cn1-ads-mock</artifactId>
    <version>${cn1.version}</version>
</dependency>
```

54.2. Enabling a provider

Enable a provider once at startup. Each library exposes a static `install()` method that registers its provider with `AdManager`:

That single call binds the provider; the rest of your code uses only the framework API in `com.codename1.ads`. If no provider is registered the API stays safe: format support reports `false` and loads fail with `AdError.CODE_UNSUPPORTED` instead of throwing, so a build without an ad library still runs.

54.3. Initialization and consent

Collecting privacy consent is mandatory on modern platforms before personalized ads can be served: in the EEA/UK the GDPR consent form must be shown (providers wrap Google's User Messaging Platform or an equivalent), and on iOS the App Tracking Transparency prompt must be presented to access the advertising identifier. The recommended order is to initialize, gather consent, then load:

`AdConfig` also carries the global compliance flags every network requires: test mode, test device ids, child directed treatment, under-age-of-consent treatment and a maximum ad content rating.



Never ship a release build with `testMode` on, and never click live ads during development - both violate ad network program policies. Use the test ad unit ids while developing.

54.4. Banner ads

`BannerAd` is a regular Codename One component. Add it to a form (typically anchored at the top or bottom) and call `load()`:

The default `SIZE_ADAPTIVE` requests an anchored adaptive banner sized to the available width, which is the recommended modern banner type.

54.5. Interstitial ads

Interstitials are event driven - load, then show when loaded, and preload the next one when the current ad is dismissed:

You can also let Codename One show an interstitial automatically on screen transitions, no more often than a given interval:

54.6. Rewarded and rewarded interstitial ads

Register a reward listener and grant the reward when it fires. For valuable rewards, verify server side rather than trusting the client:

`RewardedInterstitialAd` has the same API but is shown on a transition rather than opt-in.

54.7. App open ads

App open ads are shown while the app is brought to the foreground. Let the manager and provider handle the foreground hook and freshness window for you:

54.8. Native ads

Native ads let you render the advertiser's assets with your own components, so the ad matches the look and feel of the surrounding content. This is the format to reach for in content driven apps - a

news or social feed, a store listing, a chat or a search results screen - where a banner feels bolted on but a row styled like the others fits in. The ad must still be clearly labelled (for example "Sponsored"):

`NativeAd` exposes the headline, body, call-to-action, advertiser and rating that you bind to your own components. Native ad support is an optional provider capability; when the active provider doesn't support it, `NativeAdLoader` reports the format as unsupported.

54.9. Configuring the AdMob application id

The GMA SDK requires your per-app AdMob application id in the native project. Because it differs per app it's not baked into the library; set it with the standard build hints in your project's `codenameone_settings.properties`:

```
codename1.arg.android.xapplication=<meta-data android:name="com.google.android.gms.ads.APPLICATION_ID" android:value="ca-app-pub-XXXXXXXX~YYYYYYYY"/>
codename1.arg.ios.plistInject=<key>GADApplicationIdentifier</key><string>ca-app-pub-XXXXXXXX~YYYYYYYY</string>
```

The SDK dependencies themselves (the GMA pod / Gradle dependency and the INTERNET permission) are declared by the `cn1-admob` library, so they're added automatically when the library is on the classpath.

54.10. Implementing a provider

Third party networks plug in by implementing `com.codename1.ads.spi.AdProvider` (and the optional `NativeAdProvider` capability) and calling `AdManager.registerProvider(…)` - conventionally from a static `install()` method. No framework or build-tool changes are required.

For tests and screenshots the framework ships a deterministic, network-free provider, `MockAdProvider` (in `cn1-ads-mock`), that renders fixed, labelled ads with stable sizes and colours.

Chapter 55. Crash Protection

Crash Protection is an opt-in service that captures uncaught exceptions in your shipping app and files them as deduplicated issues on your GitHub repository. Symbolicated stack traces, scrubbed messages, and a per-bug counter are all recorded server-side; you triage from GitHub Issues like any other bug.

The service is available on the **Pro** and **Enterprise** subscription tiers. See the pricing page [https://www.codenameone.com/pricing.html] for current limits.

55.1. What gets stored where

Three things move when you opt in:

1. **Symbol bundles** (ProGuard `mapping.txt` on Android, dSYM on iOS, debug-info-bearing binaries on Mac/Linux/Win32). Uploaded by the Codename One build server after each successful release build. Stored compressed; subject to per-tier storage quota.
2. **Crash reports** (one per device occurrence). Sent by the app at runtime via `CrashProtection.capture(t)` or the automatic EDT error handler. Scrubbed on the device before transmission (see *Privacy* below).
3. **GitHub issues** — the actual triage surface. One issue per unique crash fingerprint. Subsequent occurrences of the same fingerprint bump a counter on the issue body rather than filing duplicates.

Codename One never reads or persists the unscrubbed message body. The crash data store is the issue tracker on your repository, not a separate database. The service keeps only the minimum lookup state needed to find the right issue on the next occurrence.

55.2. Opting in

Add this line to your app's `codenameone_settings.properties`:

```
codename1.arg.crashProtection.enabled=true
```

The `codename1.arg.` prefix marks it as a build hint — when the property reaches the Codename One build server, it tells the build to upload the symbol bundle (PDB / dSYM / .debug / mapping.txt) to the crash-protection service alongside the normal release artifact.

55.2.1. Runtime hook

In your `Lifecycle.init`:

`install()` registers the EDT error handler and (on Android) wires `Thread.UncaughtExceptionHandler`. It's idempotent and a no-op on the simulator. `setEnabled(boolean)` controls whether captured crashes actually get sent to the server — the captured payload is always persisted locally first, so `setEnabled(true)` after the fact drains the buffer.

You can also report a caught-but-noteworthy exception with `CrashProtection.capture(e)`.

55.3. Per-platform opt-out

Sometimes one platform's symbols are noisy or you don't ship that target. Disable a platform with:

```
codename1.arg.crashProtection.and.enabled=false
codename1.arg.crashProtection.ios.enabled=false
codename1.arg.crashProtection.mac.enabled=false
codename1.arg.crashProtection.linux.enabled=false
codename1.arg.crashProtection.win.enabled=false
```

Setting one of these to `false` makes `CrashProtection.install()` a no-op at runtime on that platform AND skips the build-time symbol upload for it. Setting to `true` (or leaving unset) keeps the platform enabled.

The runtime check uses `Display.getInstance().getPlatformName()`. Supported values: `and`, `ios`, `mac`, `linux`, `win`, `javascript`, `javase`.

55.4. Tier limits

Tier	Storage quota	Symbol retention
Pro	100 MB compressed	3 weeks
Enterprise	500 MB compressed	6 weeks

Symbol bundles are gzipped before storage. `mapping.txt` files typically compress ~10x; dSYMs ~2x. The quota admits several recent builds per app for a typical developer.

When a new symbol upload would push you over the quota, the build's symbol step is rejected with HTTP 413 (the build itself succeeds). You can free space from the **Tracked apps** tab in the console — delete an old build's symbols or remove a whole app's enrollment.

55.5. Privacy

55.5.1. What's PII

PII is short for *personally identifiable information*: any field in a crash payload that could be traced back to a specific user. The pieces of a Java exception message that are typically PII are email addresses, account numbers, phone numbers, IP addresses, full URLs to authenticated endpoints (with session tokens in the query string), and the contents of user input fields that ended up in the message via string concatenation. Stack trace frames are class / method / file / line — they describe code paths, not users, so they're not PII.

The Codename One crash-protection client runs incoming messages through a scrubber before transmission. The scrubber can be replaced with a stricter rule set if your application's exception messages routinely carry data you don't want to leave the device.

55.5.2. What gets sent

`CrashReportPayload` carries these fields. The marked one goes through `PiiScrubber` before transmission.

- `eventId` (UUID, used for idempotent dedup)
- `buildKey` (anti-spoofing token; matched against the uploaded symbol bundle)
- `packageName`, `appVersion`, `platform`, `osVersion`, `locale`
- `exceptionClass`
- `messageScrubbed` — **scrubbed**
- `frames[]` — class / method / file / line / `native` flag per frame
- `deviceMeta` — free memory + locale only; not device IDs
- `clientTs`

55.5.3. Default scrubber rules

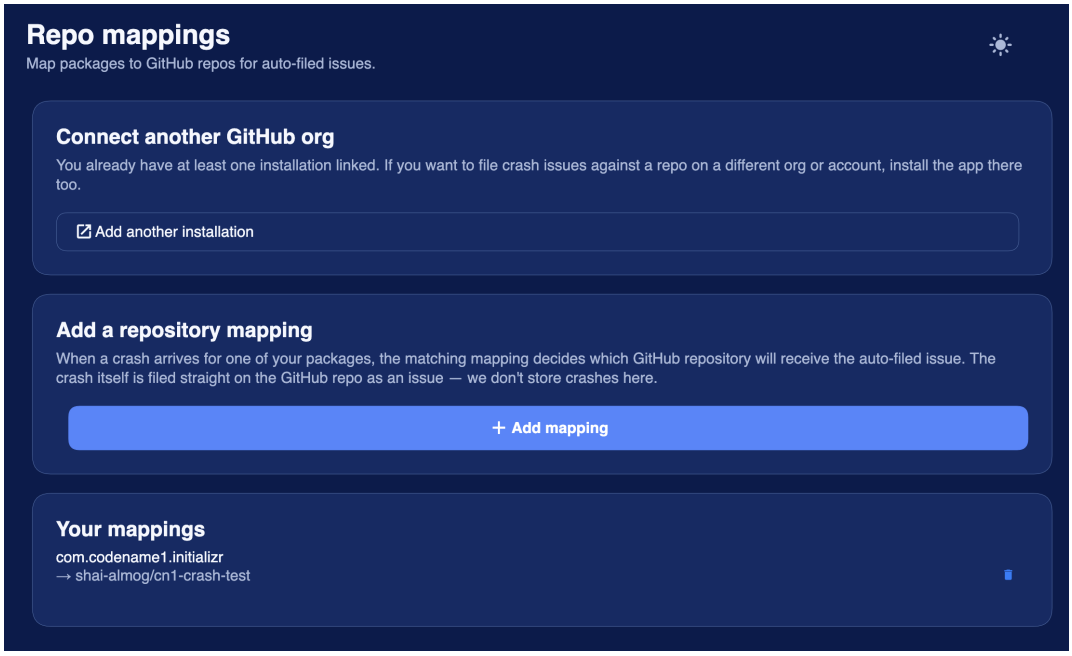
`PiiScrubber` applies these by default:

- Email addresses: redacted to the first 3 characters of the local part. `joe.smith@example.com` becomes `joe***@example.com` (the full domain is preserved so the issue still tells you *which* provider was involved).
- Long digit runs: any run of 6 or more consecutive digits collapses to `[num]`. Catches phone numbers, order IDs, account numbers, ZIP+suffix, etc.
- URLs aren't scrubbed by default. Override the scrubber if your app's URLs carry session tokens.

Override `scrubMessage(String)` or `scrubFrame(String, String)` on a `PiiScrubber` subclass and register via `CrashProtection.setScrubber(myScrubber)` if you need stricter rules.

55.6. Mapping packages to repositories

In the console at https://cloud.codenameone.com/console/index.html, open **Repo mappings**:



Click **Add mapping** and supply:

1. **Package name** — the exact value from `codenameone_settings.properties`.
2. **Installation** — which authorized GitHub App installation should file the issue. Only shown if you have more than one installation linked.
3. **Repository** — pick from the dropdown of the installation's repos.

You must have authorized the Codename One Crash Protection GitHub App for at least one of your repositories first. The mapping page renders a button labeled **Connect on GitHub** that opens the App install flow on github.com. The post-install redirect brings you back to the same page with the installation already loaded; if you already have one installation and want to add another (for example a second GitHub org), the **Connect another GitHub org** card lets you link an additional one without disturbing the existing mapping.

55.7. Setup checklist

1. Pro or Enterprise subscription.
2. Add `codename1.arg.crashProtection.enabled=true` to `codenameone_settings.properties`.
3. Call `CrashProtection.install()` in `Lifecycle.init`. Call `CrashProtection.setEnabled(true)` once you have the user's consent.
4. Build your app for release. The Codename One build server uploads the symbol bundle automatically.
5. Open the console, go to **Repo mappings**, click **Connect on GitHub**, and install the App on the repo you want crashes to land in.
6. Add a mapping for your package → repo.
7. Crashes from your release builds will now appear as GitHub issues on the mapped repo.

Chapter 56. Analytics

The analytics API records how your application is used in the field. It's built around a small provider SPI: you register one or more providers with the `Analytics` entry point, then report screen views, events, user properties and crashes. The `Analytics` class fans every call out to all registered providers, but only after the matching consent has been granted. Consent is central to this API and is explained in Consent and privacy below.

Codename One ships several providers: a first-party service that reports into the Codename One cloud (with capabilities gated by your subscription tier), Google Analytics 4, Matomo (a privacy-first, self-hostable option), Firebase, and a logging provider for the simulator and tests. You can also write your own provider for any other backend.



The older `AnalyticsService` class still works but is deprecated. It now delegates to the API described here. See Migrating from AnalyticsService at the end of this chapter.

56.1. Registering providers

Register providers once, early in your app's lifecycle. Providers are plain objects — there is no reflection or service lookup, which keeps the mechanism safe under the obfuscation applied to release builds.

Once at least one provider is registered, report usage from anywhere in your app:

`Analytics.crash` emits a lightweight exception event (for example GA4's `app_exception`) to whichever analytics backends you have registered, so exception counts appear next to your usage metrics. It's separate from Codename One Crash Protection (`com.codename1.crash.CrashProtection`), which is a dedicated pipeline that records full, symbolicated stack traces to the build cloud crash console for debugging. The two are independent: use Crash Protection to diagnose crashes, use `Analytics.crash` when you also want exception events in your analytics, and enable either or both.

56.2. Consent and privacy

The API is designed to help you comply with privacy regulations such as GDPR and CCPA. A `ConsentMode` controls what happens before the user makes an explicit choice. The default is `OPT_IN`: nothing is collected or transmitted until the application records consent.

`AnalyticsConsent.all()` grants every category and `AnalyticsConsent.none()` denies every category. The older `granted()` and `denied()` methods remain as aliases.

Consent is broken down by category, so you can honor granular choices — for example allowing crash reporting while declining behavioral analytics:

The consent choice is persisted, so it survives restarts. Screen views and events require the `analytics` category, crash reports require `crashReporting`, and `setUserId` requires `personalization`. Calls made without the matching consent are dropped.

56.2.1. Consent you already hold

If your application has already obtained consent through its own mechanism — a custom prompt, a third-party consent-management platform, an enterprise device-management policy, or a jurisdiction where you have determined consent isn't required — record it once at startup so reporting flows without a second prompt:

You can also seed every category and then revoke just one with the builder, for example to allow analytics but not ad storage:

If you would rather not gate at all, switch the default so collection is active unless the user withdraws it. This places the compliance responsibility on you as the integrator:

Note that the service records pseudonymous data rather than anonymous data. The client id described below is a persistent identifier stored on the device; it isn't tied to a real-world identity and the server discards full IP addresses, but storing and reading such an identifier for usage tracking is often treated as consent-requiring under GDPR and similar regimes. That's why opt-in is the safe default. Relax it only when you are confident the change fits your use.

56.2.2. Client identity and erasure

Each device is identified by a pseudonymous client id. It's generated on first use and stored locally — it isn't derived from any hardware identifier. To honor an erasure request ("right to be forgotten"), reset it:

For the first-party service this can be paired with a server-side deletion of the previous id's data.

56.3. Automatic events

Codename One emits a few events for you, so the common funnels work without any wiring. Like every report these are consent-gated and do nothing until a provider is registered, so they're always safe.

Event	When it fires
`purchase_initiated`	`Purchase.purchase(sku)` is called — the start of the checkout funnel.
`purchase`	A purchase completes and its receipt is posted. The gap between this and `purchase_initiated` is your checkout drop-off.
`login`	A social login (`com.codename1.social.Login`) succeeds. The login provider class is reported in the `service` parameter.
`login_failed`	A social login fails, with the provider in `service` and the reason in `error`.
`share`	The native share sheet is invoked.

You can add your own events at any time with `Analytics.event(…)`; these built-ins simply save you the boilerplate for the most common funnels.

56.4. Built-in providers

56.4.1. Codename One first-party service

`CodenameOneAnalyticsProvider` batches events and posts them to the Codename One cloud, which aggregates them into the reports shown in the developer console. App identity is read from the build-injected properties, so no API key is embedded in your app.

The capabilities you actually get — screen views, custom events, user properties, retention window, raw export — are gated server-side by your subscription tier. Higher tiers retain data longer and unlock richer reporting.

56.4.2. Google Analytics 4

`GoogleAnalyticsProvider` uses the GA4 Measurement Protocol. Create it with a measurement id (`G-XXXXXXXX`) and a Measurement Protocol API secret from the GA4 admin console:

Screen views are sent as the GA4 `screen_view` event and crashes as `app_exception`.

56.4.3. Matomo

`MatomoAnalyticsProvider` targets Matomo (formerly Piwik) through its HTTP tracking API. Matomo can be self-hosted and supports IP anonymization, which makes it a good fit for privacy-sensitive deployments:

56.4.4. Firebase

`FirebaseAnalyticsProvider` forwards to the native Firebase Analytics SDK on Android and iOS. Register it like any other provider:

Firebase reports only from a native Android or iOS build, and only when two things are in place. Without both, `FirebaseAnalyticsProvider` does nothing — which is why it's always safe to register, including in the Codename One simulator and the desktop build, where it's a no-op.

First, add the Firebase configuration file for your app (downloaded from the Firebase console) to the matching platform module's resources:

- Android: `android/src/main/resources/google-services.json`
- iOS: `ios/src/main/resources/GoogleService-Info.plist`

These are Maven project paths. The `native/android` and `native/ios` directories from the old Ant projects no longer apply.

Second, enable the SDK with build hints:

```
codename1.arg.android.firebaseAnalytics=true
codename1.arg.ios.firebaseAnalytics=true
```

These hints add the Firebase Analytics Gradle dependency (Android) and the `Firebase/Analytics` pod (iOS), and make the build generate the native bridge that `FirebaseAnalyticsProvider` talks to.



You need both the configuration file and the matching build hint. If either is missing the provider stays a no-op and no events are sent. Firebase events never show up in the Codename One simulator or a desktop run — verify on a real device (or a platform emulator/simulator) with the Firebase project configured for the build.

56.4.5. Logging provider

`LoggingAnalyticsProvider` logs every call instead of sending it anywhere. It's the recommended default in the simulator and is used as the backing provider in tests.

56.5. Writing your own provider

Implement `AnalyticsProvider`, or extend `AbstractAnalyticsProvider` and override only the calls you support. The `Analytics` class handles consent gating and fan-out for you, so a provider just forwards the data to its backend.

The `init(AnalyticsContext)` callback hands you the app name, version, platform, locale and the pseudonymous client id. `onConsentChanged(AnalyticsConsent)` lets a provider reconfigure when the user updates consent. `supports(AnalyticsCapability)` lets tooling introspect which features a provider offers.

56.6. Migrating from AnalyticsService

The deprecated `AnalyticsService` continues to compile and now routes through this API. To preserve its historical always-on behavior — which predates the consent model — `AnalyticsService.init` switches the default consent mode to `OPT_OUT`. New code should migrate to `Analytics` and manage consent explicitly:

Deprecated call	Replacement
`AnalyticsService.init(agent, domain)`	`Analytics.addProvider(new GoogleAnalyticsProvider(id, secret))`
`AnalyticsService.visit(page, referer)`	`Analytics.screen(page, referer)`
`AnalyticsService.sendCrashReport(t, msg, fatal)`	`Analytics.crash(t, msg, fatal)`
subclassing `AnalyticsService`	implement `AnalyticsProvider`

Chapter 57. Commerce

Commerce is an optional service that validates your in-app purchases and subscriptions server-side and gives your app a single, store-agnostic question to ask: *does this user have this entitlement right now?* It wraps the existing Purchase API (see the *Monetization* chapter) — purchases still go through the platform store — and adds cloud receipt validation, a normalized subscription state machine across Apple and Google, lifecycle webhooks to your backend, and a revenue console.

Codename One doesn't process payments and never touches your money — the stores (and, for physical goods, your own payment processor) do. Commerce is infrastructure that sits beside the money flow, exactly like the third-party services it replaces.

57.1. Entitlements, not SKUs

The core idea is the **entitlement**: an abstract access right such as `pro` or `remove_ads`. You map one or more store products to an entitlement, and your code only ever checks the entitlement:

An entitlement is active when **any** of its granting products is active. This matters: a promotional grant and a paid subscription, or a sandbox and a production purchase, can both map to `pro` without one hiding the other. Access is decided by a single authoritative rule — the latest expiry over the active grants, evaluated against a server timestamp — so a tampered device clock can't extend it. Everything else (will-renew, billing stage, period type, family sharing) is advisory and never removes access on its own.

57.2. Client usage

In your `Lifecycle.init` (or wherever you set up purchases):

Drive purchases through the manager (these delegate to the `Purchase` API):

After a purchase, or on app start, validate the device's receipts with the cloud and refresh the entitlement cache. `refresh()` makes a blocking network call, so run it off the EDT:

`isEntitled` returns the last cloud-validated answer when one is available, and otherwise falls back to the platform's own receipt (treating the entitlement id as a subscription SKU) so a paying user is never locked out while the network is down.

57.3. How the app finds the service

Commerce reuses the standard `build_key` that every Codename One cloud build already carries, the same token the analytics provider sends — so there is **nothing to wire into the build**. `CommerceManager.isCloudEnabled()` is true on any cloud build and false in the simulator or a local build (where every call defers to `Purchase`). Two optional build hints tune it:

Build hint	Default	Description
`commerce.cloud.enabled`	`true`	Set to `false` to switch the cloud path off in a cloud build; every call then defers to the platform `Purchase` API.

Build hint	Default	Description
`commerce.cloud.endpoint`	*(Codename One cloud)*	Base URL of the commerce service. Override only if you self-host.

57.4. Server notifications

For the entitlement state to stay correct after the initial purchase — renewals, cancellations, refunds, billing retries, grace periods, holds — point the stores' server notifications at the service:

1. **Apple** — in App Store Connect, set the App Store Server Notifications V2 URL to `https://cloud.codenameone.com/api/v2/commerce/notifications/apple`.
2. **Google** — in the Play Console, set Real-time developer notifications and route the Pub/Sub topic to `https://cloud.codenameone.com/api/v2/commerce/notifications/google`.

Notifications are idempotent (deduplicated on Apple's `notificationUUID` / Google's Pub/Sub `messageId`), so a redelivery never double-counts.

57.5. Server-side validation (connect your store credentials)

By default the service validates the store-signed receipt the device presents — enough to gate features, and it works with no setup. For **server-authoritative** validation — where the service calls Apple's and Google's own server APIs to confirm the real subscription state — provide your store credentials in the console **Secrets** page:

- **Apple** — an App Store Connect API key, as three secrets: `commerce.apple.keyId`, `commerce.apple.issuerId`, and `commerce.apple.privateKey` (the contents of the `.p8` file).
- **Google** — a service account with Android Publisher access, as `commerce.google.serviceAccountJson` (the full JSON key).

Secrets are stored AES-256-GCM encrypted in the BuildCloud vault; they're decrypted only server-side at call time and are never shown in the console again. Once set:

- Entitlements are confirmed against Apple's `Get All Subscription Statuses` and Google's `subscriptionsv2.get`, so a grant is server-verified rather than device-trusted, and
- Subscription **renewals** signalled by store notifications are recorded at their real amount (the renewal price lives only in the store's server response).

Without credentials the service stays in device-trusted mode — everything still works, you just don't get the server cross-check or accurate renewal revenue.

57.6. Lifecycle webhooks to your backend

Commerce can forward every normalized state change to your own backend. Configure a URL and a signing secret on the **Commerce** page of the console. Each event is HMAC-SHA256 signed (header `CN1-Commerce-Signature`) and carries:

- A **per-customer monotonic** `sequence` — so your consumer can reject stale events and never has to reorder defensively,
- The grant's `state_version` — reconcile against this,
- An always-present `effective_expiration_at`, and
- The event `id` for idempotent dedup.

Delivery is at-least-once, in sequence order per customer, with retries that back off exponentially. Event types mirror the familiar set: `INITIAL_PURCHASE`, `RENEWAL`, `CANCELLATION`, `UNCANCELLATION`, `EXPIRATION`, `BILLING_ISSUE`, `PRODUCT_CHANGE`, `SUBSCRIPTION_PAUSED`, `REFUND_REVERSED`, `TRANSFER`.

57.7. The console

The **Commerce** section in the console at https://cloud.codenameone.com/console/index.html is account-level by default, with an app filter to narrow to one app:

- **Overview** — the subscription dashboard: headline KPIs (MRR, ARR, ARPU, active subscriptions, active trials, churn, trial conversion, customers) and trend charts for MRR, active subscriptions and revenue.
- **Insights** — MRR movement (new vs churned), MRR forecast, realized-LTV curve, active subscriptions by store and by product, cohort-retention, and period-over-period MRR.
- **Customers** — each app user and their active entitlements.
- **Transactions** — the recent validated-event ledger.
- **Webhook** — your delivery endpoint and signing secret.

57.8. Analytics

Commerce normalizes every active subscription to a monthly figure (MRR) and reconstructs the trends from the subscription history, so the dashboard matches what dedicated subscription-analytics tools provide:

- **MRR** / **ARR** / **ARPU** — normalized recurring revenue, annualized, and per active subscription.
- **Churn** and **trial conversion** rates, **refund rate**.
- **Cohort retention** — monthly acquisition cohorts and the percentage still active each subsequent month.
- **Realized LTV** — average cumulative revenue per paying customer by months since acquisition.
- **Forecast** — projected MRR for the coming months from the current run-rate and recent growth.
- **Breakdowns** by store and by product, and **period-over-period** comparison.

Every chart pairs a graph with its underlying data table (one tab-flip away) for export-friendly inspection. All money is shown in your reporting currency; transactions in other currencies are normalized to USD via daily exchange rates.

57.9. Privacy

Commerce stores entitlement state and transaction metadata (store, product id, event type, amount, validation state) keyed by your app-user id. It never stores card numbers or payment credentials — those stay with Apple, Google, or your processor. Family shared purchases grant access but are excluded from revenue. The minimum lookup state needed to resolve the next notification is kept; grants are retained as the durable record, while the high-churn ledger and delivered webhook rows are purged on a rolling window.

57.10. Setup checklist

1. Map your store products to entitlements in the console.
2. In the app, call `CommerceManager.getInstance().subscribe(…)` / `.purchase(…)`, then `.refresh()` off the EDT, and gate features with `.isEntitled(…)`.
3. Point Apple App Store Server Notifications V2 and Google RTDN at the service notification URLs.
4. (Optional) Set a webhook URL + secret on the **Commerce** console page to receive lifecycle events on your backend.

Chapter 58. Monetization

Codename One integrates many built-in monetization options such as in-app purchases and subscriptions. Advertising is covered in its own chapter; see Advertising.

Many monetization options are available as third-party cn1libs [https://www.codenameone.com/cn1libs.html] that you can install through the Codename One website.

58.1. In-app purchase

In-app purchase is a helpful way to make app development profitable. Codename One supports in-app purchases of consumable and non-consumable products on Android and iOS. It also supports subscriptions. Even though the concept is simple, in-app purchase involves many moving parts, for subscriptions.

58.1.1. The SKU

In-app purchase support centers around the set of SKUs you want to sell. Each product, whether it's a one-month subscription, an upgrade to the "Pro" version, or "10 disco credits," has a SKU (stock-keeping unit). Ideally, you can use the same SKU across every store that sells your app.

58.1.2. Types of products

Four classifications for products exist:

1. **Non-consumable Product** - This is a product that the user purchases once to "own." They can't re-buy it. One example is a product that upgrades your app to a "Pro" version.
2. **Consumable Product** - This is a product that the user can buy more than once. For example, you might have a product for "10 Credits" that lets the user buy items in a game.
3. **Non-Renewable Subscription** - A subscription that you buy once, and won't be "auto-renewed" by the app store. These are almost identical to consumable products, except that subscriptions need to be transferable across all the user's devices. This means that non-renewable subscriptions require that you have a server that keeps track of the subscriptions.
4. **Renewable Subscriptions** - A subscription that the app store manages. The user will be automatically billed when the subscription period ends, and the subscription will renew.



These subscription categories may not be explicitly supported by a given store, or they may use different names. You can integrate each product type in a cross-platform way using Codename One. For example, Google Play doesn't distinguish between consumable products and non-renewable subscriptions, but iTunes does.

58.1.3. The "hello world" of in-app purchase

Start with a simple example of an app that sells `Worlds`. First, pick the SKU for the product. Here you use `com.codename1.world`:

```
public static final String SKU_WORLD = "com.codename1.world";
```



Although this example uses the package-name convention for a SKU, you can use any name you want, for example `UA8879`.

Next, the app's main class needs to implement the `PurchaseCallback` interface

Using these callbacks, the app receives notifications whenever a purchase changes. For this simple app, only `itemPurchased()` and `itemPurchaseError()` matter. The legacy `itemRefunded()`, `subscriptionStarted()`, and `subscriptionCanceled()` hooks are deprecated in the core API and are no longer dispatched by the stores. Instead of relying on callbacks for long-term entitlement state, query the current receipts at runtime with helpers such as `Purchase.wasPurchased(…)`, `Purchase.isSubscribed(…)`, or `Purchase.getReceipts()`. That keeps the UI aligned with the latest data from the underlying store.

Now in the `start()` method, add a button that lets the user buy the world:

At this point, the app can track the sale of the world. To make it more useful, add `ToastBar` feedback for purchase completion:



You can test out this code in the simulator without doing any more setup and it will work. If you want the code to work on Android and iOS, you'll need to set up the app and in-app purchase settings in the Google Play and iTunes stores respectively as explained below

When the app first opens you see your button:

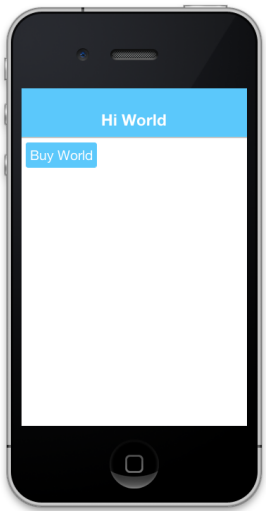


Figure 299. in-app purchase demo app

In the simulator, clicking on the "Buy World" button will bring up a prompt to ask you if you want to approve the purchase.

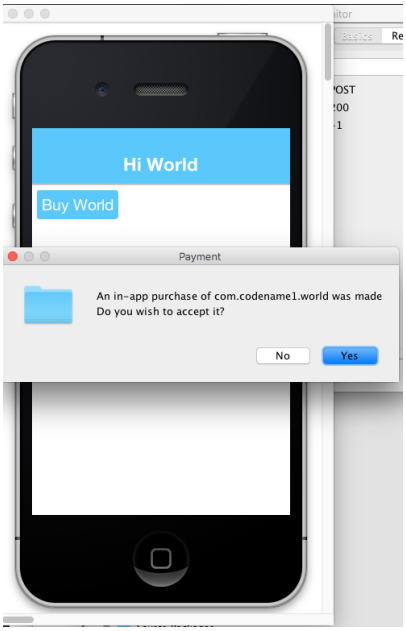


Figure 300. Approving the purchase in the simulator

Now if you try to buy the product again, it pops up the dialog to let you know that you already own it.

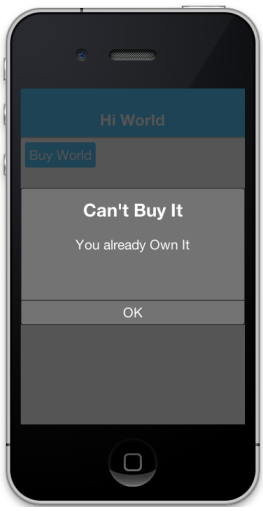


Figure 301. In App buy already owned

58.1.4. Making it consumable

In the "Buy World" example above, the "world" product was non-consumable, since you could only buy the world once. You could change it to a consumable product by disregarding whether it was purchased before & keeping track of how many times it had been purchased.

You'll use storage to keep track of the number of worlds that the user purchased. You need two methods to manage this count. One method gets the number of worlds that you own, and another adds a world to this count:

```
private static final String NUM_WORLDS_KEY = "NUM_WORLDS.dat";
public int getNumWorlds() {
 synchronized (NUM_WORLDS_KEY) {
 Storage s = Storage.getInstance();
```

```
 if (s.exists(NUM_WORLDS_KEY)) {
 return (Integer)s.readObject(NUM_WORLDS_KEY);
 } else {
 return 0;
 }
 }
}

public void addWorld() {
 synchronized (NUM_WORLDS_KEY) {
 Storage s = Storage.getInstance();
 int count = 0;
 if (s.exists(NUM_WORLDS_KEY)) {
 count = (Integer)s.readObject(NUM_WORLDS_KEY);
 }
 count++;
 s.writeObject(NUM_WORLDS_KEY, new Integer(count));
 }
}
```

Now you'll change your buy code as follows:

And your `itemPurchased()` callback will need to add a world:



When you set up the products in the iTunes store you will need to mark the product as a consumable product or iTunes will prevent you from purchasing it more than once

58.1.5. Non-Renewable subscriptions

As you discussed before, there are two types of subscriptions:

1. Non-renewable
2. Auto-renewable

Non-renewable subscriptions are the same as consumable products, except that they're shareable across devices. Auto-renewable subscriptions will continue as long as the user doesn't cancel the subscription. They will be re-billed automatically by the appropriate app-store when the chosen period expires, and the app-store handles the management details itself.



The concept of an "Non-renewable" subscription is unique to iTunes. Google Play has no formal similar option. To create a non-renewable subscription SKU that behaves the same in your iOS and Android apps you would create it as a **regular product** in Google play, and a Non-renewable subscription in the iTunes store. You'll learn more about that in a later post when you go into the specifics of app store setup.



The `Purchase` class includes both a `purchase()` method and a `subscribe()` method.

> On some platforms it makes no difference which one you use, but on Android it matters. If the product is set up as a subscription in Google Play, then you **must** use `subscribe()` to buy the product. If it's set up as a regular product, then you **must** use `purchase()`. Since you enter "Non-renewable" subscriptions as regular products in the play store, you would use the `purchase()` method.

58.1.6. Promotional offers (iOS)

Apple allows you to present discounted introductory pricing to existing subscribers via promotional offers [https://developer.apple.com/documentation/storekit/skpaymentdiscount]. Codename One surfaces this capability through overloads of both `Purchase.purchase(String, PromotionalOffer)` and `Purchase.subscribe(String, PromotionalOffer)`, which forward the promotional context to StoreKit when you start the transaction. Promotional offers are only honoured by iOS, so the overloads fall back to the regular buy flow on other platforms.

To build the signed discount payload required by Apple you can use the `ApplePromotionalOffer` helper:

Apple generates the signature and timestamp from your App Store Connect server notifications endpoint; Codename One passes them to the native StoreKit APIs. For one-time products you can call `purchase(sku, offer)` instead of `subscribe(…)`.

58.1.7. Restoring purchases and managing subscriptions

Both Apple and Google provide built-in user interfaces for restoring past purchases and managing subscription billing preferences. Codename One exposes these entry points so you can surface the native flows without reimplementing them yourself.

- Restores: Call `Purchase.isRestoreSupported()` before presenting a "Restore purchases" button. When supported (iOS implements this natively), invoke `Purchase.restore()` to prompt the operating system to re-deliver past transactions. Your app should implement `RestoreCallback` (like how you implement `PurchaseCallback`) so you can respond to individual `itemRestored(…)` events and to the completion or failure of the restore request.
- Subscription management: Use `Purchase.isManageSubscriptionsSupported()` to detect whether the platform can show the subscription management UI. When it returns `true`, calling `Purchase.manageSubscriptions(null)` opens the store-specific settings screen (Apple's subscription center on iOS and Google Play's subscription management activity on Android). On Android you can optionally pass a SKU to deep-link directly to the plan the user should manage.

Because these flows are handled by the underlying store your UI doesn't need to rebuild any billing screens. Gate the buttons on the capability checks above so that iOS and Android users get the familiar restore/manage dialogs while other platforms can fall back to your own help copy.

58.1.8. The Server-Side

Since a subscription purchased on one user device **needs** to be available across the user's devices (Apple's rules for non-renewable subscriptions), your app will need to have a server-component. In this section, you'll gloss over that & "mock" the server interface. You'll go into the specifics of the

server-side below.

The receipts API

Subscriptions, in Codename One use the "Receipts" API. It's up to you to register a receipt store with the in-app purchase instance, which allows Codename one to load receipts (from your server), and submit new receipts to your server. A `Receipt` includes information such as:

1. Store code (since you may be dealing with receipts from itunes, google play & Microsoft)
2. SKU
3. Transaction ID (store specific)
4. Expiry Date
5. Cancellation date
6. Buy date
7. Order Data (that you can use on the server-side to verify the receipt and load receipt details directly from the store it originated from).

The `Purchase` provides a set of methods for interacting with the receipt store, such as:

1. `isSubscribed([skus])` - Checks to see if the user is subscribed to any of the provided skus.
2. `getExpiryDate([skus])` - Checks the expiry date for a set of skus.
3. `synchronizeReceipts()` - Synchronizes the receipts with the receipt store. This will try to submit any pending purchase receipts to the receipt store, and the reload receipts from the receipt store.

In order for any of this to work, you must implement the `ReceiptStore` interface, and register it with the Buy instance. Your receipt store must implement two methods:

1. `fetchReceipts(SuccessCallback<Receipt[]> callback)` - Loads all the receipts from your receipt store for the current user.
2. `submitReceipt(Receipt receipt, SuccessCallback<Boolean> callback)` - Submits a receipt to your receipt store. This gives you an opportunity to add details to the receipt such as an expiry date.

58.1.9. The "hello world" of Non-Renewable subscriptions

You'll expand on the theme of "Buying" the world for this app, except, this time you will "Rent" the world for a period of time. You'll have two products:

1. A 1-month subscription
2. A 1-year subscription

Notice that you create two separate SKUs for the 1 month and 1-year subscription. **Each subscription period must have its own SKU**. The example uses an array (`PRODUCTS`) that contains both of the SKUs. This is handy, as you'll see in the examples ahead, because the APIs for checking status and expiry date of a subscription take the SKUs in a "subscription group" as input.



Different SKUs that sell the same service/product but for different periods form a "subscription group." Conceptually, customers aren't subscribing to a particular SKU, they're subscribing to the subscription group of which that SKU is a member. As an example, if a user purchases 1-month subscription to "the world," they're actually subscribing to "the world" subscription group.

It's up to you to know the grouping of your SKUs. Any methods in the `Purchase` class that check subscription status or expiry date of a SKU should be passed **all** SKUs of that subscription group. For example, If you want to know if the user is subscribed to the `SKU_WORLD_1_MONTH` subscription, it would not be enough to call `iap.isSubscribed(SKU_WORLD_1_MONTH)`, because that wouldn't consider if the user had purchased 1-year subscription. The correct way is to always call `iap.isSubscribed(SKU_WORLD_1_MONTH, SKU_WORLD_1_YEAR)`, or `iap.isSubscribed(PRODUCTS)` since you have placed both SKUs into your PRODUCTS array.

Implementing the receipt store



The receipt store is intended to interface with a server so that the subscriptions can be synced with multiple devices, as required by Apple's guidelines. For this post you'll just store your receipts on device using internal storage. Moving the logic to a server is a simple matter that this guide covers in a future post when you cover the server-side.

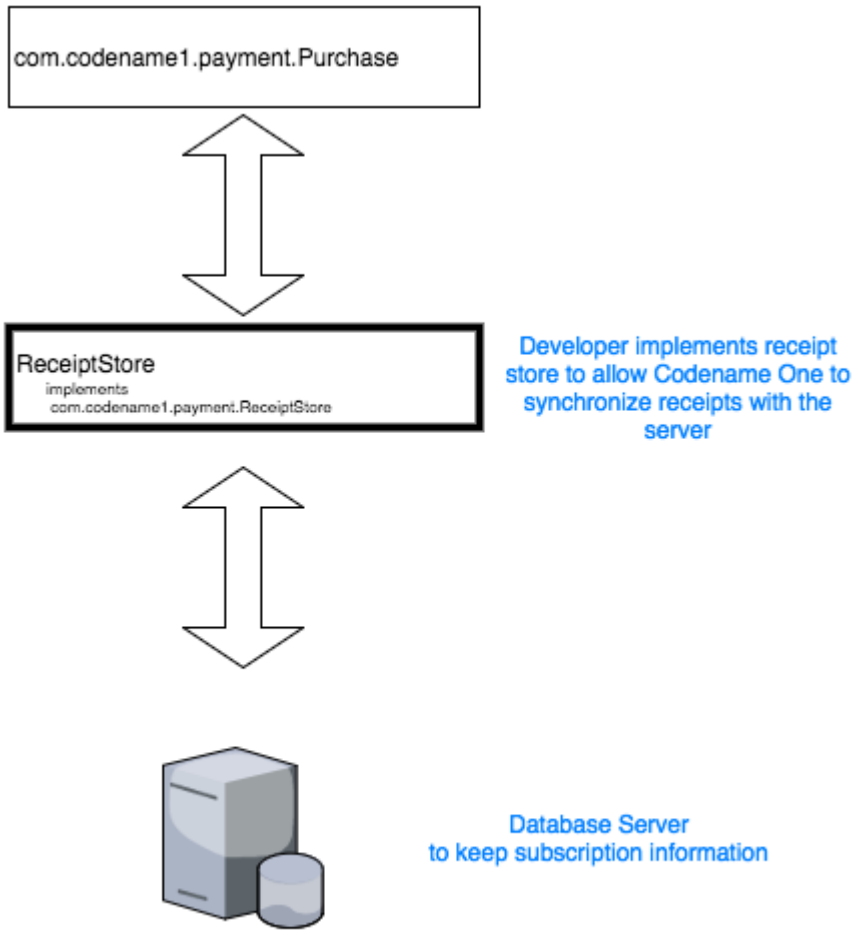


Figure 302. The Receipt store is a layer between your server and Codename One

A basic receipt store needs to implement just two methods:

1. `fetchReceipts`
2. `submitReceipt`

You'll register it in your app's init() method so that it's always available:

These methods are designed to be asynchronous since real-world apps will always be connecting to some sort of network service. Instead of returning a value, both of these methods are passed instances of the `SuccessCallback` class. It's important to make sure to call `callback.onSuccess()` **ALWAYS** when the methods have completed, even if there is an error, or the Buy class will just assume that you're taking a long time to complete the task, and will continue to wait for you to finish.

Once implemented, your `fetchReceipts()` method will look like:

This is straight forward. You're checking to see if you already have a list of receipts stored. If so you return that list to the callback. If not you return an empty array of receipts.



`Receipt` implements `Externalizable` so you are able to write instances directly to Storage.

The `submitReceipt()` method is a little more complex, as it needs to calculate the new expiry date for your subscription:

The main logic of this method involves iterating through all the existing receipts to find the **latest** current expiry date, so that when the user purchases a subscription, it's added onto the end of the current subscription (if one exists) rather than going from today's date. This enables users to renew their subscription before the subscription has expired.

In the real-world, you would implement this logic on the server-side.



The iTunes store and Play store have no knowledge of your subscription durations. This is why it's up to you to set the expiry date in the `submitReceipt` method. Non-renewable subscriptions are essentially no different than regular consumable products. It's up to you to manage the subscription logic - and Apple, in particular, requires you to do so using a server.

Synchronizing receipts

In order for your app to provide you with current data about the user's subscriptions and expiry dates, you need to synchronize the receipts with your receipt store. `Purchase` provides a set of methods for doing this. You'll call one of them inside the `start()` method, and you may resynchronize at other strategic times if you suspect that the information may have changed.

The following methods can be used for synchronization:

1. `synchronizeReceipts()` - Asynchronously synchronizes receipts in the background. You won't be notified when it's complete.

2. `synchronizeReceipts(final long ifOlderThanMs, final SuccessCallback<Boolean> callback)` - Asynchronously synchronizes receipts, but only if they haven't been synchronized in the specified time. For example, In your start() method you might decide that you only want to synchronize receipts once per day. This also includes a callback that will be called when synchronization is complete.
3. `synchronizeReceiptsSync(long ifOlderThanMs)` - Synchronously synchronizes receipts, blocking until it's complete, and will only refetch if data is older than the given time (pass `0` to always refetch). This is safe to use on the EDT as it employs `invokeAndBlock` under the covers.

In your hello world app you synchronize the subscriptions in a few places.

At the end of the `start()` method:

And you also provide a button to allow the user to manually synchronize the receipts:

Expiry dates and subscription status

Now that you have a receipt store registered, and you have synchronized your receipts, you can query the `Purchase` instance to see if a SKU or set of SKUs is subscribed. Three useful methods in this realm:

1. `boolean isSubscribed(String… skus)` - Checks to see if the user is subscribed to any of the provided SKUs.
2. `Date getExpiryDate(String… skus)` - Gets the latest expiry date of a set of SKUs.
3. `Receipt getFirstReceiptExpiringAfter(Date dt, String… skus)` - This method will return the earliest receipt with an expiry date after the given date. This is needed in cases where you need to decide if the user should have access to some content based on its publication date. For example, If you published an issue of your e-zine on March 1, and the user purchased a subscription on March 15th, then they should get access to the March 1st issue even though it doesn't necessarily fall in the subscription period. Being able to fetch the first receipt after a given date makes it easier to determine if a particular issue should be covered by a subscription.

If you need to know more information about subscriptions, you can always just call `getReceipts()` to get a list of all the current receipts and determine for yourself what the user should have access to.

In the hello world app you'll use this information in a few different places. On your main form you'll include a label to show the current expiry date, and you allow the user to press a button to synchronize receipts manually if they think the value is out of date:

Allowing the user to purchase the subscription

You should now have all the background required to implement the Hello World Subscription app. You'll return to the code and see how the user purchases a subscription.

In the main form, you want two buttons to subscribe to the `World`, for one month and one year respectively. They look like:

58.1.10. Subscribe() vs purchase()

The `Purchase` class includes two methods for initiating a purchase:

1. `purchase(sku)`
2. `subscribe(sku)`

Which one you use depends on the type of product that's being purchased. If your product is set up as a subscription in the Google Play Store, then you should use `subscribe(sku)`. Otherwise, you should use `purchase(sku)`.

Handling purchase callbacks

The buy callbacks are like the ones implemented in the regular in-app purchase examples:

Notice that, in `itemPurchased()` you don't need to explicitly create any receipts or submit anything to the receipt store. This is handled for you automatically. You do make a call to `synchronizeReceiptsSync(0)` but this is just to ensure that your toast message has the new expiry date loaded already.

58.1.11. Screenshots

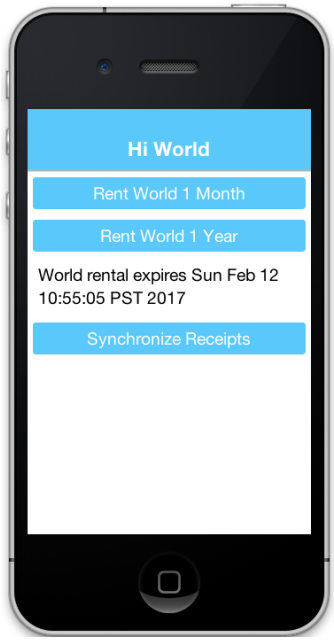


Figure 303. Main form

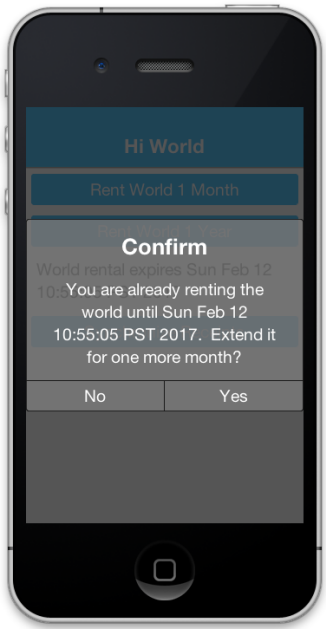


Figure 304. Dialog shown when subscribing to a product

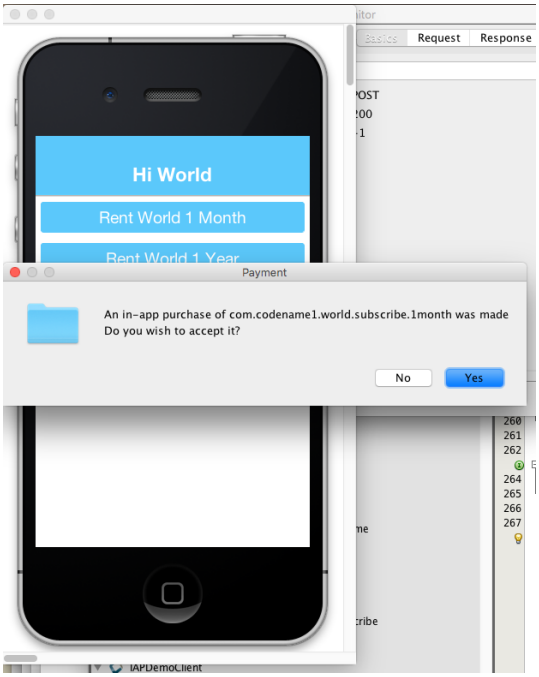


Figure 305. Simulator confirm dialog when purchasing a subscription

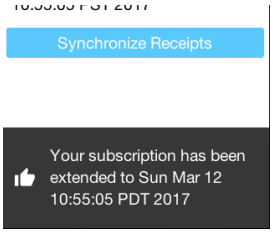


Figure 306. Upon successful buy, the toastbar message is shown

58.1.12. Summary

This section demonstrated how to set up an app to use non-renewable subscriptions using in-app purchase. Non-renewable subscriptions are the same as regular consumable products except for the fact that they're shared by all the user's devices, and thus, require a server component. The app

store has no knowledge of the duration of your non-renewable subscriptions. It's up to you to specify the expiry date of purchased subscriptions on their receipts when they're submitted. Google play doesn't formally have a "non-renewable" subscription product type. To implement them in Google play, you would just set up a regular product. It's how you handle it internally that makes it a subscription, and not just a regular product.

Codename One uses the `Receipt` class as the foundation for its subscriptions infrastructure. You, as the developer, are responsible for implementing the `ReceiptStore` interface to provide the receipts. The `Purchase` instance will load receipts from your ReceiptStore, and use them to determine whether the user is subscribed to a subscription, and when the subscription expires.

58.1.13. Auto-Renewable subscriptions

Auto-renewable subscriptions provide, arguably, an easier path to recurring revenue than non-renewable subscriptions because all the subscription stuff is handled by the app store. You defer almost entirely to the app store (iTunes for iOS, and Play for Android) for billing and management.

If there is a down-side, it would be that you are also subject to the rules of each app store - and they take their cut of the revenue.

1. For more information about Apple's auto-renewable subscription features and rules see this document [https://developer.apple.com/app-store/subscriptions/].
2. For more information about subscriptions in Google play, see this document [https://developer.android.com/google/play/billing/billing_subscriptions.html].

58.1.14. Auto-Renewable vs Non-Renewable: Choosing between them

When deciding between auto-renewable and non-renewable subscriptions, as always, the answer will depend on your needs and preferences. Auto-renewables are nice because it takes the process out of your hands. You just get paid. On the other hand, there are valid reasons to want to use non-renewables. For example, You can't cancel an auto-renewable subscription for a user. They have to do that themselves. You may also want more control over the subscription and renewal process, in which case a non-renewable might make more sense.



Some developers are opposed to auto-renewable subscriptions [https://marco.org/2013/12/02/auto-renewable-subscriptions]. There isn't enough information to make a solid recommendation on this matter.

On a practical level, if you are using auto-renewable subscriptions (and so subscription products in the Google Play Store) you must use the `Purchase.subscribe(sku)` method for initiating the purchase workflow. For non-renewable subscriptions (and so regular products in the Google Play Store), you must use the `Purchase.purchase(sku)` method.

58.1.15. Learning by example

In this section you'll describe the general workflow of subscription management on the server. You also show how use Apple's and Google's web services to check receipts and stay informed of important events (such as when users cancel or renew their subscriptions).

58.1.16. Building the IAP demo project

To aid in this process, Codename One provides a fully functional in-app purchase demo project that includes both a client app [https://gist.github.com/shannah/b61b9b6b35ea0eac923a54163f5d4deb] and a server app [https://github.com/shannah/cn1-iap-demo-server].

Setting up the client project

1. Create a new Codename One project in Netbeans, and choose the `Bare-bones Hello World Template`. You should make your package name something unique so that you are able to create real corresponding apps in both Google Play and iTunes connect.
2. Once the project is created, copy this source file [https://gist.github.com/shannah/b61b9b6b35ea0eac923a54163f5d4deb] contents into your main class file. Then change the package name, and class name in the file to match your project settings. For example, change `package ca.weblite.iapdemo;` to `package <your.package.name.here>;` and `class IAPDemo implements PurchaseCallback` to `class YourClassName implements PurchaseCallback`.
3. Add the Generic Web Service Client [https://github.com/shannah/cn1-generic-webservice-client] library to your project by going to "Codename Settings" > `Extensions`, finding that library, and click `Download`. Then "Refresh CN1 libs" as it suggests.
4. Change the `localHost` property to point to your local machine's network address. Using "http://localhost" isn't going to cut it here because when the app is running on a phone, it needs to be able to connect to your web server over the network. This address will be your local network address (for example, 192.168.0.9, or something like that).

   ```
   private static final String localHost = "http://10.0.1.32";
   ```

5. Add the `ios.plistInject` build hint to your project with the value `<key>NSAppTransportSecurity</key> <dict> <key>NSAllowsArbitraryLoads</key> <true/> </dict>`. This is so that you can use http urls in iOS. Since you don't intend to fully publish this app, you can cut corners like this. If you were creating a real app, you would use proper secure URLs.

Setting up the server project

Download the CN1-IAP-Server demo project from GitHub, and run its "install-deps" ANT task to download and install its dependencies to your local Maven repo.



For the following commands to work, make sure you have "ant," "mvn," and "git" in your environment PATH.

```
$ git clone https://github.com/shannah/cn1-iap-demo-server
$ cd cn1-iap-demo-server
$ ant install-deps
```

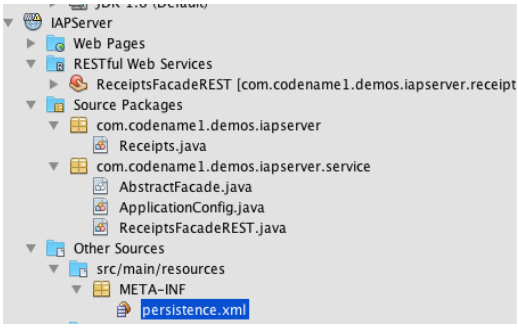
Open the project in Netbeans

Setting up the database

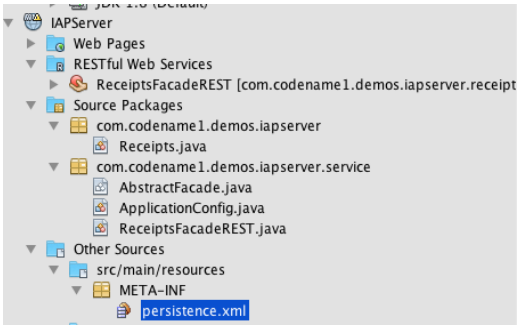
1. Create a new database in your preferred DBMS. Call it anything you like.
2. Create a new table named "RECEIPTS" in this database with the following structure:

```
create TABLE RECEIPTS
(
    TRANSACTION_ID VARCHAR(128) not null,
    USERNAME VARCHAR(64) not null,
    SKU VARCHAR(128) not null,
    ORDER_DATA VARCHAR(32000),
    PURCHASE_DATE BIGINT,
    EXPIRY_DATE BIGINT,
    CANCELLATION_DATE BIGINT,
    LAST_VALIDATED BIGINT,
    STORE_CODE VARCHAR(20) default '' not null,
    primary key (TRANSACTION_ID, STORE_CODE)
)
```

3. Open the "persistence.xml" file in the server NetBeans project.



4. Change the data source to the database you just created.



If you're not sure how to create a data source, see your previous tutorial on connecting to a MySQL database [https://www.codenameone.com/blog/connecting-to-a-mysql-database-part-2.html].

Testing the project

At this point you should be able to test out the project in the Codename One simulator to make sure it's working.

1. Build and Run the server project in Netbeans. You may need to tell it which application server

you wish to run it on. This example runs on the Glassfish 4.1 that comes bundled with Netbeans.

2. Build and run the client project in Netbeans. This should open the Codename One simulator.

When the app first opens you'll see a screen as follows:

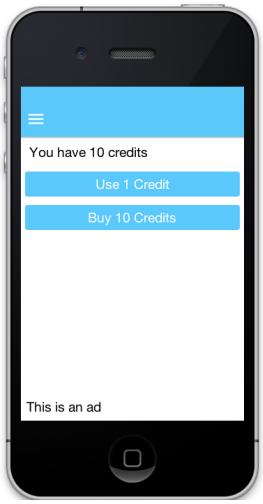


Figure 307. First screen of app

This screen is for testing consumable products, so you won't be making use of this right now.

Open the hamburger menu and select `Subscriptions`. You should see something like this:

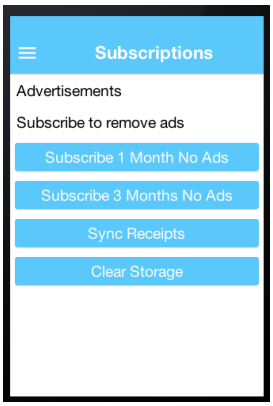


Figure 308. Subscriptions form

Click on the "Subscribe 1 Month No Ads" button. You will be prompted to accept the purchase:

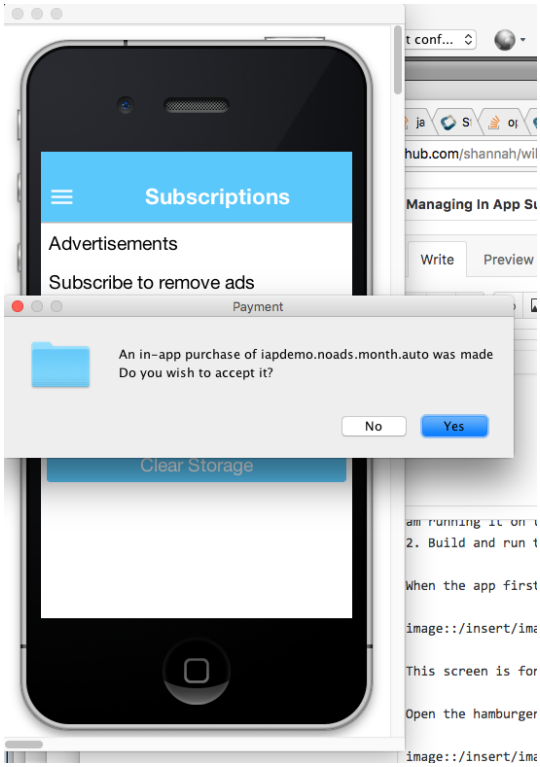


Figure 309. Approve buy dialog

Upon completion, the app will submit the purchase to your server, and if all went well, it will retrieve the updated list of receipts from your server also, and update the label on this form to say "No Ads. Expires <some date>":

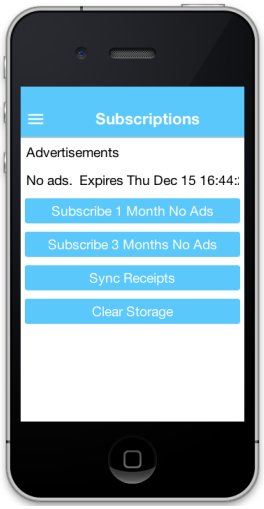


Figure 310. After successful buy



This project is set up to use an expedited expiry date schedule for purchases from the simulator. 1 month = 5 minutes. 3 months = 15 minutes. This helps for testing. That's why your expiry date may be different than expected.

Just to verify that the receipt was inserted, you should check the contents of your "RECEIPTS" table in your database. In Netbeans, you can do this from the "Services" pane. Expand the database connection down to the RECEIPTS table, right click "RECEIPTS" and select `View Data`. This will open a data table similar the following:

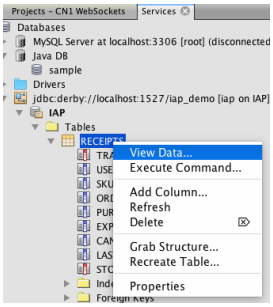


Figure 311. Receipts table after insertion

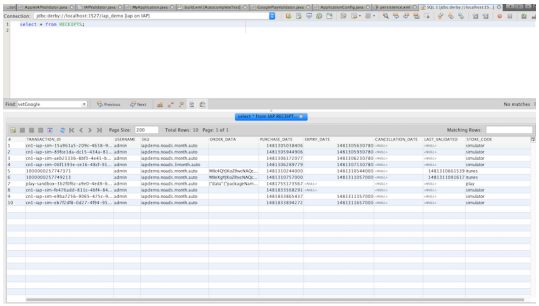


Figure 312. Table view

A few things to mention here:

1. The "username" was provided by the client. It's hard-coded to "admin," but the idea is that you would have the user log in and you would have access to their real username.
2. All dates are stored as unix timestamps in milliseconds.

If you delete the receipt from your database, then press the "Synchronize Receipts" button in your app, the app will again say "No subscriptions." Similarly if you wait 5 minutes and hit "Synchronize receipts" the app will say no subscriptions found, and the "ads" will be back.

Troubleshooting

Don't pretend that everything worked on the first try. There's a lot that could go wrong here. If you make a purchase and nothing appears to happen, the first thing you should do is check the Network Monitor in the simulator ("Simulate" > "Network" > "Network Monitor"). You should see a list of network requests. Some will be GET requests and there will be at least one POST request. Check the response of these requests to see if they succeeded.

Also check the Glassfish server log to see if there is an exception.

Common problems would be that the URL you have set in the client app for `endpointURL` is wrong, or that there is a database connection problem.

58.1.17. Looking at the source of the app

Now that you've set up and built the app, take a look at the source code so you can see how it all works.

Client side

The example uses the Generic Webservice Client Library [https://github.com/shannah/cn1-generic-

webservice-client] from inside your `ReceiptStore` implementation to load receipts from the web service, and insert new receipts to the database.

The source for your ReceiptStore is as follows:

Notice that you aren't doing any calculation of expiry dates in your client app, as you did in the previous post (on non-renewable receipts). Since you are using a server now, it makes sense to move all that logic over to the server.

The `createRESTClient()` method shown there creates a `RESTfulWebServiceClient` and configuring it to use basic authentication with a username and password. The idea is that your user would have logged into your app at some point, and you would have a username and password on hand to pass back to the web service with the receipt data so that you can connect the subscription to a user account. The source of that method is listed here:

Server-Side

On the server-side, your REST controller is a standard JAX-RS REST interface. The Netbeans web service wizard generated it and then it was modified to suit the purposes here. The methods of the `ReceiptsFacadeREST` class for the REST API are shown here:

The magic happens inside that `validateAndSaveReceipt()` method, which You'll cover in detail soon.

Notifications

It's important to note that you won't be notified by apple or google when changes are made to subscriptions. It's up to you to periodically "poll" their web service to find if any changes have been made. Changes you would be interested in are primarily renewals and cancellations. Set up a method to run periodically (once per day might be enough), using the scheduling mechanism on your server stack.

That method finds all the receipts in the database that haven't been validated in some period of time, and validates it. Again, the magic happens inside the `validateAndSaveReceipt()` method which you cover later.



This example only checks receipts from the iTunes and Play stores because those are the only stores that support auto-renewing subscriptions in Codename One.

58.1.18. The CN1-IAP-Validator library

For this tutorial, the example uses a purpose-built library to handle receipt validation in a way that hides as much of the complexity as possible. It supports both Google Play receipts and iTunes receipts.

The general usage is as follows:

As you can see from this snippet, the complexity of receipt validation has been reduced to entering three configuration strings:

1. `APPLE_SECRET` - This is a "secret" string that you will get from iTunes connect when you set up

your in-app products.

2. `GOOGLE_DEVELOPER_API_CLIENT_ID` - A client ID that you'll get from the Google developer API console when you set up your API service credentials.
3. `GOOGLE_DEVELOPER_PRIVATE_KEY` - A PKCS8 encoded string with an RSA private key that you'll receive at the same time as the `GOOGLE_DEVELOPER_API_CLIENT_ID`.

The next section walks through the steps to get these values.

58.1.19. The `validateAndSaveReceipt()` method

You are now ready to see the full magic of the `validateAndSaveReceipt()` method in all its glory:



In many of the code snippets for the Server-side code, you'll see references to both a `Receipts` class and a `Receipt` class. This is slightly confusing. The `Receipts` class is a JPA entity the encapsulates a row from the "receipts" table of your SQL database. The `Receipt` class is `com.codename1.payment.Receipt`. It's used to interface with the IAP validation library.

58.1.20. Google play setup

Creating the app in Google play

To test out in-app purchase on an Android device, you'll need to create an app the Google Play Developer Console [https://play.google.com/apps/publish/]. This section doesn't describe the process, but plenty of information around the internet covers how to do this. Some useful references for this include:

1. Getting Started With Publishing [https://developer.android.com/distribute/googleplay/start.html] - If you don't already have an account with Google to publish your apps.
2. Launch Checklist [https://developer.android.com/distribute/tools/launch-checklist.html]

Graphics, icons, etc

You are required to upload some screenshots and feature graphics. Don't waste time making these perfect. For the screenshots, you can just use the "Screenshot" option in the simulator. (Use the Nexus 5 skin). For the feature graphics, this example uses this site [https://www.norio.be/android-feature-graphic-generator/] that will generate the graphics in the correct dimensions for Google Play. You can also just leave the icon as the default Codename One icon.

Creating test accounts



You can't buy in-app products from your app using your publisher account. You need to set up at least one test account for testing the app.

To test your app, you need to set up a test account. A test account must be associated with a real gmail email address. If you have a domain that's managed by Google apps, then you can also use an address from that domain.

The full process for testing in-app billing can be found in this Google document [https://developer.android.com/google/play/billing/billing_testing.html]; the documentation is dense and can be difficult to follow.

For your purposes, you'll need to set up a tester list in Google Play. Choose "Settings" > `Tester Lists`. Then create a list with all the email address that you want to have treated as test accounts. Any purchases made by these email addresses will be treated as "Sandbox" purchases, and won't require real money to change hands.

Alpha channel distribution

To test in-app purchase on Android, you **must** first publish your app. You can't just build and install your app manually. The app needs to be published on the Play store, and it must be installed **through** the play store for in-app purchase to work. You can publish to an Alpha channel so that your app won't be publicly available.

For more information about setting up alpha testing on Google play see this Google support document on the subject [https://support.google.com/googleplay/android-developer/answer/3131213?hl=en].

Once you have set your app up for alpha testing, you can send an invitation link to your test accounts. You can find the link in the Google Play console under the APK section, under the "Alpha" tab (and assuming you've enabled alpha testing.

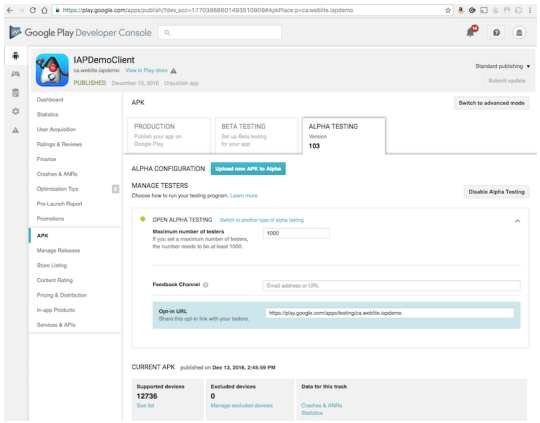


Figure 313. Alpha testing tab in google play

The format of the link is `https://play.google.com/apps/testing/your-app-id` in case you can't find it. You can email this to your alpha testers. Make sure that you have added all testers to your tester lists so that their purchases will be made in the sandbox environment.

Also, before proceeding with testing in-app purchases, you need to add the in-app products in Google Play.

Adding In-App products

After you have published your APK to the alpha channel, you can create the products. For the purposes of this tutorial, you'll just add two products:

1. **iapdemo.noads.month.auto** - The 1-month subscription.
2. **iapdemo.noads.3month.auto** - The 3-month subscription.



Since you will be adding products as "Subscriptions" in the pay store, your app **must** use the `Purchase.subscribe(sku)` method for initiating a purchase on these products, and **not** the `Purchase.purchase(sku)` method. If you use `purchase()` to buy a subscription on Android, the payment will go through, but your buy callback will receive an error.

Adding 1 month Subscription

1. Open Google Play Developer Console, and navigate to your app.
2. Click on "In-app Products" in the menu. Then click the "Add New Product" button.
3. Select `Subscription`, and enter "iapdemo.noads.month.auto" for the Product ID. Then click "Continue"

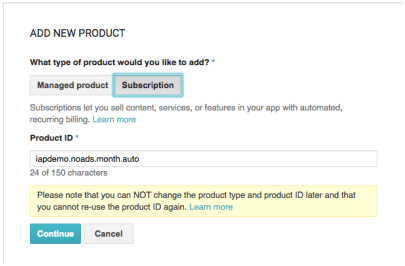


Figure 314. Add new product dialog

Now fill in the form. You can choose your own price and name for the product. The following is a screenshot of the options selected for this example.

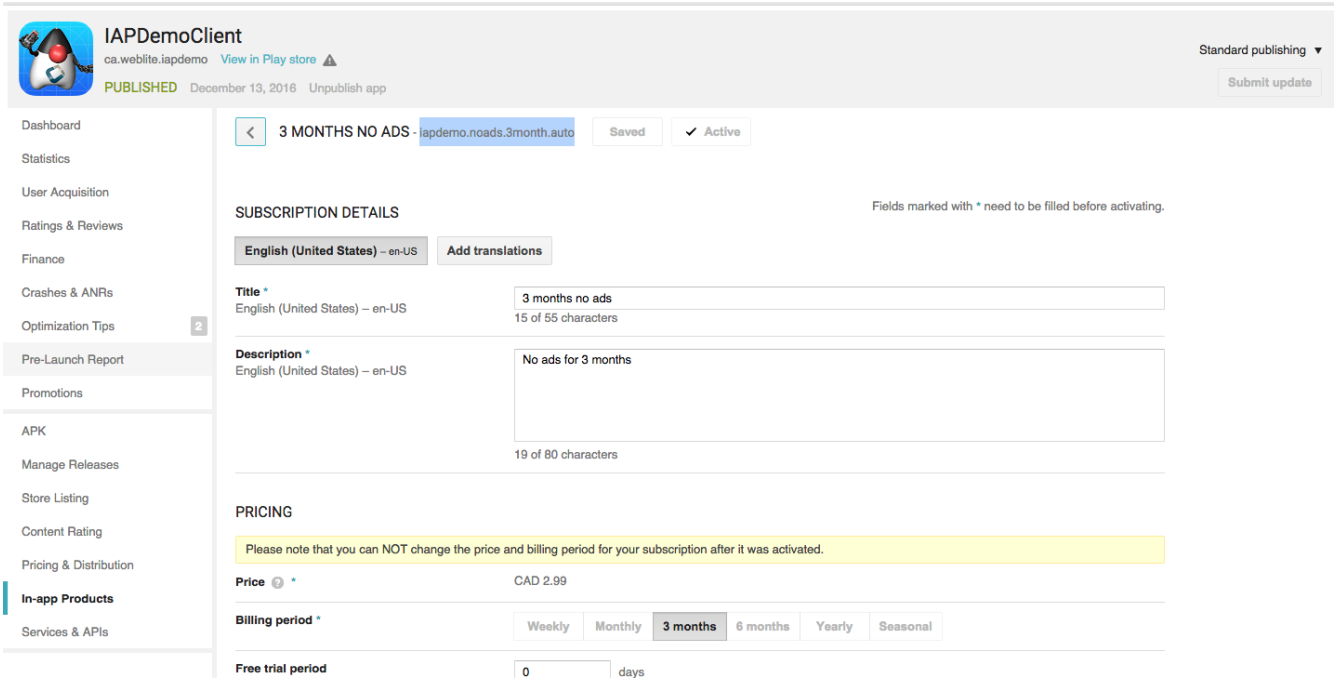


Figure 315. Add product to google

Adding 3 month Subscription

Follow the same process as for the 1-month subscription except use "iapdemo.noads.3month.auto" for the product ID, and select "3 months" for the billing period instead of `Monthly`.

Testing the app

At this point you should be ready to test your app. Assuming you've installed the app using the invite link you sent yourself from Google play, **as a test account that's listed on your testers list**, you should be good to go.

Open the app, click on `Subscriptions`, and try to buy a 1-month subscription. If all goes well, it should insert the subscription into your database. But with no expiry date, since you haven't yet implemented receipt validation yet. You'll do that next.

Creating Google play receipt validation credentials

Google play receipt validation is accomplished via the android-publisher Purchases: get API [https://developers.google.com/android-publisher/api-ref/purchases/subscriptions/get]. The CN1-IAP-Validation library shields you from most of the complexities of using this API, but you still need to get a "private key" and a "client ID" to access this API. Both of these are provided when you set up an OAuth2 Service Account [https://developers.google.com/identity/protocols/OAuth2ServiceAccount] for your app.



The following steps assume that you have already created your app in Google play and have published it to at least the alpha channel. See your previous post on this topic here (Link to be provided).

Steps:

1. Open the Google API Developer Console [https://console.developers.google.com/apis], and select your App from the menu.
2. Click on the "Library" menu item in the left menu, and then click the "Google Play Developer API" link.

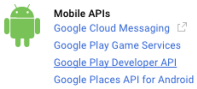


Figure 316. Google Play Developer API Link

3. Click on the button that says `Enable`. (If you already have it enabled, then just proceed to the next step).



Figure 317. Enable API button

4. Click on the "Credentials" menu item in the left menu.
5. In the "Credentials" drop-down menu, select the "Service Account Key" option.

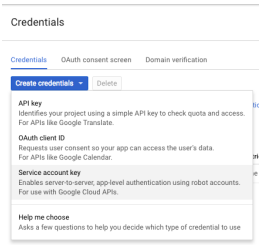


Figure 318. Credentials dropdown

6. You will be presented with a new form. In the "Service Account" drop-down, select `New Service Account`. This will give you some more options.

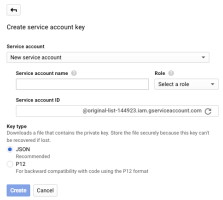


Figure 319. Create service account key

7. Enter anything you like for the `Service account name`. For the role, you'll select "Project" > "Owner" for now just so you don't run into permissions issues. You'll probably want to investigate further to fine a more limited role that only allows receipt verification, but for now, avoid any unnecessary road blocks for getting this to work. You're probably going to run into "permission denied" errors at first anyway, so the fewer reasons for this, the better.
8. It will autogenerate an account ID for you. 9., for the `Key type`, select `JSON`. Then click the "Create" button.

This should prompt the download of a JSON file that will have contents like the following:

```
{
 "type": "service_account",
 "project_id": "iapdemo-152500",
 "private_key_id": "1b1d39************7d839826b8a",
 "private_key": "-----BEGIN PRIVATE KEY-----... Some private key string -----END PRIVATE KEY-----\n",
 "client_email": "iapdemo@iapdemo-152500.iam.gserviceaccount.com",
 "client_id": "117601572633333082772",
 "auth_uri": "https://accounts.google.com/o/oauth2/auth",
 "token_uri": "https://accounts.google.com/o/oauth2/token",
 "auth_provider_x509_cert_url": "https://www.googleapis.com/oauth2/v1/certs",
 "client_x509_cert_url": "https://www.googleapis.com/robot/v1/metadata/x509/iapdemo%40iapdemo-
152500.iam.gserviceaccount.com"
}
```

This is where you get the information you're looking for. The "client_email" is what you'll use for your `googleClientId`, and the "private_key" is what you'll use for the `googlePrivateKey`.



Use the "client_email" value as your client ID, not the "client_id" value as you might be tempted to do.

You'll set these in your constants:

NOT DONE YET

Before you can use these credentials to verify receipts for your app, you need to link your app to this new service account from within Google Play.

Steps:

1. Open the Google Play Developer Console [https://play.google.com/apps/publish/], then click on "Settings" > `API Access`.
2. You should see your app listed on this page. Click the "Link" button next to your app.



Figure 320. Link to API

3. This should reveal some more options on the page. You should see a "Service Accounts" section with a list of all the service accounts that you have created. Find the one you just created, and click the "Grant Access" button in its row.

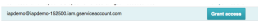


Figure 321. Grant access

4. This will open a dialog titled `Add New User`. Leave everything default, except change the "Role" to `Administrator`. This provides "ALL" permissions to this account, which probably isn't a good idea for production. Later on, after everything is working, you can circle back and try to refine permissions. For this tutorial, the goal is to remove all potential road blocks.

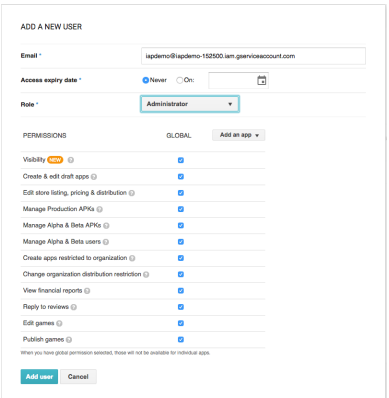


Figure 322. New User

5. Press the "Add User" button.

At this point, the service account **should** be active so you can try to check receipts.

Testing receipt validation

The `ReceiptsFacadeREST` class includes a flag to enable/disable play store validation. By default it's disabled. Enable it:

```
public static final boolean DISABLE_PLAY_STORE_VALIDATION=true;
```

Change this to `false`.

Then build and run the server app. The `validateSubscriptionsCron()` method is set to run once per minute, so you just need to wait for the timer to come up and it should try to check all the play store receipts.



You're assuming you've already added a receipt in the previous test that you did. If necessary, you should buy the subscription again in your app.

After a minute or so, you should see "---------- VALIDATING RECEIPTS ---------" written in the Glassfish log, and it will check your receipts. If it works, your receipt's expiry date will get populated in the database, and you can press "Synchronize Receipts" in your app to see this reflected. If it fails, there will likely be a big ugly stack trace and exception readout with some clues about what went wrong.

Realistically, your first try will fail for some reason. Use the error codes and stack traces to help lead you to the problem. And feel free to post questions here.

58.1.21. iTunes connect setup

The process for setting up and testing your app on iOS is much simpler than on Android (IMHO). It took you a couple of hours to get the iTunes version working, vs a couple of days on the Google Play side of things. One notable difference that makes things simpler is that you don't need to actually upload your app to the store to test in-app purchase. You can just use your debug build on your device. It's also **much** easier to roll a bunch of test accounts than on Google Play. You don't need to set up an alpha program, you just create a few "test accounts" (and this is easy to do) in your iTunes connect account, and then make sure to use one of these accounts when making a purchase. You can switch accounts on your device from the "Settings" app, where you can just log out of the iTunes store - which will cause you to be prompted in your app the next time you make a purchase.

Setting up In-App products

The process to add products in iTunes connect is outlined in this Apple developer document [https://developer.apple.com/library/content/documentation/LanguagesUtilities/Conceptual/iTunesConnectInAppPurchase_Guide/Chapters/CreatingInAppPurchaseProducts.html#//apple_ref/doc/uid/TP40013727-CH3-SW1]. You'll add your two SKUs:

1. **iapdemo.noads.month.auto** - The 1-month subscription.
2. **iapdemo.noads.3month.auto** - The 3-month subscription.

Just make sure you add them as auto-renewable subscriptions, and that you specify the appropriate renewal periods. Use the SKU as the product ID. Both of these products will be added to the same subscription group. Call the group whatever you like.

Creating test accounts

To test purchases, you need to create some test accounts. See this apple document [https://developer.apple.com/library/content/documentation/LanguagesUtilities/Conceptual/iTunesConnect_Guide/Chapters/SettingUpUserAccounts.html#//apple_ref/doc/uid/TP40011225-CH25-SW10] for details on how to create these test accounts. Don't worry, the process is much simpler than for Android. It should take you under 5 minutes.

Once you have the test accounts created, you should be set to test the app.

1. Make sure your server is running.
2. Log out from the app store. The process is described here [https://support.apple.com/en-ca/HT203983].
3. Open your app.
4. Try to buy a 1-month subscription

If all went well, you should see the receipt listed in the RECEIPTS table of your database. But the expiry date will be null. You need to set up receipt verification in order for this to work.

Setting up receipt verification

In order for receipt verification to work you need to generate a shared secret in iTunes connect. The process is described here [https://developer.apple.com/library/content/documentation/LanguagesUtilities/Conceptual/iTunesConnectInAppPurchase_Guide/Chapters/CreatingInAppPurchaseProducts.html].

Once you have a shared secret, update the ReceiptsFacadeREST class with the value:

```
public static final String APPLE_SECRET = "your-shared-secret-here";
```

And enable iTunes store validation:

```
public static final boolean DISABLE_ITUNES_STORE_VALIDATION=true;
```

Change this to `false`.

If you rebuild and run the server project, and wait for the `validateSubscriptionsCron()` method to run, it should check the receipt. After about a minute (or less), you'll see the text "---------- VALIDATING RECEIPTS ---------" written to the Glassfish log file, followed by some output from connecting to the iTunes validation service. If all went well, you should see your receipt expiration date updated in the database. If not, you'll likely see some exception stack traces in the Glassfish log.



Sandbox receipts in the iTunes store are set to run on an accelerated schedule. A 1-month subscription is actually 5 minutes, 3 months is 15 minutes etc. Also sandbox subscriptions don't seem to persist in perpetuity until the user has cancelled it. In practice they renew only 4 or 5 times before they're allowed to lapse by Apple.

Chapter 59. App Review & Feedback

Both Apple and Google provide an in-app review prompt that lets the user rate your app without leaving it. Asking for a review at the right moment is one of the most effective ways to improve your store rating. Codename One exposes this through the `com.codename1.appreview.AppReview` API, which uses the native review sheet (`SKStoreReviewController` on iOS, the Play In-App Review API on Android) when it's available and falls back to a Codename One drawn rating widget everywhere else (the simulator, desktop, the web target and older OS versions).

The native prompt is the store-sanctioned way to ask for a review, so prefer it over a hand-rolled dialog: the operating system controls how often it appears and throttles excessive prompts, which keeps your app within store policy.

59.1. Asking for a review

The simplest usage is a single call at a moment that makes sense in your app — typically right after the user completed something rewarding (finished a level, saved a document, completed an order):

`requestReview()` shows the native review sheet when the platform supports it. On platforms without a native prompt it shows the built-in rating widget instead. You decide the timing.

59.2. The engagement scheduler

Rather than picking the moment yourself, you can let `AppReview` decide based on simple engagement heuristics: how many times the app was launched, how long ago it was installed, and how long since it last asked. Configure it once (for example in your app's `init` or `start` method) and call `registerSession()` on every launch:

`registerSession()` records the launch and, once the thresholds are met and the user hasn't already rated or opted out, prompts. The bookkeeping (launch count, install date, last-prompt time and a completion flag) is stored in Preferences [https://www.codenameone.com/javadoc/com/codename1/io/Preferences.html] so it survives restarts. Once the user rates the app or chooses to stop, `AppReview` stops prompting. Call `reset()` to clear this state.



You can combine both styles — use `registerSession()` for the automatic cadence and still call `requestReview()` from a "Rate this app" menu item.

59.3. Smart feedback split

The fallback widget is presented as a bottom sheet, so the user can swipe it away instead of dismissing a blocking dialog. It asks for a star rating and then routes the user based on the result. A high rating (at or above the threshold set by `setHighRatingThreshold(int)`, four stars by default) sends the user to your store listing. A low rating opens a private feedback step instead, so unhappy users reach you directly rather than posting a one-star public review.



Figure 323. The Codename One fallback rating sheet, shown when no native review prompt is available

By default the low-rating feedback is collected through an e-mail to the address passed to `setSupportEmail(String)`. To deliver feedback through your own backend, register a `FeedbackListener`:

59.4. How the platform decides

`AppReview` checks `CN.isNativeInAppReviewSupported()` to choose between the native prompt and the fallback sheet. On Android the native path is active only when the app references the app-review API — the build adds the Play In-App Review library in that case, so apps that never ask for a review carry no extra dependency. On iOS the StoreKit review controller is linked the same way, only when the API is used.

The native prompts are drawn by the operating system, so this guide can't show them as Codename One screenshots. On iOS the prompt is the StoreKit rating panel: a star selector the system overlays on your app, submitted without leaving it. On Android it's the Play In-App Review card that slides up from the bottom. Both are styled and rate-limited by the OS — you can't theme them, force them to appear, or read the outcome, which is why the fallback sheet shown above renders on every other platform.



The native review controllers hide whether the user submitted a rating, and may decline to show the prompt at all based on their own quotas. Treat a review request as best-effort; never block your UI waiting for a result.

Chapter 60. Apple Wallet Extension

Card issuers (banks, fintechs) can let users add a payment card to Apple Wallet from inside the Wallet app itself. The user opens Wallet, taps the add button and sees the issuer's cards listed under "From apps on your iPhone" - without ever launching the issuer's app. Apple implements this through a pair of app extensions called issuer provisioning extensions, and Codename One can generate them for you from build hints with no native code in your project.

This is distinct from the in-app flow where your app shows an "Add to Apple Wallet" button. The extension flow starts inside the Wallet app, runs in a separate process and usually runs while your app isn't running at all.

60.1. How the extension works

Apple defines two extension points:

- A **non-UI extension** that Wallet queries for status and the list of available cards, and that produces the encrypted provisioning payload when the user adds a card. Wallet gives it 100ms to answer the status query, so it can't perform network calls or heavy work there.
- An optional **authorization UI extension** that presents a login screen inside Wallet when the non-UI extension reports that authentication is required.

Because the extensions run in a separate process, they can't call into your Java code. The Codename One integration bridges this in two ways:

1. Your app pre-publishes the card list, card art and an auth token through `com.codename1.payment.WalletExtension`. The data lands in a shared App Group container where the generated extension reads it instantly.
2. The one step that genuinely needs the issuer's backend - producing `encryptedPassData` from the certificates and nonce that Apple hands the extension - is performed by an HTTPS endpoint you host. The generated extension POSTs the request there and relays the response to Wallet.

60.2. Prerequisites from Apple

Before any of this works you need approvals that only Apple and your card network can grant:

- The `com.apple.developer.payment-pass-provisioning` entitlement is restricted. Apple grants it per app on request (see the In-App Provisioning documentation on the Apple developer site). After approval, enable it under the App ID's **Additional Capabilities** tab for the app **and for each extension App ID**.
- Each extension needs its own App ID (for example `com.mybank.app.WalletNonUIExtension`) and its own provisioning profile carrying the entitlement.
- The card network/PNO pass metadata must list the extension App IDs in `associatedApplicationIdentifiers`, otherwise Wallet never invokes the extensions.
- The extensions require iOS 14. The generated extension targets set their own deployment target, so your app can still target an older iOS version.

60.3. Mode 1: Generate the extensions from build hints

Add these build hints to the iOS build:

```
codename1.arg.ios.wallet.extension=true
codename1.arg.ios.wallet.appGroup=group.com.mybank.app
codename1.arg.ios.wallet.issuerEndpoint=https://api.mybank.com/wallet/provision
```

That generates the non-UI extension, wires it into the Xcode project as an embedded extension target and injects the App Group into the app and extension entitlements. To also generate the login UI extension add:

```
codename1.arg.ios.wallet.includeUI=true
codename1.arg.ios.wallet.authEndpoint=https://api.mybank.com/wallet/login
```

In Java, publish the user's cards whenever they change (typically after login) and keep a fresh token published so Wallet can skip the login screen:

Call `WalletExtension.clear()` on logout. The card art must be a PNG without personally identifiable information (Apple requires square corners and no full card number). The extension automatically filters out cards that are already provisioned on the device or the paired watch.

60.3.1. The issuer endpoint

When the user adds a card, the generated extension POSTs JSON to the `ios.wallet.issuerEndpoint` URL:

```
{
  "certificates": ["base64...", "base64..."],
  "nonce": "base64...",
  "nonceSignature": "base64...",
  "cardIdentifier": "the WalletPassEntry identifier",
  "authToken": "the token published via setAuthToken"
}
```

The token is also sent as an `Authorization: Bearer` header. Your backend performs the network-specific encryption (this always happens server side - the keys never live on the device) and responds with:

```
{
  "activationData": "base64...",
  "encryptedPassData": "base64...",
  "ephemeralPublicKey": "base64..."
}
```

For the RSA_V2 scheme return `wrappedKey` instead of `ephemeralPublicKey`.

60.3.2. The login UI extension

When you can't guarantee a fresh token, report `setRequiresAuthentication(true)` and enable the UI extension. It shows a minimal username/password form inside Wallet, POSTs `{"username", "password"}` to the `ios.wallet.authEndpoint` URL, expects `{"token"}` back and stores it where the non-UI extension picks it up. If the generated form doesn't fit your brand or login model, use the injection hints below or bring your own extension (Mode 2).

60.3.3. Customizing the generated code

The generated Objective-C contains marker comments at every interesting point, and each marker has a matching build hint that injects your snippet there: `ios.wallet.nonuiImportsInject`, `ios.wallet.statusInject`, `ios.wallet.passEntriesInject`, `ios.wallet.remotePassEntriesInject`, `ios.wallet.generateRequestInject` (runs before the POST - the mutable `payload` dictionary is in scope, so you can add fields your backend expects), `ios.wallet.generateResponseInject`, `ios.wallet.uiImportsInject`, `ios.wallet.uiViewDidLoadInject`, `ios.wallet.uiAuthRequestInject` and `ios.wallet.uiAuthResponseInject`.

Xcode build settings of the extension targets can be overridden with `ios.wallet.nonui.buildSettings.SETTING=value` and `ios.wallet.ui.buildSettings.SETTING=value` - useful in local builds for `DEVELOPMENT_TEAM` or `CODE_SIGN_STYLE`.

60.3.4. Provisioning profiles for cloud builds

Cloud device builds sign each extension with its own profile. Either place the `.mobileprovision` files under `common/src/main/resources` and name them in build hints:

```
codename1.arg.ios.wallet.nonuiProvisioningProfile=WalletNonUI.mobileprovision
codename1.arg.ios.wallet.uiProvisioningProfile=WalletUI.mobileprovision
```

Alternatively, host them at URLs the build server can reach and supply `ios.wallet.nonuiProvisioningURL` / `ios.wallet.uiProvisioningURL` instead. Profiles contain no private keys, so bundling them in resources is safe; the build keeps them out of the final app bundle. The build validates that each profile matches your distribution certificate and actually carries the payment-pass-provisioning entitlement, and fails with an actionable message when it doesn't.

In local builds ("ios-source" target) no profile hints are needed - you sign the generated Xcode project in Xcode as usual.

60.4. Mode 2: Bring your own extension

If you already have Xcode extension targets - from an issuer SDK vendor or an existing native app - skip the `ios.wallet.*` hints and drop each extension into the generic app extension mechanism instead. Create `ios/app_extensions/MyWalletExtension/` in your project containing:

- The extension's source files (`.m`, `.h`, `.swift`)

- `Info.plist` with the `NSExtension` dictionary
- `MyWalletExtension.entitlements`
- Optional `buildSettings.properties` with Xcode build settings, one per line (set `IPHONEOS_DEPLOYMENT_TARGET=14.0` for Wallet extensions)
- Optional `.mobileprovision` for cloud device builds (or supply `ios.appext.MyWalletExtension.provisioningURL`; qualify it as `ios.debug.appext.MyWalletExtension.provisioningURL` / `ios.release.appext.MyWalletExtension.provisioningURL` to use different development and distribution profiles per build type)

Each folder becomes an embedded extension target with bundle id `<your package>.<folder name>`. This mechanism isn't Wallet-specific - it embeds any iOS app extension type.

60.5. Testing

Wallet extensions can't be exercised in the iOS simulator's Wallet app; testing the full flow requires a device, a provisioning profile with the entitlement and a card network sandbox. What you can verify earlier:

- A local "ios-source" build with the hints produces the `WalletNonUIExtension`/`WalletUIExtension` targets - open the generated Xcode project and build.
- `WalletExtension.isSupported()` returns `false` in the simulator and on other platforms, and all publish calls are safe no-ops there, so the Java code needs no platform guards beyond the one shown above.

Chapter 61. Advanced topics/under the hood

61.1. Sending arguments to the build server

When you send a build to the server, you can provide more parameters. The server incorporates those parameters into the build process as build hints.

These hints are often called "build hints" or "build arguments." They behave like compiler flags that tune the build server's behavior. That makes them useful for fast iteration on new functionality without rebuilding plugin UI for every change. They're also useful when you need to expose low-level behavior such as customizing the Android manifest XML or the iOS plist.

Launch Codename One Settings from the Maven project root, then select **Hints** in the left navigation rail. The hints use the `key=value` style.

```
mvn cn1:settings
```

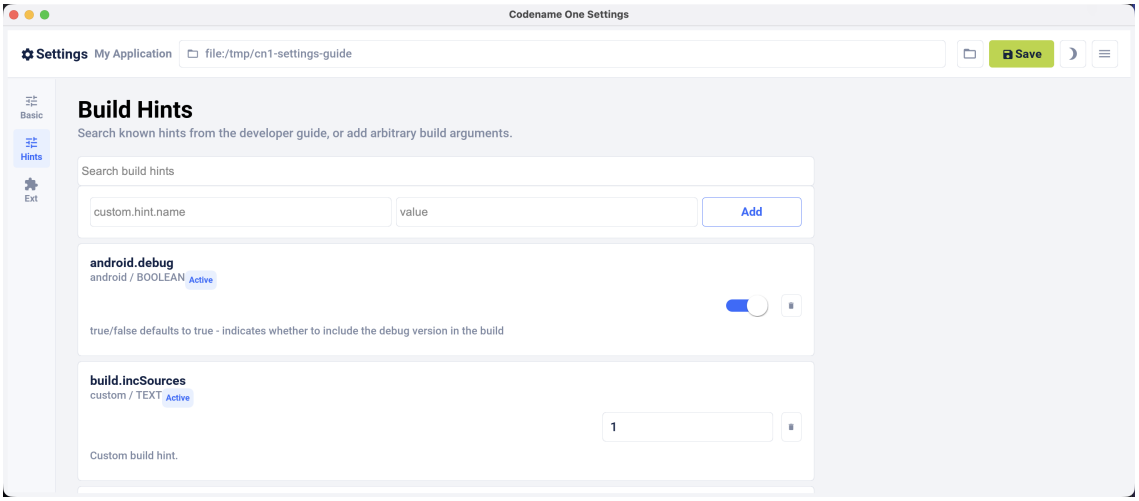


Figure 324. The build hints UI in Codename One Settings

You can also set the build hints directly in the `codenameone_settings.properties` file. When you do that, each setting must start with the `codename1.arg.` prefix. For example, `android.debug=true` becomes `codename1.arg.android.debug=true`.

Application code can read a custom build argument through `Display.getProperty()`:

```
class AppArgSnippet {
    public void readArgument() {
        String arg = Display.getInstance().getProperty("AppArg", null);
    }
}
```

Here is the current list of supported arguments. Build hints change over time, so consult the discussion forum if you don't find what you need here:

Table 12. Build hints

Name	Description
build.cn1Version	Pro/Enterprise only. Pins the cloud build to a specific released Codename One version using the Maven release scheme (for example `7.0.182`), or to `master` to build against the current development head. The build server fetches that version's framework artifacts. Pro accounts can target versions published within the last two months; Enterprise within the last six months. Requesting an older version, a version that was never published, or using this hint without a Pro/Enterprise subscription fails the build with an explanatory error. See Section 61.2, "Versioned builds".
android.debug	true/false defaults to true - indicates whether to include the debug version in the build
android.release	true/false defaults to true - indicates whether to include the release version in the build
android.onDeviceDebug	Boolean true/false defaults to false. When `true`, the generated `AndroidManifest.xml` is marked `android:debuggable="true"`, R8/proguard is disabled, and the build is pinned to debug-only (`android.release` is forced off and `android.debug` is forced on) so a stray hint can't ship a release-signed APK that's `debuggable="true"`. Pair with the `cn1:android-on-device-debugging` Maven goal (or the bundled IntelliJ run configs) to install, launch, forward JDWP, and stream logcat through adb. Has no effect on builds that don't carry it — release builds are unaffected. See the On-Device Debugging (Android) chapter for the full flow.
android.installLocation	Maps to android:installLocation manifest entry defaults to auto. Can also be set to internalOnly or preferExternal.
android.xapplication	defaults to an empty string. Allows developers of native Android code to add text within the application block to define things such as widgets, services etc.

Name	Description
android.permission.PERMISSION_NAME	true/false Whether to include a particular permission. Use of these build hints is preferred to `android.xpermissions` since they avoid possible conflicts with libraries. See Android's `Manifest.permission` docs [https://developer.android.com/reference/android/Manifest.permission.html] for a full list of permissions.
android.permission.PERMISSION_NAME.maxSdkVersion	Will be translated to the `maxSdkVersion` attribute of the `<uses-permission>` tag for the corresponding `android.permission.PERMISSION_NAME` build hint. (Optional)
android.permission.PERMISSION_NAME.required	true/false Will be translated to the `required` attribute of the `<uses-permission>` tag for the corresponding `android.permission.PERMISSION_NAME` build hint. (Optional)
android.xpermissions	more permissions for the Android manifest
android.xintent_filter	Allows adding an intent filter to the main android activity
android.tv	true/false (defaults to false). Marks the build as an Android TV / Google TV app. Adds the `LEANBACK_LAUNCHER` intent category to the launcher activity (so the app appears on the TV home screen), declares the `android.software.leanback` feature, makes `android.hardware.touchscreen` optional (so it installs on touchless TVs), and generates a 320×180 launcher banner (`@drawable/tv_banner`) from the app icon. The same APK still installs and runs on phones and tablets, and `CN.isTV()` returns true at runtime on a TV.
android.activity.launchMode	Allows explicitly setting the `android:launchMode` attribute of the main activity in android. Default is "singleTop," but for some applications you may need to change this behaviour. In particular, apps that are meant to open a file type will need to set this to "singleTask." See Android docs for the activity element [https://developer.android.com/guide/topics/manifest/activity-element.html] for more information about the `android:launchMode` attribute.

Name	Description
android.licenseKey	The license key for the Android app, this is required if you use in-app purchase on Android
android.signingV1	true/false Default true. See https://source.android.com/docs/security/features/apksigning
android.signingV2	true/false Default true. See https://source.android.com/docs/security/features/apksigning
android.signingV3	true/false Default true. See https://source.android.com/docs/security/features/apksigning
android.signingV4	true/false Default true. See https://source.android.com/docs/security/features/apksigning
android.stack_size	Size in bytes for the Android stack thread
android.statusbar_hidden	true/false defaults to false. When set to true hides the status bar on Android devices.
android.facebook_permissions	Permissions for Facebook used in the Android build target, applicable only if Facebook native integration is used.
android.googleAdUnitId	Allows integrating admob/google play ads, this is effectively identical to google.adUnitId but only applies to Android
android.googleAdUnitTestDevice	Device key used to mark a specific Android device as a test device for Google Play ads defaults to C6783E2486F0931D9D09FABC65094FDF
android.includeGPlayServices	**Deprecated, please android.playService.*!** Indicates whether Google Play Services should be included into the build, defaults to false but that might change based on the functionality of the application and other build hints. Adding Google Play Services support allows you to use a more refined location implementation and invoke some Google specific functionality from native code.

Name	Description
android.playService.plus, android.playService.auth, android.playService.base, android.playService.identity, android.playService.indexing, android.playService.appInvite, android.playService.analytics, android.playService.cast, android.playService.gcm, android.playService.drive, android.playService.fitness, android.playService.location, android.playService.maps, android.playService.ads, android.playService.vision, android.playService.nearby, android.playService.panorama, android.playService.games, android.playService.safetynet, android.playService.wallet, android.playService.wearable	Allows including only a specific play services library portion. Notice that this setting conflicts with the deprecated `android.includeGPlayServices` and only works with the Gradle-based Android build pipeline. If none of the services are defined to true then plus, auth, base, analytics, gcm, location, maps & ads will be set to true. If one or more of the `android.playService` entries are defined to something then all entries will default to false.
android.playServicesVersion	The version number of play services to build against. Experimental. **Use with caution** as building against versions other than the server default may introduce incompatibilities with some Codename One APIs.
xxx.minPlayServicesVersion	This is a special case build hint. You can use any prefix to the build hint and the convention is to use your cn1lib name. It's identical to `android.minPlayServicesVersion` with the exception that the "highest version wins." That way if your cn1lib requires play services 9+ and uses: `myLib.minPlayServicesVersion=9.0.0` and another library has `otherLib.minPlayServicesVersion=10.0.0` then play services will be 10.0.0
android.multidex	Boolean true/false defaults to false. Multidex allows Android binaries to reference more than 65536 methods. This slows builds a bit so you have it off by default but if you get a build error mentioning this limit you should turn this on.

Name	Description
android.headphoneCallback	Boolean true/false defaults to false. When set to true it assumes the main class has two methods: `headphonesConnected` & `headphonesDisconnected` which it invokes appropriately as needed
android.gpsPermission	Indicates whether the GPS permission should be requested, it's autodetected by default if you use the location API. But, some code might want to explicitly define it
android.asyncPaint	Boolean true/false defaults to true. Toggles the Android pipeline between the legacy pipeline (false) and new pipeline (true)
android.stringsXml	Allows injecting more entries into the strings.xml file using a value that includes something like this `<string name="key1">value1</string><string name="key2">value2</string>`
android.supportV4	Boolean true/false defaults to false but that can change based on usage (for example, push implicitly activates this). Indicates whether the android support v4 library should be included in the build
android.style	Allows injecting more data into the `styles.xml` file right before the closing resources tag
android.enableAdaptiveIcons	Boolean true/false defaults to false. Enables Android adaptive icon generation in Android Gradle builds. When enabled, Codename One generates `mipmap` launcher resources (`ic_launcher`, `ic_launcher_foreground`, and adaptive XML in `mipmap-anydpi-v26`) and uses them in the application manifest (`android:icon` and `android:roundIcon`).
android.adaptiveIconBackground	Background color to use for adaptive icons when `android.enableAdaptiveIcons=true` and no background image is supplied. Defaults to `#ffffff` and is written as `@color/ic_launcher_background`.
android.adaptiveIconBackgroundImage	Optional path (relative to the root of the native Android project) to an image file to use as the adaptive icon background when `android.enableAdaptiveIcons=true`. If this property is set, it overrides `android.adaptiveIconBackground`.

Name	Description
android.cusom_layout1	Applies to any number of layouts as long as they're in sequence (for example, android.cusom_layout2, android.cusom_layout3 etc.). Will write the content of the argument as a layout XML file and give it the name `cusom_layout1.xml` onwards. This can be used by native code to work with XML files
android.keyboardOpen	Boolean true/false defaults to true. Toggles the new async keyboard mode that leaves the keyboard open while you move between text components
android.versionCode	Allows overriding the auto generated version number with a custom internal version number specifically used for the XML attribute `android:versionCode`
android.captureRecord	Indicates whether the `RECORD_AUDIO` permission should be requested. Can be `enabled` or any other value to disable this option
android.nonconsumable	Comma delimited string of items that are non-consumable in the in-app purchase API
android.removeBasePermissions	Boolean true/false defaults to false. Disables the built-in permissions specifically `INTERNET` permission (that is, no networking…)
android.blockExternalStoragePermission	Boolean true/false defaults to false. Disables the external storage (SD card) permission
android.blockReadMediaPermissions	Boolean true/false, defaults to the value of `android.blockExternalStoragePermission`. Suppresses the `READ_MEDIA_VIDEO` and `READ_MEDIA_AUDIO` permissions that playing a URI adds on API 33 and above
android.requestReadMediaPermissions	Boolean true/false defaults to false. Declares `READ_MEDIA_IMAGES`, `READ_MEDIA_VIDEO` and `READ_MEDIA_AUDIO` on API 33 and above even when the build detected no media playback. `READ_MEDIA_IMAGES` is only ever added by this hint
android.min_sdk_version	The least SDK required to run this app, the default value changes based on functionality but can be as low as 7. This corresponds to the XML attribute `android:minSdkVersion`.

Name	Description
android.manifest.queries	Embeds XML content into the <queries> section of the Android manifest file. This is required in Android 11 for package visibility [https://developer.android.com/training/package-visibility]. See queries element Android documentation [https://developer.android.com/guide/topics/manifest/queries-element].
android.mockLocation	Boolean true/false defaults to true. Toggles the mock location permission which is on by default, this allows easier debugging of Android device location based services
android.smallScreens	Boolean true/false defaults to true. Corresponds to the `android:smallScreens` XML attribute and allows disabling the support for small phones
android.xapplication_attr	Allows injecting more attributes into the `application` tag in the Android XML
android.xactivity	Allows injecting more attributes into the `activity` tag in the Android XML
android.streamMode	The mode in which the volume key should behave, defaults to OS default. Allows setting it to `music` for music playback apps
android.pushVibratePattern	Comma delimited long values to describe the push pattern of vibrate used for the `setVibrate` native method
android.enableProguard	Boolean true/false defaults to true. Allows disabling the proguard obfuscation even on release builds, notice that this isn't recommended
android.proguardKeep	Arguments for the keep option in proguard allowing you to keep a pattern of files for example, `-keep class com.mypackage.ProblemClass { *; }`
android.shrinkResources	Boolean true/false defaults to false. Used only in conjunction with android.enableProguard. Strips out unused resources to reduce apk size. Since 7.0
android.sharedUserId	Allows adding a manifest attribute for the sharedUserId option
android.sharedUserLabel	Allows adding a manifest attribute for the sharedUserLabel option

Name	Description
android.targetSDKVersion	Indicates the Android SDK used to compile the Android build defaults to 21. Notice that not all targets will work since the source might have some limitations and not all SDK targets are installed on the build servers.
android.useAndroidX	Use Android X instead of support libraries. This will also run a find/replace on all source files to replace support libraries and artifacts with AndroidX equivalents.
android.rootCheck	Boolean true/false defaults to false. Indicates whether the app should check for root access on the device. If root access is detected, the app will exit.
android.fridaDetection	Boolean true/false defaults to false. Indicates whether the app should check for the presence of the Frida [https://www.frida.re/] dynamic instrumentation toolkit on the device. If Frida is detected, the app will exit. This uses the [frida-blocker](https://github.com/shannah/frida-blocker) library to perform the frida detection.
android.fridaVersion	x.y.z The version of [frida-blocker](https://github.com/shannah/frida-blocker) to use to perform frida detection. This is only relevant if `android.fridaDetection=true`. If omitted, it will use the latest tested version in the build server.
android.fridaDebugLogging	Boolean true/false defaults to false. If true, it will add verbose debug logs during frida detection to show which check if fails on.
android.theme	Light or Dark defaults to Light. On Android 4+ the default Holo theme is used to render the native widgets sometimes and this indicates whether holo light or holo dark is used. This doesn't affect the Codename One theme but that might change in the future.
android.web_loading_hidden	true/false defaults to false - set to true to hide the progress indicator that appears when loading a web page on Android.

Name	Description
block_server_registration	true/false flag defaults to false. By default Codename One applications register with the Codename One server. Setting this to true blocks them from sending information to the Codename One cloud, which is kept for statistical purposes and may be used to provide more installation stats in the future.
facebook.appId	The application ID for an app that requires native Facebook login integration, this defaults to null which means native Facebook support shouldn't be in the app
facebook.clientToken	The client token for an app that requires native Facebook login integration, this is required if the facebook.appId is set.
gcm.sender_id	The Android/chrome push identifier, see the push section for more details
android.background_push_handling	Deliver push messages on Android when the app is minimized by setting this to "true." Default behaviour is to deliver the message only if the app is in the foreground when received, or after the user taps on the notification to open the app, if the app was in the background when the message was received.
desktop.mac.plist.PLISTKEY	Set the key `PLISTKEY` in the Info.plist file for desktop mac build. For example, `desktop.mac.plist.LSApplicationCategoryType=public.app-category.business`. See Apple Documentation of Info.plist keys and values for a full list of supported keys [https://developer.apple.com/library/archive/documentation/General/Reference/InfoPlistKeyReference/Introduction/Introduction.html]. + Only supported for App Store builds. See macOS Desktop Build Options [https://www.codenameone.com/developer-guide.html#_mac_os_desktop_build_options] for more information.

Name	Description
desktop.mac.plistInject	Injects raw XML into the Info.plist file for desktop builds. For example, `desktop.mac.plistInject=<key>LSApplicationCategoryType</key><string>public.app-category.business</string>` + Only supported for App Store builds. See macOS Desktop Build Options [https://www.codenameone.com/developer-guide.html#_mac_os_desktop_build_options] for more information.
windows.arch	Native Windows (`windows-device`) target CPU: `x64` (default), `arm64`, or `both`. An x64 binary also runs on Windows-on-ARM via the OS's x64 emulation. See Chapter 84, *Working with the native Windows port*.
windows.debug	Native Windows target: `true`/`false` (default `false`). When `false` the `.exe` is optimized and stripped (with the `.pdb` in its own file); `true` keeps symbols (a single x64 build) for crash symbolication. Optimizations stay on either way.
windows.signing.pkcs12 / windows.signing.password / windows.signing.timestampUrl / windows.signing.digest / windows.signing.name / windows.signing.url	Native Windows Authenticode signing of the produced `.exe` (via `osslsigncode`). A certificate is taken from `windows.signing.pkcs12` or the build's uploaded certificate; set `windows.signing=false` to skip. Unsigned binaries run but trip SmartScreen / "Unknown publisher."
linux.arch	Native Linux (`linux-device`) target CPU: `x64` (default), `arm64`, or `both`. See [Working with the native Linux port].
linux.debug	Native Linux target: `true`/`false` (default `false`). When `false` the ELF is optimized and stripped (debug info is split into a separate `.debug`); `true` keeps symbols (`RelWithDebInfo`) for crash symbolication. Optimizations stay on either way.
linux.libc	Native Linux target: `glibc` (default) or `musl`. The default compiles against an old glibc so the ELF runs on essentially any mainstream distro; `musl` targets Alpine (where the GTK stack is itself musl-built). A glibc binary and a musl binary aren't interchangeable.

Name	Description
ios.associatedDomains	Comma-delimited list of domains associated with this app. Each domain should be prefixed by a supported prefix. For example, "applinks:" or "webcredentials:." See Apple's documentation on Associated domains [https://developer.apple.com/documentation/security/password_autofill/setting_up_an_app_s_associated_domains?language=objc] for more information.
ios.bitcode	true/false defaults to false. Enables bitcode support for the build.
ios.debug.archs	Can be set to "armv7" to force iOS debug builds to be 32 bit. By default, debug builds are 64 bit only.
ios.release.archs	Can be set to "arm64" to only build iOS release builds for 64 bit. By default, release builds are both 32 and 64 bit.
ios.distributionMethod	Specifies distribution type for debug iOS builds. This is used for enterprise or ad-hoc builds (using values "enterprise" and "ad-hoc" respectively).
ios.debug.distributionMethod	Specifies distribution type for debug iOS builds only. This is used for enterprise or ad-hoc builds (using values "enterprise" and "ad-hoc" respectively).
ios.release.distributionMethod	Specifies distribution type for release iOS builds only. This is used for enterprise or ad-hoc builds (using values "enterprise" and "ad-hoc" respectively).
ios.keyboardOpen	Flips between iOS keyboard open mode and autofold keyboard mode. Defaults to true which means the keyboard will remain open and not fold automatically when editing moves to another field.
ios.uiscene	true/false (defaults to true). Enables iOS UIScene lifecycle support. UIScene lets iOS manage one or more app UI sessions independently, improving lifecycle handling in modern iOS versions. Apple has indicated UIScene will be required starting with iOS 27, so this is now on by default; set the flag to `false` only if you need to temporarily fall back to the legacy `UIApplicationDelegate` lifecycle.

Name	Description
ios.urlScheme	Allows intercepting a URL call using the syntax `<string>urlPrefix<string>`
ios.useAVKit	Use AVKit for video components on iOS rather than `MPMoviePlayerController` on iOS versions 8 through 12. iOS 13 will always use AVKit, and iOS 7 and lower will always use `MPMoviePlayerController`. Default value `false`
ios.teamId	Specifies the team ID associated with the iOS provisioning profile and certificate. Use `ios.debug.teamId` and `ios.release.teamId` to specify different team IDs for debug and release builds respectively.
ios.debug.teamId	Specifies the team ID associated with the iOS debug provisioning profile and certificate.
ios.release.teamId	Specifies the team ID associated with the iOS release provisioning profile and certificate.
ios.project_type	one of ios, ipad, iphone (defaults to ios). Indicates whether the resulting binary is targeted to the iphone only or ipad only. Notice that the IDE plugin has a "Project Type" combo box you **should** use under the iOS section.
ios.rpmalloc	`true`/`false` Use rpmalloc [https://github.com/rampantpixels/rpmalloc] instead of malloc/free for memory allocation in ParparVM. This will cause the deployment target to be changed to a minimum of iOS 8.0.
ios.statusbar_hidden	true/false defaults to false. Hides the iOS status bar if set to true.
ios.newStorageLocation	true/false defaults to false but defined on new projects as true by default. This changes the storage directory on iOS from using caches to using the documents directory which is the recommended location but might break compatibility. This is described in this issue [https://github.com/codenameone/CodenameOne/issues/1480]
ios.prerendered_icon	true/false defaults to false. The iOS build process adapts the submitted icon for iOS conventions (adding an overlay) that might not be appropriate on some icons. Setting this to true leaves the icon unchanged (only scaled).

Name	Description
ios.app_groups	Space-delimited list of app groups that this app belongs to as described in Apple's documentation [https://developer.apple.com/library/content/documentation/Miscellaneous/Reference/EntitlementKeyReference/Chapters/EnablingAppSandbox.html#//apple_ref/doc/uid/TP40011195-CH4-SW19]. These are added to the entitlements file with key `com.apple.security.application-groups`.
ios.keychainAccessGroup	Space-delimited list of keychain access groups that this app has access to as described in Apple's documentation [https://developer.apple.com/library/content/documentation/Security/Conceptual/keychainServConcepts/02concepts/concepts.html#//apple_ref/doc/uid/TP30000897-CH204-SW11]. These are added to the entitlements file with the key `keychain-access-groups`.
ios.application_exits	true/false (defaults to false). Indicates whether the application should exit on home button press. The default is to exit, leaving the application running is only tested at the moment.
ios.blockScreenshotsOnEnterBackground	true/false (defaults to false). Indicates that app should prevent iOS from taking screenshots when app enters background. Described here [https://shannah.github.io/cn1-recipes/#_hiding_sensitive_data_when_entering_background].
ios.detectJailbreak	true/false (defaults to false). When true, the iOS app will exit on launch if it detects that it's running on a jailbroken device.

Name	Description
ios.notificationPermissionAtLaunch	true/false (defaults to false). Backward-compatibility flag for the pre-issue-#4876 behavior. By default, the iOS notification permission prompt is deferred until the app calls `Push.register()` or schedules a `LocalNotification`, matching the Android flow and giving the developer a chance to display a rationale screen first. Set this hint to `true` to restore the legacy behavior in which the prompt fires automatically inside `application:didFinishLaunchingWithOptions:` as soon as the app launches. Existing apps relying on the prompt being shown at launch should set this to `true`; new apps should leave it disabled and trigger the prompt explicitly when they're ready to ask for permission.
ios.applicationQueriesSchemes	Comma separated list of url schemes that `canExecute` will respect on iOS. If the url scheme isn't mentioned here `canExecute` will return false starting with iOS 9. Notice that this collides with `ios.plistInject` when used with the `<key>LSApplicationQueriesSchemes</key>⋯` value so you should use one or the other. For example, to enable `canExecute` for a url like `myurl://xys` you can use: `myurl,myotherurl`
ios.themeMode	`auto` (default), `modern`, `ios7`, `legacy`. `auto` (unset) keeps the existing iOS 7 flat theme so pre-refactor screenshot goldens and apps see no behavior change. `modern` / `liquid` opts in to the CSS-generated iOS Modern (liquid-glass) theme shipped from `native-themes/ios-modern/theme.css`. `ios7` / `flat` is the same as `auto` - pre-liquid iOS 7 flat theme; `legacy` / `iphone` loads the pre-iOS 7 iPhone theme. The `auto` → modern flip is planned for a future release.

Name	Description
and.themeMode	`auto`, `modern` / `material`, `hololight` (default for existing apps), `legacy`. `auto` and `modern` / `material` opt in to the CSS-generated Android Material 3 theme from `native-themes/android-material/theme.css`. `hololight` is Android Holo Light (what the framework shipped on API 14+ before this refactor). `legacy` loads the pre-Holo Android theme. The legacy alias `cn1.androidTheme` is still accepted, and `and.hololight=true` still maps to `hololight`. The default stays on `hololight` for existing apps until you flip in a future release.
nativeTheme	`modern`, `legacy`, `custom` (default unset). Cross-platform override that sets both `ios.themeMode` and `and.themeMode` together when those aren't set explicitly. `modern` = liquid glass + Material 3, `legacy` = iOS 7 flat + Holo Light, `custom` disables the framework native theme entirely. The legacy alias `cn1.nativeTheme` is still accepted.
ios.interface_orientation	UIInterfaceOrientationPortrait by default. Indicates the orientation, one or more of (separated by colon :): `UIInterfaceOrientationPortrait`, `UIInterfaceOrientationPortraitUpsideDown`, `UIInterfaceOrientationLandscapeLeft`, `UIInterfaceOrientationLandscapeRight`. Notice that the IDE plugin has an "Interface Orientation" combo box you **should** use under the iOS section.
ios.xcode_version	The version of Xcode used on the server. Defaults to 4.5; accepts 5.0 as an option and nothing else.
ios.multitasking	Set to true to enable iOS multitasking and split-screen support. This only works if `ios.xcode_verson=9.2`.
java.version	Valid values include 5 or 8. Indicates the JVM version that should be used for server compilation, this is defined by default for newly created apps based on the Java 8 mode selection

Name	Description
javascript.inject_proxy	true/false (defaults to `true`). The ParparVM builder generates a same-origin proxy bundle and configures the app to use it. Setting this to `false` disables both proxy generation and proxy URL injection.
javascript.inject.beforeHead	Content to be injected into the index.html file at the beginning of the `<head>` tag.
javascript.inject.afterHead	Content to be injected into the index.html file at the end of the `<head>` tag.
javascript.minifying	true/false (defaults to `true`). By default the JavaScript code is minified to reduce file size. You may optionally disable minification by setting `javascript.minifying` to `false`.
javascript.port	`parparvm` (default) or `teavm`. Selects the public JavaScript compiler for cloud builds. `teavm` retains the original builder as a compatibility fallback.
javascript.proxy.allowedTargets	Comma-separated target origins, host names, or wildcard subdomains that a generated proxy may access, for example `https://api.example.com,*.services.example.org`. If omitted, the proxy accepts any HTTP or HTTPS target and the build emits a warning.
javascript.proxy.target	The generated ParparVM proxy deployment platform. Supported values are `jakarta-servlet` (default), `javax-servlet`, `node`, `php`, `aws-lambda`, `google-cloud-functions`, `cloudflare-workers`, and `none`.
javascript.proxy.url	The URL of an existing proxy to use for network requests. Setting it suppresses generated proxy packaging unless `javascript.proxy.target` is also set. If `javascript.inject_proxy` is `false`, this build hint is ignored.
javascript.sourceFilesCopied	true/false (defaults to `false`). Setting this flag to `true` will cause available java source files to be included in the resulting .zip and .war files. These may be used by Chrome during debugging.

Name	Description
javascript.stopOnErrors	true/false (defaults to `true`). Causes a TeaVM JavaScript build to fail when the compiler reports warnings. Setting this to `false` may allow the fallback builder to complete, but can turn compiler diagnostics into runtime failures that are more difficult to debug.
javascript.teavm.version	(Optional) The version of TeaVM to use for the build. **Use caution**, only use this property if you know what you're doing!
google.adUnitId	Allows integrating Admob/Google Play ads into the application see this [https://www.codenameone.com/blog/adding-google-play-ads.html]
ios.entitlementsInject	Content to inject into the iOS entitlements file. This should be in the Plist XML format. See Apple Entitlements Documentation [https://developer.apple.com/documentation/bundleresources/entitlements?language=objc].
ios.plistInject	entries to inject into the iOS plist file during build.
ios.includePush	true/false (defaults to false). Whether to include the push capabilities in the iOS build. Notice that the IDE plugin has an "Include Push" check box you **should** use under the iOS section.
ios.newPipeline	Boolean true/false defaults to true. Allows toggling the OpenGL ES 2.0 drawing pipeline off to the older OGL ES 1.0 pipeline.
ios.headphoneCallback	Boolean true/false defaults to false. When set to true it assumes the main class has two methods: `headphonesConnected` & `headphonesDisconnected` which it invokes appropriately as needed
ios.facebook_permissions	Permissions for Facebook used in the Android build target, applicable only if Facebook native integration is used.
ios.applicationDidEnterBackground	Objective-C code that can be injected into the iOS callback method (message) `applicationDidEnterBackground`.
ios.enableAutoplayVideo	Boolean true/false defaults to false. Makes videos "autoplay" when loaded on iOS
ios.googleAdUnitId	Allows integrating admob/google play ads, this is effectively identical to google.adUnitId but only applies to iOS

Name	Description
ios.viewDidLoad	Objective-C code that can be injected into the iOS callback method (message) `viewDidLoad`
ios.googleAdUnitIdPadding	Indicates the amount of padding to pass to the Google Ads placed at the bottom of the screen with `google.adUnitId`
ios.enableBadgeClear	Boolean true/false defaults to true. Clears the badge value with every load of the app, this is useful if the app doesn't manually keep track of number values for the badge
ios.glAppDelegateHeader	Objective-C code that can be injected into the iOS app delegate at the top of the file. For example, if you need to include headers or make special imports for other injected code
ios.glAppDelegateBody	Objective-C code that can be injected into the iOS app delegate within the body of the file before the end. This only makes sence for methods that aren't already declared in the class
ios.beforeFinishLaunching	Objective-C code that can be injected into the iOS app delegate at the top of the body of the didFinishLaunchingWithOptions callback method
ios.afterFinishLaunching	Objective-C code that can be injected into the iOS app delegate at the bottom of the body of the didFinishLaunchingWithOptions callback method
ios.locationUsageDescription	This flag is required for iOS 8 and newer if you're using the location API. It needs to include a description of the reason for which you need access to the users location

Name	Description
ios.NSXXXUsageDescription	iOS privacy flags for using certain APIs. Starting with Xcode 8, you're required to add usage description strings for certain APIs. Find a full list of the available keys in Apple's docs [https://developer.apple.com/library/content/documentation/General/Reference/InfoPlistKeyReference/Articles/CocoaKeys.html]. Some relevant ones include `ios.NSCameraUsageDescription`, `ios.NSContactsUsageDescription`, `ios.NSLocationAlwaysUsageDescription`, `NSLocationUsageDescription`, `ios.NSMicrophoneUsageDescription`, `ios.NSPhotoLibraryAddUsageDescription`, `ios.NSSpeechRecognitionUsageDescription`, `ios.NSSiriUsageDescription`
ios.add_libs	A semicolon separated list of libraries that should be linked to the app to build it
ios.pods	A comma separated list of Cocoa Pods [https://cocoapods.org/] that should be linked to the app to build it. For example, `AFNetworking ~> 2.6, ORStackView ~> 3.0, SwiftyJSON ~> 2.3`
ios.pods.platform	Sets the Cocoapods 'platform' for the Cocoapods. Some Cocoapods require a minimum platform level. For example, `ios.pods.platform=7.0`.
ios.deployment_target	Sets the deployment target for iOS builds. This is the minimum version of iOS required by a device to install the app. For example, `ios.deployment_target=8.0`. Default is '6.0'. Note: This build hint interacts with the `ios.rpmalloc` build hint. If `ios.deployment_target` is 8.0 or higher, ParparVM will use rpmalloc [https://github.com/rampantpixels/rpmalloc] by default. You can disable this default and revert back to using malloc/free by setting the `ios.rpmalloc=false` build hint.
ios.bundleVersion	Indicates the version number of the bundle, this is useful if you want to create a minor version number change for the beta testing support
ios.objC	Added the `-ObjC` compile flag to the project files which some native libraries require

Name	Description
ios.testFlight	Boolean true/false defaults to false and works only for pro accounts. Enables the testflight support in the release binaries for easy beta testing. Notice that the IDE plugin has a "Test Flight" check box you **should** use under the iOS section.
ios.metal	Boolean true/false defaults to true. Selects the Metal rendering backend (`CAMetalLayer`) over the legacy OpenGL ES 2 path (`CAEAGLLayer`). Metal is the supported iOS graphics API; OpenGL ES is deprecated. Set to `false` to opt out if you hit a Metal-only rendering regression. See Working with iOS / Metal renderer [#_metal_renderer] for details.
ios.metal.colorSpace	Selects the `CAMetalLayer.colorspace` for the Metal renderer. Accepts `sRGB` (default), `displayP3`, `deviceRGB`, `linearSRGB`, `extendedSRGB`, `extendedLinearSRGB`, or `none`. Has no effect when `ios.metal=false`. See Working with iOS / Choosing a color space [#_choosing_a_color_space_for_the_metal_renderer] for the full table.
ios.generateSplashScreens	Boolean true/false defaults to false. Enables legacy generation of splash screen images instead of the current launch storyboards.
ios.onDeviceDebug	Boolean true/false defaults to false. When `true`, the iOS build links a small JDWP listener thread (`cn1_debugger`) into the binary and the ParparVM translator emits source-line and locals metadata so a desktop proxy can serve the running app to any JDWP-speaking debugger. Has no effect on release builds. See the On-Device Debugging (iOS) chapter for the full flow.
ios.onDeviceDebug.proxyHost	Hostname or IP address the device-side listener dials to reach the desktop proxy. Default `127.0.0.1` (correct for the native iOS simulator). For a physical device, set this to the developer laptop's LAN IP. Has no effect unless `ios.onDeviceDebug=true`.
ios.onDeviceDebug.proxyPort	TCP port on `ios.onDeviceDebug.proxyHost` where the proxy is listening for the device. Default `55333`. Has no effect unless `ios.onDeviceDebug=true`.

Name	Description
ios.onDeviceDebug.waitForAttach	Boolean true/false defaults to false. When `true`, the app blocks at startup until the proxy connects and the IDE tells the VM to continue. Useful when the breakpoint to investigate fires during app boot. Has no effect unless `ios.onDeviceDebug=true`.
ios.wallet.extension	Boolean true/false defaults to false. Generates an Apple Wallet issuer provisioning extension (the "From apps on your iPhone" flow in the Wallet app) and embeds it in the build. Requires `ios.wallet.appGroup` and `ios.wallet.issuerEndpoint`. See the Apple Wallet Extension chapter.
ios.wallet.appGroup	App Group id starting with `group.` shared by the app and the generated Wallet extensions. The app publishes pass entries into this group through `com.codename1.payment.WalletExtension` and the group is added to the app and extension entitlements automatically. Required when `ios.wallet.extension=true`.
ios.wallet.issuerEndpoint	HTTPS URL of the issuer backend endpoint that produces the encrypted provisioning payload. The generated extension POSTs Apple's certificates/nonce plus the card identifier and auth token there as JSON. Required when `ios.wallet.extension=true`.
ios.wallet.includeUI	Boolean true/false defaults to false. Also generates the Wallet authorization UI extension - a login form shown inside the Wallet app when the app reports that authentication is required. Requires `ios.wallet.authEndpoint`.
ios.wallet.authEndpoint	HTTPS URL the generated login UI extension POSTs `{"username","password"}` to; the JSON response's `token` is stored in the App Group for the provisioning request. Required when `ios.wallet.includeUI=true`.
ios.wallet.nonuiExtensionName / ios.wallet.uiExtensionName	Names of the generated extension targets, also used as the bundle id suffix (`<package>.<name>`). Default `WalletNonUIExtension` / `WalletUIExtension`. The matching App IDs must be registered with the payment-pass-provisioning entitlement and listed by the card network.

Name	Description
ios.wallet.nonuiProvisioningProfile / ios.wallet.uiProvisioningProfile	Cloud device builds only. File name of the extension's `.mobileprovision` placed under `common/src/main/resources`. The profile must match the app's distribution certificate and carry the `com.apple.developer.payment-pass-provisioning` entitlement; the build keeps it out of the app bundle.
ios.wallet.nonuiProvisioningURL / ios.wallet.uiProvisioningURL	Cloud device builds only. URL fallback for the extension provisioning profile when it isn't bundled in resources, mirroring `ios.notificationServiceExtensionProvisioningURL`.
ios.wallet.nonui.buildSettings.SETTING / ios.wallet.ui.buildSettings.SETTING	Extra Xcode build settings applied to the generated extension targets, for example `ios.wallet.nonui.buildSettings.DEVELOPMENT_TEAM=ABCD123456`. Applied last so they override the generated defaults.
ios.wallet.nonuiImportsInject, ios.wallet.statusInject, ios.wallet.passEntriesInject, ios.wallet.remotePassEntriesInject, ios.wallet.generateRequestInject, ios.wallet.generateResponseInject, ios.wallet.uiImportsInject, ios.wallet.uiViewDidLoadInject, ios.wallet.uiAuthRequestInject, ios.wallet.uiAuthResponseInject	Objective-C code injected at the matching marker comment in the generated Wallet extension sources, for custom behavior at each callback (for example adding fields to the issuer endpoint payload in `generateRequestInject`). See the Apple Wallet Extension chapter.
ios.appext.NAME.provisioningURL	Cloud device builds only. URL of the provisioning profile for a generic app extension dropped into `ios/app_extensions/NAME/` (or a generated extension such as `CN1Widgets`), used when the extension folder doesn't bundle a `.mobileprovision` itself. The profile is installed on the build machine and added to the export options per bundle id. Used for both debug and release builds unless a qualified variant (below) is set.

Name	Description
ios.debug.appext.NAME.provisioningURL / ios.release.appext.NAME.provisioningURL	Cloud device builds only. Build-type-specific variants of `ios.appext.NAME.provisioningURL`: point the `debug` variant at the extension's development profile and the `release` variant at its distribution profile. The Maven build resolves the variant matching the build target (`ios-device` and `ios-on-device-debug` are debug, `ios-device-release` is release) into the unqualified hint before submitting the build; the unqualified hint acts as the fallback. The same qualifiers work for the local-path settings `codename1.ios.debug.appext.NAME.provision` / `codename1.ios.release.appext.NAME.provision`.
codename1.mac.appid	Mac Native cloud builds only. The Mac bundle identifier registered in App Store Connect / Apple Developer. Distinct from `codename1.ios.appid` because Apple treats the iOS and Mac App Store records as separate products. Required for cloud Mac builds.
codename1.mac.certificate	Mac Native cloud builds only. Path to the `.p12` file containing the Mac signing certificate(s) — *Mac App Distribution* (3rd Party Mac Developer Application) for App Store builds, *Developer ID Application* for Developer ID builds, or both bundled into the same P12 when `macNative.distribution=both`. Not interchangeable with the iOS distribution certificate. Required for cloud Mac builds.
codename1.mac.certificatePassword	Mac Native cloud builds only. Password to unlock the P12 referenced by `codename1.mac.certificate`. Required for cloud Mac builds.
codename1.mac.provision	Mac Native cloud builds only. Path to the Mac provisioning profile (`.provisionprofile`). Apple issues distinct provisioning profiles for Mac App Store and Developer ID distribution — pass the one that matches the chosen channel.
macNative.distribution	Mac Native builds only. `appStore` (default), `developerID`, or `both`. Selects which entitlements + ExportOptions plist + signing certificate to emit. `both` emits parallel `-AppStore.entitlements` `/-DeveloperID.entitlements` and matching `ExportOptions-*-Mac.plist` files so a single project can be archived to either channel.

Name	Description
macNative.teamId	Mac Native builds only. Apple Developer Team ID (alphanumeric). Falls back to `ios.release.teamId` → `ios.teamId` → `ios.debug.teamId` since most apps share a single Apple Developer Team for iOS and Mac.
macNative.bundleId	Mac Native builds only. Used only when `macNative.deriveBundleId=false`. Default: `<packageName>.mac`.
macNative.deriveBundleId	Mac Native builds only. `true` (default) maps to Xcode's `DERIVE_MACCATALYST_PRODUCT_BUNDLE_IDENTIFIER=YES` (Xcode appends `.maccatalyst` to the iOS bundle ID). Set to `false` to take the bundle ID verbatim from `macNative.bundleId`.
macNative.minDeploymentTarget	Mac Native builds only. Minimum macOS version (`MACOSX_DEPLOYMENT_TARGET`). Default `10.15` — earlier versions don't support Mac Catalyst.
macNative.iosMinDeploymentTarget	Mac Native builds only. iOS deployment-target floor for the Catalyst slice (`IPHONEOS_DEPLOYMENT_TARGET`). Default `13.1`. The plugin coerces the iOS slice's minimum upward when set.
macNative.appCategory	Mac Native builds only. `LSApplicationCategoryType` in the generated Info.plist. Default `public.app-category.utilities`. See Apple's category list [https://developer.apple.com/documentation/bundleresources/information_property_list/lsapplicationcategorytype].
macNative.copyright	Mac Native builds only. `NSHumanReadableCopyright` in the Info.plist. Defaults to `Copyright (c) <year> <vendor>`.
macNative.signing.style	Mac Native builds only. `automatic` (default) lets Xcode pick the signing certificate; `manual` forces the certificate identity hints below to be respected verbatim.
macNative.signingIdentity.appStore	Mac Native builds only. Signing certificate identity for the App Store channel. Default `Apple Distribution`.
macNative.signingIdentity.developerID	Mac Native builds only. Signing certificate identity for the Developer ID channel. Default `Developer ID Application`.

Name	Description
macNative.provisioningProfile.appStore	Mac Native builds only. Provisioning profile name for App Store distribution — used only when `macNative.signing.style=manual`.
macNative.provisioningProfile.developerID	Mac Native builds only. Provisioning profile name for Developer ID distribution — used only when `macNative.signing.style=manual`.
macNative.entitlements.appSandbox	Mac Native builds only. `true` enables `com.apple.security.app-sandbox`. Default is `true` for the `appStore` channel (Mac App Store requires the sandbox), `false` for `developerID`.
macNative.entitlements.network.client	Mac Native builds only. Toggles `com.apple.security.network.client`. Default `true`.
macNative.entitlements.network.server	Mac Native builds only. Toggles `com.apple.security.network.server`. Default `false`.
macNative.entitlements.files.userSelected	Mac Native builds only. `readwrite` (default), `readonly`, or `none`. Sets the matching `com.apple.security.files.user-selected.*` entitlement.
macNative.entitlements.hardenedRuntime	Mac Native builds only. `true` enables hardened runtime restrictions. Default is `true` for `developerID` (notarization requires it), `false` for `appStore`.
macNative.entitlements.allowJit	Mac Native builds only. `true` enables `com.apple.security.cs.allow-jit` for hardened runtime. ParparVM is AOT-compiled so this is `false` by default; flip when bundling a JIT-using cn1lib.
macNative.entitlements.extra	Mac Native builds only. Free-form XML inserted verbatim inside the `<dict>⋯</dict>` of the generated entitlements plist. Use for entitlements Codename One doesn't expose individually.

Name	Description
macNative.fixedWindowSize	Mac Native builds only. Opt-in. Format `<width>x<height>` — for example `1024x685`. When set, the Catalyst window's `UISceneSession.sizeRestrictions` minimum and maximum are pinned to the requested size so every launch produces a byte-identical window. Default unset, in which case the window is resizable. The CI screenshot pipeline turns this on to keep the strict-pixel golden comparison stable; production apps should leave it off.
desktop.width	Width in pixels for the form in desktop builds, will be doubled for retina grade displays. Defaults to 800.
desktop.height	Height in pixels for the form in desktop builds, will be doubled for retina grade displays. Defaults to 600.
desktop.adaptToRetina	Boolean true/false defaults to true. When set to true some values will ve implicitly doubled to deal with retina displays and icons etc. Will use higher DPI's
desktop.resizable	Boolean true/false defaults to true. Indicates whether the UI in the desktop build is resizable
desktop.fontSizes	Indicates the sizes in pixels for the system fonts as a comma delimited string containing 3 numbers for small,medium,large fonts.
desktop.theme	Name of the theme res file (without the ".res" extension) to use as the "native" theme. By default this is native indicating iOS theme on Mac and Windows Metro on Windows. If its something else then the app will try to load the file /themeName.res (placed in native/Java SE directory).
desktop.themeMac	Same as `desktop.theme` but specific to macOS
desktop.themeWin	Same as `desktop.theme` but specific to Windows
desktop.windowsOutput	Can be exe or msi depending on desired results
desktop.win.cef	Whether to use CEF for media and BrowserComponent instead of JavaFX in windows desktop builds. true/false. Default value is `false` (Jan 2021), but this will be changed to `true` in a future version.

Name	Description
desktop.mac.cef	Whetherto use CEF for media or BrowserComponent instead of JavaFX in Mac desktop builds. true/false. Default value is `false` (Jan 2021), but this will be changed to `true` in a future version.
tvNative.enabled	true/false (defaults to false). Adds an Apple TV (tvOS) application target to the iOS build. The tvOS app is a separate `appletvos` target built from the same Java/Kotlin sources through ParparVM (UIKit + Metal; tvOS has no OpenGL ES). Enabling it doesn't change the iOS app — in particular it doesn't override the iOS app's `ios.metal` setting. Also turned on implicitly by `codename1.tvMain`.
tvNative.mainClass (a.k.a. codename1.tvMain)	Fully-qualified tvOS lifecycle entry class. Setting it auto enables the tvOS target. If omitted while `tvNative.enabled=true`, the tvOS app reuses the phone main class.
tvNative.bundleId	Bundle identifier of the tvOS app. Defaults to `<packageName>.tvos`.
tvNative.minDeploymentTarget	`TVOS_DEPLOYMENT_TARGET` for the tvOS target. Defaults to `13.0`.
tvNative.displayName	The tvOS app name shown on Apple TV. Defaults to the app's display name.
tvNative.teamId	Apple Developer Team ID used to sign the tvOS target. Falls back to the iOS team id (`ios.release.teamId` / `ios.teamId` / `ios.debug.teamId`).
mac.desktop-vm	The JVM the should be bundled with Mac desktop build. Mac desktop builds only. Supported values: zuluFx8, zulu11, zuluFx11
win.desktop-vm	The JVM that should be bundled in the Windows desktop build. Windows desktop builds only. Supported values: zulu8, zuluFx8, zulu8-32bit, zuluFx8-32bit, zulu11, zuluFx11, zulu11-32bit, zuluFx11-32bit
windows.extensions	Historical build hint for the discontinued UWP target. It's retained here only for legacy reference and isn't used by current supported build targets.

Name	Description
win.vm32bit	true/false (defaults to false). Forces windows desktop builds to use the Win32 JVM instead of the 64 bit VM making them compatible with older Windows Machines. This is off by default at the moment because of a bug in JDK 8 update 112 that might cause this to fail for some cases
win.installDirName	Windows desktop builds only. Overrides the default installation folder name suggested by the installer (under `Program Files`). Defaults to the application's main class name for backward compatibility. Use this build hint to set a user-friendly installation folder name (for example, `win.installDirName=My Application`). The application ID used by Windows for upgrade detection is unaffected, so existing installations continue to upgrade.
win.shortcutName	Windows desktop builds only. Overrides the name used for the Start Menu shortcut, the Desktop shortcut and (when `win.launchOnStart=true`) the autostart shortcut. Defaults to the application's main class name for backward compatibility. Use this build hint to set a user-friendly shortcut label (for example, `win.shortcutName=My Application`).
noExtraResources	true/false (defaults to false). Blocks codename one from injecting its own resources when set to true, the only effect this has is in slightly reducing archive size. This might have adverse effects on some features of Codename One so it isn't recommended.
windows.arch	Native Windows port only (the `windows-native` build target — not the JVM `win.*` desktop hints above). Target CPU architecture for the standalone `.exe`: `x64` (the default) or `arm64`. Accepts the usual synonyms (`x86_64`/`amd64`, `aarch64`). clang-cl cross-compiles to the chosen architecture from either host. See the Working with the native Windows port chapter [# _working_with_the_native_windows_port].

Name	Description
windows.debug	Native Windows port only. true/false (defaults to false). When `false` the `.exe` is built optimized and **stripped** — no PDB, dead-stripped unreferenced code (`/OPT:REF`) and folded identical functions (`/OPT:ICF`) — which is the shipping default. Set `true` to keep debug symbols (a `.pdb` next to the exe, via `RelWithDebInfo` / clang-cl `/Zi` + linker `/DEBUG`) so a native crash address can be symbolized during development. Optimizations stay on in both cases.
windows.sdkRoot	Native Windows port only; used when building on a **non-Windows** host (for example a Linux build server). Path to a Windows SDK laid out by `xwin splat` [https://github.com/Jake-Shadle/xwin] (a directory containing `crt/include` and `sdk/include/um`), used to cross-compile the `.exe` with clang-cl + lld-link instead of a Visual Studio environment. If unset, the `CN1_XWIN_SYSROOT` environment variable is used. Ignored on Windows hosts, which build through Visual Studio. The same SDK serves both `windows.arch` targets (its `x86_64` / `aarch64` lib subdirs).
(signing certificate)	Native Windows port only. The code-signing certificate itself isn't a build-hint argument; configure it through project settings as `codename1.windows.signing.certificate` (path to a PKCS#12 `.pfx`/`.p12` file holding the certificate + key) and `codename1.windows.signing.password`. The build uses it for both local and cloud builds — a cloud build uploads it with the build request automatically, exactly like the iOS / Android signing certificates. When a certificate is present the produced `.exe` is Authenticode-signed with `osslsigncode` (which signs Windows PE files on any OS, so it works in the Linux build cloud); without one the exe ships unsigned (it runs, but shows "Unknown publisher" in UAC and trips SmartScreen on download).
windows.signing.timestampUrl	Native Windows port only. RFC 3161 timestamp server used when signing, so the signature stays valid after the certificate expires. Default `http://timestamp.digicert.com`; set empty to disable timestamping.

Name	Description
windows.signing.digest	Native Windows port only. Signature digest algorithm. Default `sha256`.
windows.signing.name / windows.signing.url	Native Windows port only. The description and URL embedded in the signature (the "More info" shown by Windows). Default: the app's display name, and no URL.
windows.signing	Native Windows port only. `true`/`false` (default `true`). Set `false` to force an unsigned build even when a certificate is available.

61.2. Versioned builds

By default a cloud build always uses the current Codename One release. With a *versioned build* you can instead pin the build to a specific previously released framework version, so you get reproducible output over time. This helps when stabilizing a release, supporting an older app, or isolating a regression: if an app that built correctly in the past starts failing, rebuilding against the last known good version tells you whether the change came from your code or from the framework and build server.

Set the `build.cn1Version` build hint to the version you want to target. Versions use the standard Maven release scheme published to Maven Central, for example `7.0.182`:

```
codename1.arg.build.cn1Version=7.0.182
```

The build server fetches that version's framework artifacts from Maven Central and builds against them. To keep your simulator and local compile classpath aligned with the same release, set the matching `cn1.version` in your project `pom.xml`.

Versioned builds are a Pro and Enterprise feature, and how far back you can target depends on the subscription tier:

Tier	Versions you can target
Pro	Any version released within the last two months
Enterprise	Any version released within the last six months

If you request a version older than your tier's window, a version that was never published, or you aren't on a Pro or Enterprise plan, the build fails with an explanatory message instead of falling back to the current release.

61.2.1. Building against master

Set `build.cn1Version` to `master` to build against the current development head (the latest `master` of the Codename One repository) instead of a released version:

```
codename1.arg.build.cn1Version=master
```

Every push to `master` publishes a fresh set of framework artifacts and the build server always grabs the latest, so this is the quickest way to verify an unreleased fix or feature end to end on a device. Because `master` tracks active development it can be less stable than a release, so use it for verification rather than for shipping production builds. Building against `master` is available to Pro and Enterprise subscribers.

61.3. Android permissions

One of the annoying tasks when programming native Android applications is tuning all the required permissions to match your codes requirements, Codename One aims to simplify this. The build server automatically introspects the classes sent to it as part of the build and injects the right set of permissions required by the app.

For example, sometimes developers might find the permissions that come up a bit confusing and might not understand why a specific permission came up. This maps Android permissions to the methods/classes in Codename One that would trigger them. Notice that this list isn't exhaustive as the API is rather large:

`android.permission.WRITE_EXTERNAL_STORAGE` - included by default (up to Android 13) so file APIs continue to work unless you block it with `android.blockExternalStoragePermission=true`.

`android.permission.INTERNET` - always requested because the networking stack depends on it even for offline features like local web views.

`android.permission.CAMERA` & `android.permission.RECORD_AUDIO` - required when capturing photos, video, or audio. The builder injects them automatically and the Android port prompts the user the first time the API is used.

`android.permission.READ_PHONE_STATE` - used by telephony-aware APIs like `Display.getMsisdn()` and by media integration so audio pauses on calls.

`android.permission.ACCESS_FINE_LOCATION` & `android.permission.ACCESS_COARSE_LOCATION` - requested when using `com.codename1.location` or embedded maps so location fixes can be delivered.

`android.permission.ACCESS_BACKGROUND_LOCATION` - added when background geofencing or fetch tasks are enabled; the Android port verifies the permission on Android 10+.

`android.permission.POST_NOTIFICATIONS` - prompted on Android 13+ for both push registration and local notifications.

`package.permission.C2D_MESSAGE`, `com.google.android.c2dm.permission.RECEIVE`, and `android.permission.RECEIVE_BOOT_COMPLETED` - bundled with the push subsystem so Firebase Cloud Messaging can wake the app after device restarts.

`android.permission.READ_CONTACTS` - requested when accessing the device address book through `Display.getAllContacts()` and related APIs.

`android.permission.READ_MEDIA_VIDEO` & `android.permission.READ_MEDIA_AUDIO` - added on API 33 and above when the app plays a URI through `MediaManager.createMedia(String, boolean)` or `createMediaAsync(String, boolean, Runnable)`, since that URI can point at the shared media store. The `InputStream` overloads don't add them: the Android port copies the stream into app private storage and reads nothing shared. Block them with `android.blockReadMediaPermissions=true`.

`android.permission.READ_MEDIA_IMAGES` - never added by media playback, because no Codename One API requests it at runtime. Declaring it alongside `READ_MEDIA_VIDEO` places the app under Google Play's Photo and Video Permissions policy, so ask for it only when you need it, with `android.requestReadMediaPermissions=true`.

61.3.1. Permissions under Marshmallow (Android 6+)

Starting with Marshmallow (Android 6+ API level 23) Android shifted to a permissions system that prompts users for permission the first time an API is used for example: when accessing contacts the user will receive a prompt whether to allow contacts access.



Permission can be denied and a user can later on revoke/grant a permission through external settings UI

This is great as it allows apps to be installed with a single click and no permission prompt during install which can increase conversion rates!

Enabling permissions

By default, Codename One's Gradle 8 based Android builder uses the highest Android SDK you've installed, with a minimum of API 36, for both the compile and target SDK versions. If you override the target through the `android.targetSDKVersion` build hint, the builder honors it while keeping the compile SDK at least as high as the target SDK. Lowering the target may disable some compatibility libraries. Keeping the target current is strongly recommended for Play Store compliance.

Permission prompts

To test this API see the following simple contacts app:

```
Form f = new Form("Contacts", BoxLayout.y());
f.add(new InfiniteProgress());
Display.getInstance().invokeAndBlock(() -> {
    Contact[] ct = Display.getInstance().getAllContacts(true, true, false, true, true, false);
    Display.getInstance().callSerially(() -> {
        f.removeAll();
        for (Contact c : ct) {
            MultiButton mb = new MultiButton(c.getDisplayName());
            mb.setTextLine2(c.getPrimaryPhoneNumber());
            f.add(mb);
        }
        f.revalidate();
    });
});
```

```
f.show();
```

If you explicitly lower the target SDK (for example: `android.targetSDKVersion=21`) and install this app on an Android 6 device you will still see the legacy install prompt with all permissions listed up front:

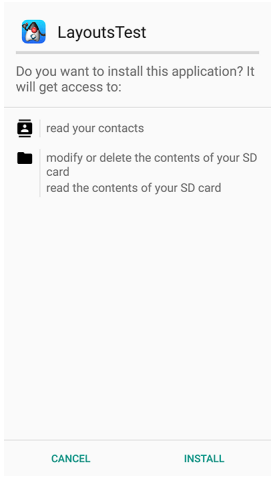


Figure 325. Install UI when using the old permissions system

When you keep the default target (API 36+) the installer defers to the runtime permission flow and the installation UI looks like this instead:

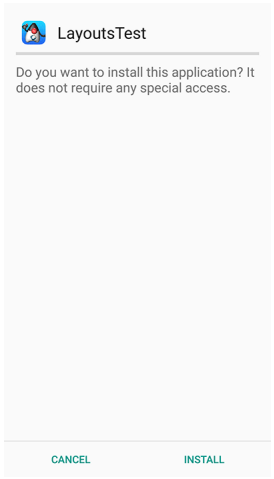


Figure 326. Install UI when using the new permissions system

When you launch the UI under the old permissions system you see the contacts instantly. In the new system you're presented with this UI:

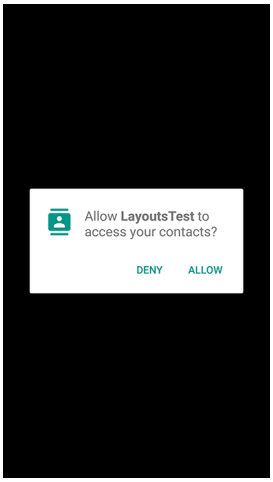


Figure 327. Native permission prompt first time

If you accept and allow all is good and the app loads as usual but if you deny then Codename One provides the user another chance to request the permission. Notice that in this case you can customize the prompt string as explained below.

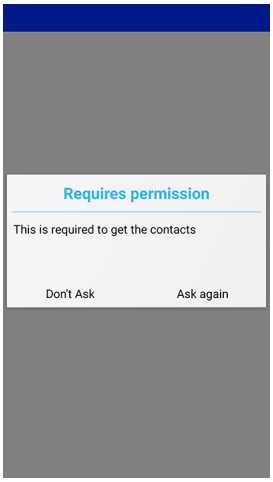


Figure 328. Codename One permission prompt

If you select don't ask then you will get a blank screen since the contacts will return as a 0 length array. This makes sense as the user is aware he denied permission and the app will still function as expected on a device where no contacts are available. For example, if the user realizes his mistake he can double back and ask to re-prompt for permission in which case he will see this native prompt:

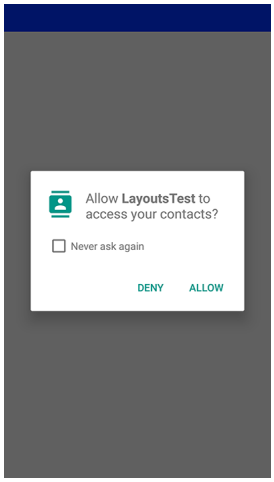


Figure 329. Native permission prompt second time

Notice that denying this second request won't trigger another Codename One prompt.

Code changes

No explicit code changes needed for this functionality to "work." The respective APIs will work like they always worked and will prompt the user seamlessly for permissions.



Some behaviors that never occurred on Android but were legal in the past might start occurring with the switch to the new API. For example: the location manager might be null and your app must always be ready to deal with such a situation

When permission is requested a user will be seamlessly prompted/warned. You can customize the permission text through `Display` properties. For example: to customize the rationale text of the contacts permission:

```
Display.getInstance().setProperty(
        "android.permission.READ_CONTACTS",
        "MyCoolChatApp needs access to your contacts so we can show you which of your friends already have MyCoolChatApp
installed");
```

The Android port also checks `UIManager.localize()` for localized values before falling back to `Display` properties. You can provide localized strings for the permission body and dialog buttons/title using keys based on the permission name:

```
com.codename1.ui.plaf.UIManager.getInstance().setBundle(new java.util.Hashtable<String, String>() {{
    put("android.permission.READ_CONTACTS", "Localized rationale for contacts");
    put("android.permission.READ_CONTACTS.title", "Localized permission title");
    put("android.permission.READ_CONTACTS.askAgain", "Localized ask again");
    put("android.permission.READ_CONTACTS.dontAsk", "Localized don't ask");
}});
```

For example, if the permission key is `android.permission.READ_CONTACTS`, you can localize these keys:

- `android.permission.READ_CONTACTS` (dialog body)
- `android.permission.READ_CONTACTS.title`

- android.permission.READ_CONTACTS.askAgain
- android.permission.READ_CONTACTS.dontAsk

The same pattern applies to other permissions. android.permission.ACCESS_BACKGROUND_LOCATION also supports: android.permission.ACCESS_BACKGROUND_LOCATION.settings, android.permission.ACCESS_BACKGROUND_LOCATION.cancel, android.permission.ACCESS_BACKGROUND_LOCATION.explanation_title, android.permission.ACCESS_BACKGROUND_LOCATION.explanation_body, and android.permission.ACCESS_BACKGROUND_LOCATION.ok.

This is optional as there is a default value defined. You can define this once in the init(Object) method but for some extreme cases permission might be needed for different things for example: you might ask for this permission with one reason at one point in the app and with a different reason at another point in the app.

Simulating prompts

You can simulate permission prompts by checking that option in the simulator menu.

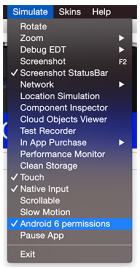


Figure 330. Simulate permission prompts menu item in the simulator

This will produce a dialog to the user whenever this happens in Android and will try to act in a similar way to the device. Notice that you can test it in the iOS simulator too.

AndroidNativeUtil's checkForPermission

If you write Android native code using native interfaces you're probably familiar with the AndroidNativeUtil class from the com.codename1.impl.android package.

This class provides access to many low-level capabilities you would need as a developer writing native code. Since native code might need to request a permission Codename One uses the same underlying logic, namely: checkForPermission.

To get a permission you can use this code as such:

```
if (!AndroidNativeUtil.checkForPermission(
        android.Manifest.permission.READ_PHONE_STATE,
        "This should be the description shown to the user...")) {
    // you didn't get the permission, you might want to return here
}
// you have the permission, do what you need
```

This will prompt the user with the native UI and later on with the fallback option as described above. Notice that the `checkForPermission` method is a blocking method and it will return when there is a final conclusion on the subject. It uses `invokeAndBlock` and can be invoked on the event dispatch thread without concern.

By default, fallback prompts are displayed using Codename One's `Dialog.show(…)`. If you're writing native Android code and need to use your own prompt implementation, you can install a custom callback:

```
AndroidNativeUtil.setPermissionPromptCallback(new PermissionPromptCallback() {
    @Override
    public boolean showPermissionPrompt(String permission, String title, String body, String positiveButtonText, String
negativeButtonText) {
        return com.codename1.ui.Dialog.show(title, body, positiveButtonText, negativeButtonText);
    }

    @Override
    public void showPermissionMessage(String permission, String title, String body, String okButtonText) {
        com.codename1.ui.Dialog.show(title, body, okButtonText, null);
    }
});
```

Pass `null` to `AndroidNativeUtil.setPermissionPromptCallback()` to restore the default behavior.

61.4. On device debugging

On-device debugging now has two dedicated chapters, one per platform:

- Chapter 78, *On-Device Debugging (iOS)* covers attaching a standard Java debugger (IntelliJ, jdb, VS Code) to an iOS app running on a device or in the native iOS simulator, as well as debugging the generated Xcode project natively.
- Chapter 79, *On-Device Debugging (Android)* covers the equivalent adb/JDWP-based flow for Android devices and emulators, including wireless debugging and debugging the generated Gradle project from Android Studio.

61.5. Native interfaces

Sometimes you may wish to use an API that's unsupported by Codename One or integrate with a 3rd party library/framework that isn't supported. These are achievable tasks when writing native code and Codename One lets you encapsulate such native code using native interfaces.

61.5.1. Introduction

Notice that when you say "native" you don't mean C/C++ always but rather the platforms "native" environment. For Android, Java or Kotlin code can be invoked with full access to the Android API. In case of iOS, Objective-C or Swift code can be invoked and so forth.



You can still access C code under Android either by using JNI from the Android native code or by using a library

Native interfaces are designed to allow primitive types, Strings, arrays of primitive types (single dimension) & PeerComponent [https://www.codenameone.com/javadoc/com/codename1/ui/PeerComponent.html] values. Any other kind of parameter/return type is prohibited. For example, once in the native layer the native code can act and query the Java layer for more information.



The reason for the limits is the disparity between the platforms. Mapping a Java `Object` to an Objective-C `NSObject` is possible but leads to odd edge cases and complexity for example: GC vs. ARC in a disparate object graph

Furthermore, native methods should avoid features such as overloading, varargs (or any Java 5+ feature for that matter) to allow portability for languages that don't support such features.



Don't rely on pass by reference/value behavior since they vary between platforms

Implementing a native layer effectively means:

1. Creating an interface that extends NativeInterface [https://www.codenameone.com/javadoc/com/codename1/system/NativeInterface.html] and defines methods with the arguments/return values declared in the previous paragraph.
2. Creating the proper native implementation hierarchy based on the call conventions for every platform within the native directory

For example: to create a simple hello world interface do something like:

```
public interface MyNative extends NativeInterface {
    String helloWorld(String hi);
}
```

You now need to right-click the class in the IDE and select the `Generate Native Access` menu item:

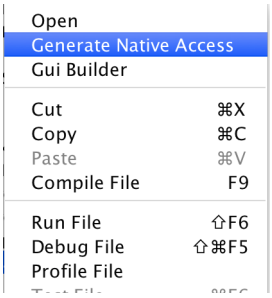


Figure 331. Generating the native code

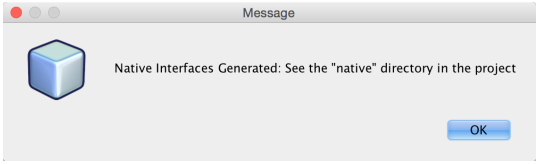


Figure 332. Once generated you're prompted that the native code is in the "native" directory

You can now look int the `native` directory in the project root (in NetBeans you can see that in the `Files` tab) and you can see something that looks like this:

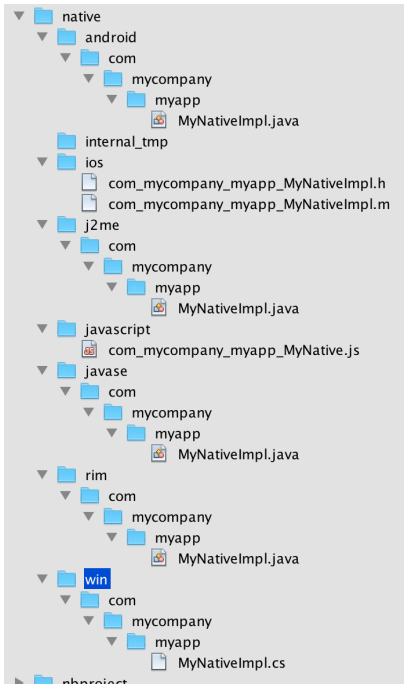


Figure 333. Native directory structure containing stubs for the various platforms

These are effectively stubs you can edit to implement the methods in native code.



If you re-run the Generate Native Access tool you will get this dialog, if you answer yes all the files will be overwritten, if you answer no files you deleted/renamed will be recreated

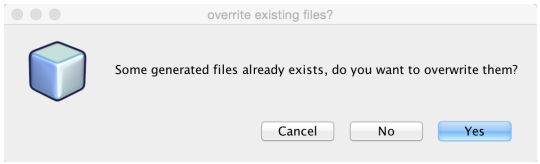


Figure 334. Running "Generate Native Access" when some/all the native files exist already

For now lets leave the stubs and come back to them soon. From the Codename One Java code you can call the implementation of this native interface using:

```
MyNative my = NativeLookup.create(MyNative.class);
if (my != null && my.isSupported()) {
    Log.p(my.helloWorld("Hi"));
}
```

Notice that for this to work you must implement the native code on all supported platforms.

You will start with Android which should be familiar and intuitive to many developers. The helper `MyNativeImplStub` class below mirrors the stub that the tooling generates under the `native/android` directory:

```
class MyNativeImplStub {
    public String helloWorld(String param) {
        return null;
```

```
    }

    public boolean isSupported() {
        return false;
    }
}
```

The stub implementation always returns `false`, `null` or `0` by default. The `isSupported` also defaults to `false` thus allowing you to implement a `NativeInterface` on some platforms and leave the rest out without knowing anything about these platforms.

You can implement the Android version in Java or Kotlin. Here is the Java version:

```
public class MyNativeImpl { ② ③
    public String helloWorld(String param) {
        Log.d("MyApp", param); ①
        return "Tada";
    }

    public boolean isSupported() {
        return true; ④
    }
}
```

① Notice that you're using the Android native `android.util.Log` class which isn't accessible from standard Codename One code

② The impl class doesn't physically implement the `MyNative` interface!
This is intentional and due to the `PeerComponent` functionality mentioned below. You don't need to add an implements clause.

③ Notice that there is no constructor and the class is public. It's crucial that the system will be able to assign the class without obstruction. You can use a constructor but it can't have any arguments and you shouldn't rely on semantics of construction.

④ You implemented the native method and that you set `isSupported` to true.



The IDE won't provide completion suggestions and will claim that there are errors in the code!
Codename One doesn't include the native platforms in its bundle for example: the full Android SDK or the full Xcode Objective-C runtime. For example, since the native code is compiled on the servers (where these runtimes are present) this shouldn't be a problem



When implementing a non-trivial native interface, send a server build with the "Include Source" option checked. Implement the native interface in the native IDE then copy and paste the native code back into Codename One

The implementation of this interface is identical for Android (Java/Kotlin) & Java SE.

Swift (iOS) and Kotlin (Android) options

For iOS native interfaces you can implement the generated ⋯`Impl` class in Objective-C *or* Swift.
For Android native interfaces you can implement the generated ⋯`Impl` class in Java *or* Kotlin.

Platform	Typical file locations for native interface implementations
Android (Java/Kotlin)	`android/src/main/java/com/mycompany/myapp/MyNativeImpl.java` `android/src/main/java/com/mycompany/myapp/MyNativeImpl.kt`

iOS (Objective-C/Swift)	`ios/src/main/objectivec/com_mycompany_myapp_MyNativeImpl.h`
	`ios/src/main/objectivec/com_mycompany_myapp_MyNativeImpl.m`
	`ios/src/main/objectivec/com_mycompany_myapp_MyNativeImpl.swift`

Native Windows (win32, C)	`win/src/main/c/com/mycompany/myapp/MyNativeImplCodenameOne.c`
Native Linux ©	`linux/src/main/c/com/mycompany/myapp/MyNativeImplCodenameOne.c`
JavaScript	`javascript/src/main/javascript/com_mycompany_myapp_MyNative.js`



The native Windows and native Linux ports compile your app to C (no JVM), so their native interface implementations are plain C, generated alongside the others by the "Generate Native Sources" tool. The native Mac target reuses the iOS Objective-C/Swift implementation, since it rides the same build.

Use the Android main thread (native EDT)

iOS, Android & pretty much any modern OS has an EDT like thread that handles events etc. The problem is that they differ in their nuanced behavior. For example: Android will respect calls off of the EDT and iOS will often crash. Some OSes enforce EDT access and will throw an exception when you violate that…

You don't need to know about these things, hidden functionality within your implementation bridges between your EDT and the native EDT to provide consistent cross-platform behavior. However, when you write native code you need awareness.

Why not Implicitly call Native Interfaces on the Native EDT?

Calling into the native EDT includes overhead and it might not be necessary for some features (for example, IO, polling etc.). Furthermore, some calls might work well with asynchronous calls while others might need synchronous results and you can't know in advance which ones you would need.

How do you access the native EDT

Within your native code in Android do something like:

```
com.codename1.impl.android.AndroidNativeUtil.getActivity().runOnUiThread(new Runnable() {
    public void run() {
```

```
        // your native code here...
    }
});
```

This will execute the block within `run()` asynchronously on the native Android UI thread. If you need synchronous execution you have a special method for Codename One:

```
com.codename1.impl.android.AndroidImplementation.runOnUiThreadAndBlock(new Runnable() {
    public void run() {
        // your native code here...
    }
});
```

This blocks in a way that's OK with the Codename One EDT which is unique to your Android port.

Gradle dependencies

Integrating a native OS library isn't hard but it sometimes requires some juggling. Most instructions target developers working with Xcode or Android Studio & you need to twist your head around them. In Android the steps for integration in most modern libraries include a Gradle dependency.

For example: you published a library that added support for Intercom [https://www.codenameone.com/blog/intercom-support.html]. The native Android integration instructions for the library looked like this:

Add the following dependency to your app's `build.Gradle` file:

```
dependencies {
    compile 'io.intercom.android:intercom-sdk:3.+'
}
```

Which instantly raises the question: "How do you do that in Codename One"?

Well, it's actually pretty simple. You can add the build hint:

```
android.gradleDep=compile 'io.intercom.android:intercom-sdk:3.+'
```

This would "work" but there is a catch...

You might need to define the specific version of the Android SDK used and specific version of Google Play Services version used. Intercom is pretty sensitive about those and demanded that you also add:

```
android.playServices=9.8.0
android.sdkVersion=25
```

Once those were defined the native code for the Android implementation became trivial to write and the library was easy as there were no jars to include.

61.5.2. Objective-C and Swift (iOS)

When generating the Objective-C code the "Generate Native Sources" tool produces two files by default: `com_mycompany_myapp_MyNativeImpl.h` & `com_mycompany_myapp_MyNativeImpl.m`. If you enable Swift stub generation in the Maven goal, it can also produce `com_mycompany_myapp_MyNativeImpl.swift`.

The `.m` files are the Objective-C equivalent of `.c` files and `.h` files contain the header/include information. In this case the `com_mycompany_myapp_MyNativeImpl.h` contains:

```
#import <Foundation/Foundation.h>

@interface com_mycompany_myapp_MyNativeImpl : NSObject {
}

-(NSString*)helloWorld:(NSString*)param;
-(BOOL)isSupported;
@end
```

And `com_mycompany_myapp_MyNativeImpl.m` contains:

```
#import "com_mycompany_myapp_MyNativeImpl.h"

@implementation com_mycompany_myapp_MyNativeImpl

-(NSString*)helloWorld:(NSString*)param{
    return nil;
}

-(BOOL)isSupported{
    return NO;
}

@end
```


Objective-C relies on argument names as part of the message (method) signature. `-(NSString*)helloWorld:(NSString*)param` isn't the same as `-(NSString*)helloWorld:(NSString*)iChangedThisName!`
don't change argument names in the Objective-C native interface!

Here is a simple implementation like above:

```
#import "com_mycompany_myapp_MyNativeImpl.h"
```

```
@implementation com_mycompany_myapp_MyNativeImpl

-(NSString*)helloWorld:(NSString*)param{
    NSLog(@"MyApp: %@", param);
    return @"Tada";
}

-(BOOL)isSupported{
    return YES;
}

@end
```

If you prefer Swift for iOS native interfaces, keep the same class naming convention (`com_mycompany_myapp_MyNativeImpl`) and annotate the class with `@objc(…)` so the runtime can discover it.

Using the iOS main thread (native EDT)

iOS has a native thread you should use for all calls like Android. Check out the Native EDT on Android section above for reference.

On iOS this is pretty like Android (if you consider objective-c to be similar). This is used for asynchronous invocation:

```
dispatch_async(dispatch_get_main_queue(), ^{
    // your native code here...
});
```

You can use this for synchronous invocation, notice the lack of the `a` in the dispatch call:

```
dispatch_sync(dispatch_get_main_queue(), ^{
    // your native code here...
});
```

The problem with the synchronous call is that it will block the caller thread, if the caller thread is the EDT this can cause performance issues and even a deadlock. It's important to be cautious with this call!

Use Cocoapods for dependencies

Cocoapods are the iOS equivalent of Gradle dependencies.

CocoaPods allow you to add a native library dependency to iOS far more than Gradle. By default you target iOS 7.0 or newer which is supported by Intercom for older versions of the library. Annoyingly CocoaPods might seem to work but some specific APIs won't work since it fell back to an older version… To solve this you've to explicitly define the build hint `ios.pods.platform=8.0` to force iOS 8 or newer. You might need to force it to even newer versions as some libraries force an

iOS 9 least etc.

Including intercom itself required a single build hint: `ios.pods=Intercom` which you can obviously extend by using commas to include many libraries. You can search the CocoaPods website [https://cocoapods.org/] for supported 3rd party libraries which includes everything you would expect. One important advantage when working with CocoaPods is the faster build time as the upload to the Codename One website is smaller and the bandwidth you've to CocoaPods is faster. Another advantage is the ability to keep up with the latest developments from the library providers.

61.5.3. C (native Windows and native Linux)

The native Windows (win32) and native Linux ports have no JVM: the "clean" ParparVM target compiles your whole app to C and then to a native binary. A native interface on these ports is therefore implemented in plain **C** rather than C#/Objective-C/Java. The "Generate Native Sources" tool emits one C file per interface:

- `win/src/main/c/com/mycompany/myapp/MyNativeImplCodenameOne.c` (native Windows)
- `linux/src/main/c/com/mycompany/myapp/MyNativeImplCodenameOne.c` (native Linux)

The two files are identical, so a native interface written once works on both ports; only the directory differs. (The native Mac target reuses the iOS Objective-C/Swift implementation, because it rides the iOS build.)

The translator emits one C function per interface method. Unlike the Objective-C stub, the function name is the fully qualified, mangled C symbol the translated app links against, so **don't rename the functions or change their signatures**; just fill in the bodies. Each function takes the thread state as its first argument and `JAVA_*` typed parameters. For example, the interface:

```
public interface MyNative extends NativeInterface {
    String helloWorld(String hi);
}
```

This generates the C stub below (returning `JAVA_NULL` until you implement it):

```
#include "cn1_globals.h"

JAVA_OBJECT com_mycompany_myapp_MyNativeImplCodenameOne_helloWorld___java_lang_String_R_java_lang_String
(CODENAME_ONE_THREAD_STATE, JAVA_OBJECT param1) {
    return JAVA_NULL;
}
```

Type mapping: the primitives (`int`, `long`, `boolean`, `byte`, `short`, `char`, `float`, `double`) become their `JAVA_INT`/`JAVA_LONG`/`JAVA_BOOLEAN` equivalents; `String`, arrays and other objects become `JAVA_OBJECT`; a `void` method returns `JAVA_VOID` and omits the `R` suffix in its name; and a `com.codename1.ui.PeerComponent` is passed and returned as a `JAVA_LONG` native peer handle. Use the core helpers declared in `cn1_globals.h` (for example `newStringFromCString` / `stringToUTF8`) to convert between C and Java values. As with the other ports, you can drop additional `.c`/`.h` sources and prebuilt libraries into the same `src/main/c` directory and they're compiled into the binary.

61.5.4. JavaScript

Native interfaces in JavaScript look a little different than the other platforms since JavaScript doesn't natively support threads or classes. The native implementation should be placed in a file with name matching the name of the package and the class name combined where the "." elements are replaced by underscores.

The default generated stubs for the JavaScript build look like this `com_mycompany_myapp_MyNative`:

```
(function(exports){

var o = {};

    o.helloWorld__java_lang_String = function(param1, callback) {
        callback.error(new Error("Not implemented yet"));
    };

    o.isSupported_ = function(callback) {
        callback.complete(false);
    };

exports.com_mycompany_myapp_MyNative= o;

})(cn1_get_native_interfaces());
```

A simple implementation looks like this:

```
(function(exports){

var o = {};

    o.helloWorld__java_lang_String = function(param1, callback) {
        callback.complete("Hello World!!!");
    }

    o.isSupported_ = function(callback) {
        callback.complete(true);
    };

exports.com_my_code_MyNative = o;

})(cn1_get_native_interfaces());
```

Notice that you use the `complete()` method of the provided callback to pass the return value rather than using the `return` statement. This is to work around the fact that JavaScript doesn't natively support threads. The **Java** thread that's calling your native interface will block until your method calls `callback.complete()`. This allows you to use asynchronous APIs inside your native method while still allowing Codename One to work use your native interface through a synchronous API.



Make sure you call either `callback.complete()` or `callback.error()` in your method at some point, or you will cause a deadlock in your app (code calling your native method will sit and "wait" forever for your method to return a value).

The naming conventions for the methods themselves are modeled after the naming conventions shown in the previous examples:

`<method-name>__<param-1-type>_<param-2-type>_⋯<param-n-type>`

Where `<method-name>` is the name of the method in Java, and the `<param-X-type>`s are a string representing the parameter type. The general rule for these strings are:

1. Primitive types are mapped to their type name. (For example: `int` to "int," `double` to "double," etc.).
2. Reference types are mapped to their fully qualified class name with '.' replaced with underscores. For example: `java.lang.String` would be `java_lang_String`.
3. Array parameters are marked by their scalar type name followed by an underscore and `1ARRAY`. For example: `int[]` would be "int_1ARRAY" and `String[]` would be `java_lang_String_1ARRAY`.

JavaScript examples

Java API:

```
public interface JavaScriptNativeApi {
    void print(String str);
}
```

becomes

```
o.print__java_lang_String = function(param1, callback) {
    console.log(param1);
    callback.complete();
}
```

Java API:

```
interface JavaScriptNativeApiAddTwoArgs {
    int add(int a, int b);
}
```

becomes

```
o.add__int_int = function(param1, param2, callback) {
    callback.complete(param1 + param2);
}
```

```
interface JavaScriptNativeApiAddArray {
    int add(int[] a);
}
```

becomes

```
o.add__int_1ARRAY = function(param1, callback) {
    var c = 0, len = param1.length;
    for (var i =0; i<len; i++) {
        c += param1[i];
    }
    callback.complete(c);
}
```

61.5.5. Native GUI components

PeerComponent [https://www.codenameone.com/javadoc/com/codename1/ui/PeerComponent.html] return values are automatically translated to the platform native peer as an expected return value. For example: for a `NativeInterface` method such as this:

```
interface NativePeerProvider {
    PeerComponent createPeer();
}
```

Android native implementation would use:

```
class AndroidPeerImplementation {
    public View createPeer() {
        return null;
    }
}
```

The iphone would need to return a pointer to a view for example:

```
- (UIView*)createPeer;
```



Not all platforms support native peers. Specifically Java SE doesn't support them due to the way the Java SE native interfaces are mapped to their implementation. Note that this won't limit the code from running on an unsupported platform. Only that specific method won't work.

JavaScript would expect a DOM Element (for example: a `<div>` tag to be returned.). For example:

```
o.createHelloComponent_ = function(callback) {
    var c = jQuery('<div>Hello World</div>')
            .css({'background-color' : 'yellow', 'border' : '1px solid blue'});
    callback.complete(c.get(0));
};
```

Notice that if you want to use a native library (jar,.a file etc.) places it within the appropriate native directory and it will be packaged into the final executable. You would be able to reference it from the native code and not from the Codename One code, which means you will need to build native interfaces to access it.

This is discussed further below.

61.5.6. Type mapping & rules

Several rules govern the creation of NativeInterfaces and you covered some of them.

- The implementation class must have a default public constructor or no constructor at all
- Native methods can't throw exceptions, checked or otherwise
- A native method can't have the name `init` as this is a reserved method in Objective-C
- Only the supported types listed below can be used
- Native implementations can't rely on pass by reference/value semantics as those might change between platforms
- `hashCode`, `equals` & `toString` are reserved and won't be mapped to native code

Table 13. NativeInterface Supported Types

Java	Android	Java SE	Obj-C	C#
byte	byte	byte	char	sbyte
boolean	boolean	boolean	BOOL	bool
char	char	char	int	char
short	short	short	short	short
int	int	int	int	int
long	long	long	long	long
float	float	float	float	float
double	double	double	double	double
String	String	String	NSString*	String
byte[]	byte[]	byte[]	NSSData*	sbyte[]
boolean[]	boolean[]	boolean[]	NSData*	bool[]
char[]	char[]	char[]	NSData	char[]
short[]	short[]	short[]	NSData*	short[]

Java	Android	Java SE	Obj-C	C#
int[]	int[]	int[]	NSData*	int[]
long[]	long[]	long[]	NSData*	long[]
float[]	float[]	float[]	NSData*	float[]
double[]	double[]	double[]	NSData*	double[]
PeerComponent	android.view.View	PeerComponent	void*	FrameworkElement



JavaScript is excluded from the table above as it isn't a type safe language and thus has no such type mapping



`PeerComponent` on iOS is `void*` but `UIView` is expected as a result

The examples below show the signatures for this method on all platforms:

Listing 21. NativeInterface definition

```
interface AllTypesNativeApi {
    void test(byte b, boolean boo, char c, short s,
              int i, long l, float f, double d, String ss,
              byte[] ba, boolean[] booa, char[] ca, short[] sa, int[] ia,
              long[] la, float[] fa, double[] da,
              PeerComponent cmp);
}
```

Listing 22. Android Version

```
class AndroidAllTypesImplementation {
    public void test(byte param, boolean param1, char param2,
                     short param3, int param4, long param5, float param6,
                     double param7, String param8, byte[] param9,
                     boolean[] param10, char[] param11, short[] param12,
                     int[] param13, long[] param14, float[] param15,
                     double[] param16, android.view.View param17) {
    }
}
```

Listing 23. iOS Version

```
-(void)test:(char)param param1:(BOOL)param1
    param2:(int)param2 param3:(short)param3 param4:(int)param4
    param5:(long long)param5 param6:(float)param6
    param7:(double)param7 param8:(NSString*)param8
    param9:(NSData*)param9 param10:(NSData*)param10
    param11:(NSData*)param11 param12:(NSData*)param12
    param13:(NSData*)param13 param14:(NSData*)param14
```

```
    param15:(NSData*)param15 param16:(NSData*)param16
    param17:(void*)param17;
```



You had to break lines for the print version, the JavaScript version is a long method name that broke the book!

Listing 24. JavaScript Version

```
o["test__byte_boolean_char_short_int_long_float_double_java_lang_String_byte_1ARRAY_boolean_1ARRAY_char_1ARRAY_short_1ARRAY_int_1ARRAY_long_1ARRAY_float_1ARRAY_double_1ARRAY_com_codename1_ui_PeerComponent"] = function(param1, param2, param3, param4, param5, param6, param7, param8, param9, param10, param11, param12, param13, param14, param15, param16, param17, param18, callback) {
    callback.error(new Error("Not implemented yet"));
};
```

Listing 25. Java SE Version

```
class JavaScriptAllTypesImplementation {
    public void test(byte param, boolean param1, char param2, short param3, int param4,
                     long param5, float param6, double param7, String param8, byte[] param9,
                     boolean[] param10, char[] param11, short[] param12, int[] param13,
                     long[] param14, float[] param15, double[] param16,
                     PeerComponent param17) {
    }
}
```

61.5.7. Android native permissions

Permissions in Codename One are seamless. Codename One traverses the bytecode and automatically assigns permissions to Android applications based on the API's used by the developer.

For example, when accessing native functionality this won't work since native code might require specialized permissions and you don't/can't run any serious analysis on it (it can be about anything).

If you require more permissions in your Android native code you need to define them in the build arguments using `android.permission.<PERMISSION_NAME>=true` for each permission you want to include. A full list of permissions are listed in Android's `Manifest.permission` documentation [https://developer.android.com/reference/android/Manifest.permission.html].

For example:

```
android.permission.ADD_VOICEMAIL=true
android.permission.BATTERY_STATS=true
...
```

You can specify the maximum SDK version in which the permission is needed using the `android.permission.<PERMISSION_NAME>.maxSdkVersion` build hint. You can also specify whether the

permission is **required** for the app to run using the `android.permission.<PERMISSION_NAME>.required` build hint.

For example:

```
android.permission.ADD_VOICEMAIL=true
android.permission.BATTERY_STATS=true
android.permission.ADD_VOICEMAIL.required=false
android.permission.ADD_VOICEMAIL.maxSdkVersion=18
...
```

You can or use the `android.xpermissions` build hint to inject `<uses-permission>` tags into the manifest file. For example:

```
android.xpermissions=<uses-permission android:name="android.permission.READ_CALENDAR" />
```



You need to include the full XML snippet. You can unify many lines into a single line in the GUI as XML allows that.

61.5.8. Native AndroidNativeUtil

If you do any native interfaces programming in Android you should be familiar with the `AndroidNativeUtil` class which allows you to access native device functionality more from the native code. For example: many Android APIs need access to the `Activity` which you can get by calling `AndroidNativeUtil.getActivity()`.

The native util class includes a few other features such as:

- `runOnUiThreadAndBlock(Runnable)` - this is such a common pattern that it was generalized into a public static method. Its identical to `Activity.runOnUiThread` but blocks until the runnable finishes execution.
- `addLifecycleListener/removeLifecycleListener` - These essentially provide you with a callback to lifecycle events: `onCreate` etc. which can be pretty useful for some cases.
- `registerViewRenderer` - PeerComponent [https://www.codenameone.com/javadoc/com/codename1/ui/PeerComponent.html]'s are shown on top of the UI since they're rendered within their own thread outside of the EDT cycle. When you need to show a Dialog [https://www.codenameone.com/javadoc/com/codename1/ui/Dialog.html] on top of the peer you grab a screenshot of the peer, hide it and then show the dialog with the image as the background (the same applies for transitions). Some components (specifically the MapView) might not render and require custom code to implement the transferal to a native Bitmap, this API allows you to do that.

You can work with `AndroidNativeUtil` using native code such as this:

```
class NativeCallsImpl {
     public void nativeMethod() {
        AndroidNativeUtil.getActivity().runOnUiThread(new Runnable() {
```

```
            public void run() {
                // ...
            }
        });
    }
    // ...
}
```

61.5.9. Broadcast receiver

A common way to implement features in Android is the `BroadcastReceiver` API. This allows intercepting operating system events for common use cases.

A good example is intercepting incoming SMS which is specific to Android so you'd need a broadcast receiver to implement that. This is often confusing to developers who sometimes derive the impl class from broadcast receiver. That's a mistake...

The solution is to place any native Android class into the `native/android` directory. It will get compiled with the rest of the native code and "works." You can therefore place this class under `native/android/com/codename1/sms/intercept`:

```
public class SMSListener extends BroadcastReceiver {

    @Override
    public void onReceive(Context cntxt, Intent intent) {
        if (intent.getAction().equals("android.provider.Telephony.SMS_RECEIVED")) {
            Bundle bundle = intent.getExtras();
            SmsMessage[] msgs = null;
            if (bundle != null) {
                try {
                    Object[] pdus = (Object[]) bundle.get("pdus");
                    msgs = new SmsMessage[pdus.length];
                    for (int i = 0; i < msgs.length; i++) {
                        msgs[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);
                        String msgBody = msgs[i].getMessageBody();
                        SMSCallback.smsReceived(msgBody);
                    }
                } catch (Exception e) {
                    Log.e(e);
                    SMSCallback.smsReceiveError(e);
                }
            }
        }
    }
}
```

The code above is pretty standard native Android code, it's a callback in which most of the logic is like the native Android code mentioned in this stackoverflow question [https://stackoverflow.com/questions/39526138/broadcast-receiver-for-receive-sms-is-not-working-when-declared-in-manifeststat].

However, there is still more you need to do. To implement this natively you need to register the permission and the receiver in the `manifest.xml` file as explained in that question. This is how their native manifest looked:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.bulsy.smstalk1">
    <uses-permission android:name="android.permission.RECEIVE_SMS" />
    <uses-permission android:name="android.permission.READ_SMS" />
    <uses-permission android:name="android.permission.SEND_SMS"/>
    <uses-permission android:name="android.permission.READ_CONTACTS" />

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <receiver android:name="com.bulsy.smstalk1.SmsListener"
               android:enabled="true"
               android:permission="android.permission.BROADCAST_SMS"
               android:exported="true">
            <intent-filter android:priority="2147483647">//this doesnt work
                <category android:name="android.intent.category.DEFAULT" />
                <action android:name="android.provider.Telephony.SMS_RECEIVED" />
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

You need the broadcast permission XML and the permission XML. Both are doable through the build hints. The former is pretty easy:

```
android.xpermissions=<uses-permission android:name="android.permission.RECEIVE_SMS" />
```

The latter isn't much harder; notice that many lines have been merged into a single line for convenience:

```
android.xapplication=<receiver android:name="com.codename1.sms.intercept.SMSListener"  android:enabled="true"
android:permission="android.permission.BROADCAST_SMS"  android:exported="true">                    <intent-filter
android:priority="2147483647"><category android:name="android.intent.category.DEFAULT" />        <action
android:name="android.provider.Telephony.SMS_RECEIVED" />                </intent-filter>             </receiver>
```

Here it's formatted for readability:

```
<receiver android:name="com.codename1.sms.intercept.SMSListener"
              android:enabled="true"
              android:permission="android.permission.BROADCAST_SMS"
              android:exported="true">
                   <intent-filter android:priority="2147483647">
                          <category android:name="android.intent.category.DEFAULT" />
                          <action android:name="android.provider.Telephony.SMS_RECEIVED" />
                   </intent-filter>
</receiver>
```

Listening & permissions

You will notice that these don't include the actual binding or permission prompts you would expect for something like this. To do this you need a native interface.

The native sample in stack overflow bound the listener in the activity but here you want the app code to decide when you should bind the listening:

```
public interface NativeSMSInterceptor extends NativeInterface {
    void bindSMSListener();
    void unbindSMSListener();
}
```

That's easy!

Notice that `isSupported()` returns false for all other OSes so you won't need to ask whether this is "Android" you can use `isSupported()`.

The implementation is pretty easy too:

```
public class NativeSMSInterceptorImpl {
    private SMSListener smsListener;
    public void bindSMSListener() {
        if (AndroidNativeUtil.checkForPermission(Manifest.permission.RECEIVE_SMS, "We can automatically enter the SMS
code for you")) { ①
            smsListener = new SMSListener();
            IntentFilter filter = new IntentFilter();
            filter.addAction("android.provider.Telephony.SMS_RECEIVED");
            AndroidNativeUtil.getActivity().registerReceiver(smsListener, filter); ②
        }
    }

    public void unbindSMSListener() {
        AndroidNativeUtil.getActivity().unregisterReceiver(smsListener);
    }

    public boolean isSupported() {
        return true;
    }
}
```

① This will trigger the permission prompt on Android 6 and newer. Even though the permission is declared in XML this isn't enough for 6+. Notice that even when you run on Android 6 you still need to declare permissions in XML!

② Here you actually bind the listener, this allows you to grab one SMS and not listen in on every SMS coming through

61.5.10. Native code callbacks

Native interfaces standardize the invocation of native code from Codename One, but it doesn't standardize the reverse of callbacks into Codename One Java code. The reverse is more complicated since its platform specific and more error prone.

A common "trick" for calling back is to define a static method and then trigger it from native code. This works for Android & Java SE since those platforms use Java for their "native code." Mapping this to iOS requires some basic understanding of how the iOS VM works.

For this explanation lets pretend you have a class called NativeCallback in the src hierarchy under the package `com.mycompany` that has the method: `public static void callback()`:

```
public class NativeCallback {
    public static void callback() {
        // do stuff
    }

    public static void callback(int arg) {
        // do stuff
    }

    public static void callback(String arg) {
        // do stuff
    }

    public static int callbackR(int arg) {
        // do stuff
        return arg;
    }
}
```

If you want to call it from Android or all the Java based platforms you can write this in the "native" code:

```
public class NativeCallbackUsage {
    public void invokeCallbacks() {
        NativeCallback.callback();
        NativeCallback.callback("My Arg");
    }
}
```

You can also pass an argument as you do later on:

```
public class NativeCallbackUsage {
    public void invokeCallbacks() {
        NativeCallback.callback();
        NativeCallback.callback("My Arg");
    }
}
```

Accessing callbacks from Objective-C

If you want to invoke that method from Objective-C you need to do the following.

Add an include statement as such:

```
#include "com_mycompany_NativeCallback.h"
#include "CodenameOne_GLViewController.h"
```

Notice that the `CodenameOne_GLViewController.h` include defines various macros such as `CN1_THREAD_STATE_PASS_SINGLE_ARG`.

Then when you want to trigger the method do:

```
com_mycompany_NativeCallback_callback__(CN1_THREAD_STATE_PASS_SINGLE_ARG);
```



For most callbacks you should use the macro `CN1_THREAD_GET_STATE_PASS_SINGLE_ARG` instead of `CN1_THREAD_STATE_PASS_SINGLE_ARG` also make sure to add `#include "cn1_globals.h" in the file

The VM passes the thread context along method calls to save on API calls (thread context is used in Java for synchronization, gc and more).

You can pass arguments like:

```
public static void callback(int arg) {
    // do stuff
}
```

Which maps to native as (notice the extra _ before the int):

```
com_mycompany_NativeCallback_callback___int(CN1_THREAD_GET_STATE_PASS_ARG intValue);
```

Notice that there is no comma between the CN1_THREAD_GET_STATE_PASS_ARG and the value!

Why No Comma?

The comma is included as part of the macro which makes for code that isn't as readable.

The reason for this dates to the migration from XMLVM [1] to the current ParparVM implementation. CN1_THREAD_GET_STATE_PASS_ARG is defined as nothing in XMLVM since it didn't use that concept. Yet under ParparVM it will include the necessary comma.

A common use case is passing string values to the Java side, or NSString* which is iOS equivalent. Assuming a method like this:

```
public static void callback(String arg) {
    // do stuff
}
```

You would need to convert the `NSString*` value you already have to a `java.lang.String` which the callback expects.

The `fromNSString` function also needs this special argument so you will need to change the method as such:

```
com_mycompany_NativeCallback_callback___java_lang_String(CN1_THREAD_GET_STATE_PASS_ARG fromNSString
(CN1_THREAD_GET_STATE_PASS_ARG nsStringValue));
```

And you might want to return a value from callback as such:

```
public static int callbackR(int arg) {
    // do stuff
    return arg;
}
```

This is tricky since the method name changes to support covariant return types and so the signature would be:

```
com_mycompany_NativeCallback_callback___int_R_int(intValue);
```

The upper case R allows you to differentiate between void `callback(int,int)` and `int callback(int)`.



Covariant return types are a little known Java 5 feature. For example: the method `Object getX()` can be overridden by `MyObject getX()`. For example, in the VM level they can both exist side by side.

Accessing callbacks from JavaScript

The mechanism for invoking static callback methods from JavaScript (for the JavaScript port) is like Objective-C's. The `this` object in your native interface method contains a property named `$GLOBAL$` that provides access to static Java methods. This object will contain JavaScript mirror objects for each Java class (though the property name is mangled by replacing "." with underscores). Each mirror object contains a wrapper method for its underlying class's static methods where the method name follows the same naming convention as is used for the JavaScript native methods themselves (and like the naming conventions used in Objective-C).

For example, the Google Maps project includes the static callback method:

```
class MapContainerCallbacks {
    static void fireMapChangeEvent(int mapId, final int zoom, final double lat, final double lon) {
        // ...
    }
}
```

It's defined in the `com.codename1.googlemaps.MapContainer` class.

This method is called from JavaScript inside a native interface using the following code:

```
var fireMapChangeEvent = this.$GLOBAL$.com_codename1_googlemaps_MapContainer.fireMapChangeEvent__int_int_double_double;
google.maps.event.addListener(this.map, 'bounds_changed', function() {
    fireMapChangeEvent(self.mapId, self.map.getZoom(), self.map.getCenter().lat(), self.map.getCenter().lng());
});
```

In this example you first get a reference to the `fireMapChangeEvent` method, and then call it later. For example, you could have called it directly also.



Your code **MUST** contain the full string path `this.$GLOBAL$.your_class_name.your_method_name` or the build server won't be able to recognize that your code requires this method. The `$GLOBAL$` object is populated by the build server with those classes and methods that are used inside your native methods. If the build server doesn't recognize that the methods are being used (through this pattern) it won't generate the necessary wrappers for your JavaScript code to access the Java methods.

Callbacks of the SMS receiver

The SMS Broadcast Receiver code from before also used callbacks such as this:

```
package com.codename1.sms.intercept; ①

import com.codename1.util.FailureCallback;
import com.codename1.util.SuccessCallback;
import static com.codename1.ui.CN.*;

class SMSCallback {
```

```
    static SuccessCallback<String> onSuccess;
    static FailureCallback onFail;

    public static void smsReceived(String sms) {
        if (onSuccess != null) {
            SuccessCallback<String> s = onSuccess;
            onSuccess = null;
            onFail = null;
            SMSInterceptor.unbindListener();
            callSerially(() -> s.onSucess(sms)); ②
        }
    }

    public static void smsReceiveError(Exception err) {
        if (onFail != null) {
            FailureCallback f = onFail;
            onFail = null;
            SMSInterceptor.unbindListener();
            onSuccess = null;
            callSerially(() -> f.onError(null, err, 1, err.toString()));
        } else {
            if (onSuccess != null) {
                SMSInterceptor.unbindListener();
                onSuccess = null;
            }
        }
    }
}
```

① Notice that the package is the same as the native code and the other classes. This allows the callback class to be package protected so it isn't exposed through the API (the class doesn't have the public modifier)

② You wrap the callback in call serially to match the Codename One convention of using the EDT by default. The call will probably arrive on the Android native thread so it makes sense to normalize it and not expose the Android native thread to the user code

Asynchronous callbacks & threading

One of the problematic aspects of calling back into Java from JavaScript is that JavaScript has no notion of multi-threading. If the method you're calling uses Java's threads at all (for example: It includes a `wait()`, `notify()`, `sleep()`, `callSerially()`, etc.) you need to call it asynchronously from JavaScript. You can call a method asynchronously by appending `$async` to the method name. For example: With the Google Maps example above, you would change :

```
```

to

```
var asyncFireMapChangeEvent =
```

```
this.$GLOBAL$.com_codename1_googlemaps_MapContainer.fireMapChangeEvent__int_int_double_double$async;
```

This will cause the call to be wrapped in the appropriate bootstrap code to work with threads - and it's necessary in cases where the method **may** use threads of any kind. The side-effect of calling a method with the `$async` suffix is that you can't use return values from the method.



Usually you should use the **async** version of a method when calling it from your native method. Only use the synchronous (default) version if you're sure that the method doesn't use any threading primitives.

61.6. Libraries - cn1lib

Support for JAR files in Codename One has been a source of confusion so its probably a good idea to revisit this subject again and clarify all the details.

The first source of confusion is changing the classpath. You should NEVER change the classpath or add an external JAR through the IDE classpath UI. The reasoning here is simple, these IDEs don't package the JARs into the final executable and even if they did these JARs would probably use features unavailable or inappropriate for the device (for example: `java.io.File` etc.).

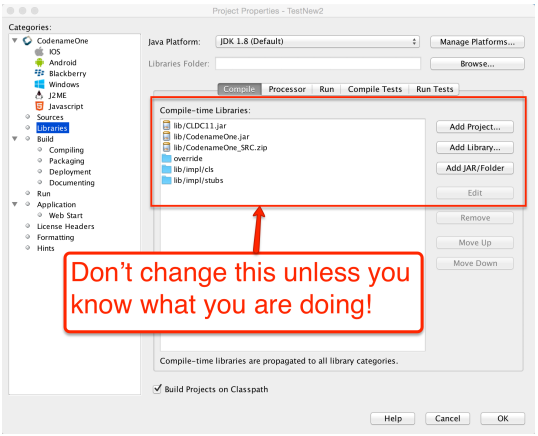


Figure 335. Don't change the classpath, this is how it should look for a typical Java 8 Codename One application

Cn1libs are Codename One's file format for 3rd party extensions. It's physically a zip file containing other zip files and some meta-data.

61.6.1. Why not use JAR

A jar can be compiled with usage of any Java API that might not be supported, it can be compiled with a Java target version that isn't tested.

Jars don't include support for writing native code; you could use JNI in jars but that doesn't match Codename One's needs for native support (see section above).

Jars don't support "proper" code completion, a common developer trick is to stick source code into the jar but that prevents usage with proprietary code. Cn1libs provide full IDE code completion (with JavaDoc hints) without exposing the sources.

There are two use cases for wanting JARs and they both have different solutions:

1. Modularity
2. Working with an existing JARs

Cn1lib's address the modularity aspect allowing you to break that down. Existing jars can sometimes be used native code settings but you would want to adapt the code to abide by Codename One restrictions.

61.6.2. How to use cn1libs

Codename One has a large repository of 3rd party cn1libs [https://www.codenameone.com/cn1libs.html]. With the Maven build tool, you consume a cn1lib the same way you consume any other Maven dependency: by adding it to the `<dependencies>` section of your application's `common/pom.xml` file. A typical entry looks like:

```
<dependency>
    <groupId>com.example</groupId>
    <artifactId>mylib-lib</artifactId>
    <version>1.0.0</version>
    <type>pom</type>
</dependency>
```

Add the dependency **before** the `<!-- INJECT DEPENDENCIES -->` comment in `common/pom.xml`. The `-lib` suffix on the `artifactId` and the `<type>pom</type>` declaration are both important - they tell Maven to pull in the library's "lib" aggregator module rather than a plain jar. See Section 1.5, "Codename One libraries" for the full library-consumption guide.



Some cn1libs require additional configuration such as build hints. Always read the library author's instructions when integrating a 3rd party library.

Legacy Ant projects

If you're still maintaining an Ant-based project, you can install a cn1lib by dropping its `.cn1lib` file into the project's `lib` directory and then right-clicking the project and selecting Codename One → Refresh cn1lib files (which invokes the `refresh-libs` Ant task). This unpacks classes and stub sources into `lib/impl/cls` and `lib/impl/stubs`, and native sources into `lib/impl/native`. New projects should use Maven instead - see Chapter 1, *Maven Project Workflow*.

61.6.3. Creating a simple cn1lib

Codename One libraries are generated from the `cn1lib-archetype` Maven archetype, which creates a multi-module project containing a `common` module for your cross-platform Java sources and one module per native platform (iOS, Android, Java SE, JavaScript) for any native interfaces. The detailed walkthrough - command-line, IntelliJ IDEA, NetBeans, and Eclipse - lives in Section 1.5,

“Codename One libraries”.

The quickest path from a shell:

```
mvn archetype:generate \
  -DarchetypeArtifactId=cn1lib-archetype \
  -DarchetypeGroupId=com.codenameone \
  -DarchetypeVersion=LATEST \
  -DgroupId=com.example.mylib \
  -DartifactId=mylib \
  -Dversion=1.0-SNAPSHOT \
  -DinteractiveMode=false
```

Once the project is generated, add your cross-platform Java sources under `common/src/main/java`, CSS under `common/src/main/css`, and run `mvn install` from the project root to build. The build produces both a Maven artifact for pom-style consumption and a legacy `.cn1lib` file in `common/target` for users still on Ant.

61.6.4. Build hints in cn1libs

Some cn1libs are pretty simple to install - just add them as a Maven dependency in `common/pom.xml`. For example, many of the more elaborate cn1libs need some pretty complex configurations. This is the case when native code is involved where you need to add permissions or plist entries for the various native platforms to get everything to work. This makes the cn1lib’s helpful but less than seamless which is where you want to go.

Codename One cn1libs include two files that can be placed into the root: `codenameone_library_required.properties` & `codenameone_library_appended.properties`.

In these files you can write a build hint as `codename1.arg.ios.plistInject=…` for the various hints.



Notice the usage of the properties syntax for the build hint with the `codename1.arg` prefix you would also need to escape reserved characters for properties files.
The best way to discover the right syntax for such build hints is to set them through the build hints GUI in a regular project and copy/paste them from `codenameone_settings.properties` into the cn1lib file.

The obvious question is why do you need two files?

There are two types of build hints: required and appended.

Required build hints can be something like `ios.objC=true` which you want to always work. For example: if a cn1lib defines `ios.objC=true` and another cn1lib defines `ios.objC=false` things won’t work since one cn1lib won’t get what it needs…
In this case you’d want the build to fail so you can remove the faulty cn1lib.



If two cn1libs define `ios.objC=true` there will be no collision as the value would be identical

An appended property would be something like `ios.plistInject=<key>UIBackgroundModes</key><array><string>audio</string> </array>`

Notice that this can still collide for example: if a different cn1lib defines its own background mode. For example, there are many valid cases where `ios.plistInject` can be used for other things. In this case you will append the content of the `ios.plistInject` into the build hint if it isn't already there.

Keep these things in mind:

- Properties are merged with every "refresh libs" call not dynamically on the server. This means it should be pretty simple for the developer to investigate issues in this process.
- Changing flags is problematic - there is no "uninstall" process. Since the data is copied into the `codenameone_settings.properties` file. If you need to change a flag later on you might need to alert users to make changes to their properties essentially negating the value of this feature... be careful when adding properties here.

It's your responsibility as a library developer to decide which build hint goes into which file! Codename One can't automate this process as the whole process of build hints is by definition an ad hoc process.

The rule of thumb is that a build hint with a numeric or boolean value is always a required property. If an entry has a string that you can append with another string then its probably an appended entry.

These build hints are probably of the "required" type:

```
android.debug
android.release
android.installLocation
android.licenseKey
android.stack_size
android.statusbar_hidden
android.googleAdUnitId
android.includeGPlayServices
android.headphoneCallback
android.gpsPermission
android.asyncPaint
android.supportV4
android.theme
android.cusom_layout1
android.versionCode
android.captureRecord
android.removeBasePermissions
android.blockExternalStoragePermission
android.min_sdk_version
android.smallScreens
android.streamMode
android.enableProguard
android.targetSDKVersion
android.web_loading_hidden
```

```
facebook.appId
facebook.clientToken
ios.keyboardOpen
ios.project_type
ios.newStorageLocation
ios.prerendered_icon
ios.application_exits
ios.themeMode
ios.xcode_version
javascript.inject_proxy
javascript.minifying
javascript.proxy.url
javascript.sourceFilesCopied
javascript.teavm.version
google.adUnitId
ios.includePush
ios.headphoneCallback
ios.enableAutoplayVideo
ios.googleAdUnitId
ios.googleAdUnitIdPadding
ios.enableBadgeClear
ios.locationUsageDescription
ios.bundleVersion
ios.objC
ios.testFlight
ios.metal
desktop.width
desktop.height
desktop.adaptToRetina
desktop.resizable
desktop.fontSizes
desktop.theme
desktop.themeMac
desktop.themeWin
desktop.windowsOutput
noExtraResources
android.permission.<PERMISSION_NAME>
```

These build hints should probably be appended:

```
android.xapplication
android.xpermissions
android.xintent_filter
android.facebook_permissions
android.stringsXml
android.style
android.nonconsumable
android.xapplication_attr
android.xactivity
android.pushVibratePattern
```

```
android.proguardKeep
android.sharedUserId
android.sharedUserLabel
ios.urlScheme
ios.interface_orientation
ios.plistInject
ios.facebook_permissions
ios.applicationDidEnterBackground
ios.viewDidLoad
ios.glAppDelegateHeader
ios.glAppDelegateBody
ios.beforeFinishLaunching
ios.afterFinishLaunching
ios.add_libs
```

cn1lib Structure/File Format

The cb1lib file format is quite simple, it's a zip file containing zip files within it with fixed names to support the various features.

The table below covers the files that can/should be a part of a cn1lib file:

Table 14. cn1lib structure

File Name	Required	Purpose
main.zip	✔	Contains the bytecode and the library binary data. This is effectively the portable portion of the jar
stubs.zip	✔	Stub source files (autogenerated) containing javadocs to provide code completion
manifest.properties	×	General properties of the library, this isn't used for much at the moment
codenameone_ library_ appended.properties	×	Discussed above
codenameone_ library_ required.properties	×	Discussed above
nativeios.zip	×	Native iOS sources if applicable
nativeand.zip	×	Native Android sources if applicable
nativejavascript.zip	×	Native JavaScript sources if applicable
nativese.zip	×	Native Java SE sources if applicable
nativewin.zip	×	Native Windows sources if applicable
nativeme.zip	×	Native Java ME sources if applicable

61.7. Integrating Android 3rd party libraries & JNI

While its pretty easy to use native interfaces to write Android native code some things aren't necessarily as obvious. For example: if you want to integrate a 3rd party library, specifically one that includes native C JNI code this process isn't as straightforward.

If you need to integrate such a library into your native calls you have the following options:

1. The first option (and the easiest one) is to place a Jar file in the native/android directory. This will link your binary with the jar file. Just place the jar under the native/android and the build server will pick it up and will add it to the classpath.
   Notice that Android release apps are obfuscated by default which might cause issues with such libraries if they reference APIs that are unavailable on Android. You can work around this by adding a build hint to the proguard obfuscation code that blocs the obfuscation of the problematic classes using the build hint:
   `android.proguardKeep=-keep class com.mypackage.ProblemClass { *; }`
2. Another option is the `aar` file is a binary format Google introduced to represent an Android Library project (similarly to the cn1lib format). One of the problem with the Android Library projects was the fact that it required the project sources which made it difficult for 3rd party vendors to publish libraries.
   As a result so android introduced the `aar` file which is a binary format that represents a Library project. To learn more about arr you can read this [https://developer.android.com/studio/projects/android-library.html#aar-contents].

   You can link an `aar` file by placing it under the native/android and the build server will link it to the project.
3. Another **obsolete approach** exists that you're mentioning for legacy purposes (for example: if you need to port code written with this legacy option). This predated the `aar` option from Google... Not all 3rd party tools can be packaged as a simple jar, some 3rd party tools need to declare activities add permissions, resources, assets, and/or even add native code (`.so` files).
   To link a Library project to your Codename One project open the Library project in Eclipse or Android Studio and make sure the project builds, after the project was built successfully remove the bin directory from the project and zip the whole project.

   Rename the extension from `.zip` to `.andlib` and place the andlib file under the `native/android` directory. The build server will pick it up and will link it to the project.

61.8. Drag & drop

Unlike other platforms that tried to create overly generic catch all APIs Codename One tried to make things as simple as possible.

In Codename One components can be dragged and drop targets are always components. The logic of actually performing the operation indicated by the drop is the responsibility of the person implementing the drop.



Some platforms for example: AWT allow dragging abstract concepts such as mime

> type elements. This allows dragging things like a text file into the app, but that use case isn't realistic in mobile

The code below allows you to rearrange the items based on a sensible order. Notice it relies on the default `Container` drop behavior:

```
public class RearrangeableItemsDemo {
    public void show() {
        Form hi = new Form("Rearrangeable Items", new BorderLayout());
        String[] buttons = {"A Game of Thrones", "A Clash Of Kings",  "A Storm Of Swords",
            "A Feast For Crows", "A Dance With Dragons", "The Winds of Winter", "A Dream of Spring" };

        Container box = new Container(BoxLayout.y());
        box.setScrollableY(true);
        box.setDropTarget(true);
        java.util.List<String> got = Arrays.asList(buttons);
        Collections.shuffle(got);
        for (String current : got) {
            MultiButton mb = new MultiButton(current);
            box.add(mb);
            mb.setDraggable(true);
        }

        hi.add(BorderLayout.NORTH, "Arrange The Titles").add(BorderLayout.CENTER, box);
        hi.show();
    }
}
```

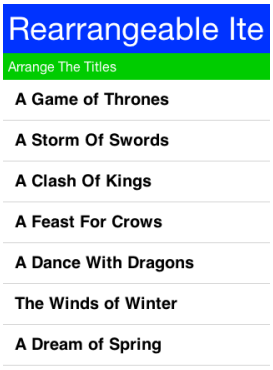


Figure 336. Drag demo

To enable dragging a component it must be flagged as draggable using `setDraggable(true)`, to allow dropping the component onto another component you must first enable the drop target with `setDropTarget(true)` and override some methods (more on that later).

When dragging on top of a child component of a drop target the code recursively searches for a drop target parent. Dropping a component on the child will automatically find the right drop target, hence there is no need to make "everything" into a drop target.

You can override these methods in the draggable components:

- `getDragImage` - this generates an image preview of the component that will be dragged. This automatically generates a sensible default so you don't need to override it.

- `drawDraggedImage` - this method will be invoked to draw the dragged image at a given location, it might be useful to override it if you want to display some drag related information such as an icon based on location etc. (for example: a move/copy icon).

In the drop target you can override the following methods:

- `draggingOver` - returns true if a drop operation at this point is permitted. Otherwise releasing the component will have no effect.
- `dragEnter/Exit` - useful to track and cleanup state related to dragging over a specific component.
- `drop` - the logic for dropping/moving the component must be implemented here!

61.9. Android Lollipop ActionBar customization

When running on Android Lollipop (5.0 or newer) the native action bar will use the Lollipop design. This isn't applicable if you use the Toolbar [https://www.codenameone.com/javadoc/com/codename1/ui/Toolbar.html] or SideMenuBar [https://www.codenameone.com/javadoc/com/codename1/ui/SideMenuBar.html] this will be used in the task switcher.

To customize the colors of the native `ActionBar` on Lollipop define a `colors.xml` file in the `native/android` directory of your project. It should look like this:

```
<resources>
   <color name="colorPrimary">#ff00ff00</color>
   <color name="colorPrimaryDark">#80ff0000</color>
   <color name="colorAccent">#800000ff</color>
</resources>
```

61.10. Intercepting URLs on iOS & Android

A common trick in mobile application development, is communication between two unrelated applications.

In Android you can use intents which are pretty elaborate and can be used through `Display.execute`, but what if you would like to expose the functionality of your application to a different application running on the device. This would allow that application to launch your application.

This isn't something built into Codename One, but it does expose enough of the platform capabilities to enable that functionality rather on Android.

On Android you need to define an intent filter which you can do using the `android.xintent_filter` build hint, this accepts the XML to filter whether a request is relevant to your application:

```
android.xintent_filter=<intent-filter>   <action android:name="android.intent.action.VIEW" />    <category
android:name="android.intent.category.DEFAULT" />    <category android:name="android.intent.category.BROWSABLE" />
<data android:scheme="myapp" />  </intent-filter>
```

You can read more about it in this stack overflow question [http://stackoverflow.com/questions/11421048/android-ios-custom-uri-protocol-handling].

To bind the `myapp://` URL to your application. As a result typing `myapp://x` into the Android browser will launch the application.

61.10.1. Passing launch arguments to the app

You can access the value of the URL that launched the app using:

```
public class RearrangeableItemsDemo {
    public void show() {
        Form hi = new Form("Rearrangeable Items", new BorderLayout());
        String[] buttons = {"A Game of Thrones", "A Clash Of Kings",  "A Storm Of Swords",
            "A Feast For Crows", "A Dance With Dragons", "The Winds of Winter", "A Dream of Spring" };

        Container box = new Container(BoxLayout.y());
        box.setScrollableY(true);
        box.setDropTarget(true);
        java.util.List<String> got = Arrays.asList(buttons);
        Collections.shuffle(got);
        for (String current : got) {
            MultiButton mb = new MultiButton(current);
            box.add(mb);
            mb.setDraggable(true);
        }

        hi.add(BorderLayout.NORTH, "Arrange The Titles").add(BorderLayout.CENTER, box);
        hi.show();
    }
}
```

This value would be null if the app was launched through the icon.

61.10.2. Android App-to-App contracts (Wallet/Issuer style flows)

Some integrations (wallet provisioning, issuer verification, secure step-up) need more than a URL launch. They require:

- a custom Android intent action,
- caller validation,
- asynchronous app logic (login/back end verification),
- a native `setResult(…)` response back to the caller.

In Codename One, keep this flow in two layers:

1. **Android bridge activity** for contract lifecycle and `setResult(…)`.
2. **Codename One app code** for business logic in `start()`.

Why this isn't implicitly generated by AndroidGradleBuilder

A generic bridge can't infer:

- expected caller package(s),
- required result extra names,
- approval/decline mapping rules,
- timeout/failure policy.

Those are contract-specific, so the bridge activity is intentionally app-defined.

Registering the bridge activity

Use `android.xapplication` to inject a dedicated bridge activity and bind the action it should receive:

```
codename1.arg.android.xapplication=<activity android:name="com.example.wallet.WalletBridgeActivity"
android:exported="true" android:launchMode="singleTop" android:noHistory="true" android:excludeFromRecents="true"
android:theme="@android:style/Theme.Translucent.NoTitleBar"><intent-filter><action
android:name="com.example.wallet.ACTION_VERIFY"/><category android:name="android.intent.category.DEFAULT"/></intent-
filter></activity>
```

Keep `codename1.arg.android.activity.launchMode=singleTask` on your main CN1 activity so app resume behavior is predictable.

Bridge activity responsibilities

A minimal bridge activity should:

1. Receive and parse the incoming intent.
2. Capture caller identity using `getCallingPackage()`.
3. Launch or resume the CN1 app activity.
4. Hold contract state until CN1 completes verification.
5. Return `setResult(…)` and `finish()`.



For payment/wallet trust decisions, treat only `getCallingPackage()` as verified identity. Don't rely on referrer extras for caller authentication because callers can spoof those values. This implies the contract should be launched with `startActivityForResult(…)` so caller identity is available.

Wallet Contract Security Guidelines

Use this checklist when the integration can activate payment credentials or accounts:

1. **Fail closed on missing verified caller**. If `android.intent.caller` is empty (or `android.intent.caller.verified` isn't `true`), reject the request and return a failure/cancel result.
   Where: CN1 `start()` guard before any business logic.

2. **Use an allow-list** of trusted caller package names and reject everything else.
   Where: CN1 `start()` (or native bridge) before network calls.
3. **Verify signing certificate fingerprints natively** for each allowed package. Package name checks alone aren't enough in high-trust wallet scenarios.
   Where: native Android code (bridge activity/native interface), because this requires Android `PackageManager` APIs.
4. **Check payload freshness** (nonce/timestamp/challenge) before approving.
   Where: back end/API service as the source of truth.
5. **Bind response to request** (request ID / correlation ID) and reject mismatched or replayed responses.
   Where: back end first, then enforce in CN1/native completion flow.
6. **Time-box bridge activity state** and fail closed on timeout, process death, or missing bridge instance.
   Where: native Android bridge + CN1 timeout handling.
7. **Audit log** caller package, request ID, decision, and failure reason (without storing full PAN/token data).
   Where: back end and security telemetry pipeline.

Optional native Android example for certificate fingerprint verification (called from bridge/native layer before trusting caller):

```
public static boolean isTrustedCallerSignature(Context ctx, String packageName, Set<String> allowedSha256) {
    if (ctx == null || packageName == null || packageName.isEmpty() || allowedSha256 == null || allowedSha256.isEmpty())
{
        return false;
    }
    try {
        PackageManager pm = ctx.getPackageManager();
        PackageInfo info;
        if (Build.VERSION.SDK_INT >= 28) {
            info = pm.getPackageInfo(packageName, PackageManager.GET_SIGNING_CERTIFICATES);
            Signature[] sigs = info.signingInfo != null
                    ? info.signingInfo.getApkContentsSigners()
                    : null;
            return hasAllowedFingerprint(sigs, allowedSha256);
        } else {
            info = pm.getPackageInfo(packageName, PackageManager.GET_SIGNATURES);
            return hasAllowedFingerprint(info.signatures, allowedSha256);
        }
    } catch (PackageManager.NameNotFoundException | NoSuchAlgorithmException ex) {
        return false;
    }
}

private static boolean hasAllowedFingerprint(Signature[] sigs, Set<String> allowedSha256)
        throws NoSuchAlgorithmException {
    if (sigs == null) {
        return false;
    }
    MessageDigest md = MessageDigest.getInstance("SHA-256");
    for (Signature s : sigs) {
        String fp = toHex(md.digest(s.toByteArray()));
        if (allowedSha256.contains(fp)) {
            return true;
```

```
        }
    }
    return false;
}

private static String toHex(byte[] data) {
    char[] digits = "0123456789ABCDEF".toCharArray();
    StringBuilder out = new StringBuilder(data.length * 2);
    for (byte b : data) {
        int value = b & 0xff;
        out.append(digits[value >>> 4]);
        out.append(digits[value & 0x0f]);
    }
    return out.toString();
}
```

```
public class WalletBridgeActivity extends Activity {
    private static WeakReference<WalletBridgeActivity> active;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        active = new WeakReference<>(this);

        Intent in = getIntent();
        String caller = getCallingPackage();
        String payload = in == null ? null : in.getStringExtra("payload");

        Intent launch = getPackageManager().getLaunchIntentForPackage(getPackageName());
        if (launch == null) {
            finish();
            return;
        }
        launch.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK | Intent.FLAG_ACTIVITY_CLEAR_TOP | Intent.
FLAG_ACTIVITY_SINGLE_TOP);
        launch.putExtra("wallet_payload", payload);
        launch.putExtra("wallet_caller", caller == null ? "" : caller);
        startActivity(launch);
        // Do not finish yet. CN1 will finish this activity via native callback once verification completes.
    }

    @Override
    protected void onDestroy() {
        WalletBridgeActivity current = getActive();
        if (current == this) {
            active = null;
        }
        super.onDestroy();
    }

    public static WalletBridgeActivity getActive() {
        return active == null ? null : active.get();
    }
}
```

Reading contract data in `start()`

Codename One exposes launch metadata through `AppArg` and `Display` properties. Read these in `start()`:

- `AppArg` (primary launch payload)

- android.intent.action
- android.intent.data
- android.intent.type
- android.intent.caller
- android.intent.caller.verified
- android.intent.extra.<KEY>

```
public void start() {
    Display d = Display.getInstance();
    String action = d.getProperty("android.intent.action", null);
    if ("com.example.wallet.ACTION_VERIFY".equals(action)) {
        boolean callerVerified = "true".equals(d.getProperty("android.intent.caller.verified", "false"));
        String payload = d.getProperty("android.intent.extra.wallet_payload", d.getProperty("AppArg", null));
        String caller = d.getProperty("android.intent.caller", "");

        if (!callerVerified || !isAllowedCaller(caller)) {
            failClosed(); // return declined/canceled via bridge completion API
            return;
        }

        verifyWalletPayload(payload, caller);
        return;
    }
    showMainForm();
}
```

Returning results to the wallet API

Implement a small native interface method (Android implementation) that:

1. gets `WalletBridgeActivity.getActive()`,
2. verifies `getActive()` is non-null and fails closed if the bridge timed out/canceled,
3. builds caller-specific result extras,
4. calls `setResult(Activity.RESULT_OK, resultIntent)`,
5. calls `finish()` on the bridge activity.

This is where you bind to the wallet contract's exact response schema. Keep these extra names in one place so upgrades are manageable.

Foreground behavior on Android 14/15

Foreground transitions are best-effort because Android may block background launches in some states. Build for:

- immediate resume when allowed,
- notification fallback when blocked.

iOS is practically identical to Android with some small caveats, iOS's equivalent of the manifest is

the plist.

You can inject more data into the plist by using the `ios.plistInject` build hint.

The equivalent on the iOS side would be:

```
ios.plistInject=<key>CFBundleURLTypes</key>     <array>         <dict>             <key>CFBundleURLName</key>
<string>com.yourcompany.myapp</string>         </dict>         <dict>             <key>CFBundleURLSchemes</key>
<array>                 <string>myapp</string>             </array>         </dict>     </array>
```

For example, that can conflict with the Facebook integration if you use `FacebookConnect` which needs access to the schemes. To work around this you can use the build hint `ios.urlScheme` for example:

```
ios.urlScheme=<string>myapp</string>
```

61.11. Native peer components

Many Codename One developers don't truly grasp the reason for the separation between peer (native) components and Codename One components. This is a crucial thing you need to understand if you plan on working with native widgets for example: Web Browser, native maps, text input, media and native interfaces (which can return a `PeerComponent`).

Codename One draws all its widgets on its own, this is a concept which was modeled in part after Swing. This allows functionality that can't be achieved in native widget platforms:

1. The Codename One GUI builder & simulator are almost identical to the device - notice that this also enables the build cloud, otherwise device specific bugs would overwhelm development and make the build cloud redundant.
2. Ability to override everything - paint, pointer, key events are all overridable and replaceable. Developers can also paint over everything for example: glasspane and layered pane.
3. Consistency - provides identical functionality on all platforms.

This all contributes to your ease of working with Codename One and maintaining Codename One. More than 95% of Codename One's code is in Java hence its portable and pretty easy to maintain!

61.11.1. Why does Codename One need native widgets at all

You need the native device to do input, html rendering etc. These are too big and too complex tasks for Codename One to do from scratch.

They're sometimes impossible to perform without the native platform. For example: the virtual keyboard input on the devices is tied directly to the native text input. It's impractical to implement everything from scratch for all languages, dictionaries etc. The result would be sub-par.

A web browser can't be implemented in this age without a JavaScript JIT and including a JIT within an iOS app is prohibited by Apple.

61.11.2. Problems with native widgets

Codename One does pretty much everything on the EDT (Event Dispatch Thread), this provides a lot of cool features for example: modal dialogs, invokeAndBlock etc.

For example native widgets have to be drawn on the devices native UI thread.
This means that drawing looks something like this:

1. Loop over all Codename One components and paint them.
2. Loop over all native peer components and paint them.

This means that all peer components are drawn on top of the Codename One components.



This was also the case in AWT/Swing to one degree or another…

61.11.3. Showing dialogs on top of peer components

Codename One grabs a screenshot of the peer, hide it and then you can show the screenshot. Since the screenshot is static it can be rendered through the standard UI. You can't do that always since grabbing a screenshot is an expensive process on all platforms and must be performed on the native device thread.

61.11.4. Why can't you combine peer component scrolling and Codename One scrolling

Since the form title/footer etc. Are drawn by Codename One the peer component might paint itself on top of them. Clipping a peer component is often pretty difficult. Furthermore, if the user drags his finger within the peer component he might trigger the native scroll within the might collide with your scrolling?

61.11.5. Native components in the first Form

Another problem might be counterintuitive. iOS has screenshot images representing the first form. If your first page is an HTML or a native map (or other peer widget) the screenshot process on the build server will show fallback code instead of the real thing thus providing sub-par behavior.

Its impractical to support something like HTML for the screenshot process since it would also look different from the web component running on the device.



You can read more about the screenshot process here [https://www.codenameone.com/manual/appendix-ios.html#section-ios-screenshots].

61.12. Integrating 3rd party native SDKs

The following is a description of the procedure that was used to create the Codename One FreshDesk library [http://shannah.github.io/cn1-freshdesk/]. This process can be adapted to wrap any native SDK on Android and iOS.

61.12.1. Step 1 : Review the FreshDesk SDKs

Before you begin, you will need to review the Android and iOS SDKs.

1. **FreshDesk Android SDK**: Integration Guide [http://developer.freshdesk.com/mobihelp/android/integration_guide/] | API Docs [http://developer.freshdesk.com/mobihelp/android/api/reference/com/freshdesk/mobihelp/package-summary.html]
2. **FreshDesk iOS SDK**: Integration Guide [http://developer.freshdesk.com/mobihelp/ios/integration_guide/] | API Docs [http://developer.freshdesk.com/mobihelp/ios/api/]

In reviewing the SDKs, you're looking for answers to two questions:

1. What should your Codename One FreshDesk API look like?
2. What will be involved in integrating the native SDK in your app or lib?

61.12.2. Step 2: Designing the Codename One public API

When designing the Codename One API, you should begin by looking at the Javadocs [http://developer.freshdesk.com/mobihelp/android/api/reference/com/freshdesk/mobihelp/package-summary.html] for the native Android SDK. If the class hierarchy doesn't look too elaborate, you may decide to model your Codename One public API on the Android API. But, if you need a small part of the SDK's functionality, you may choose to create your abstractions around the functionality that you need.

For the FreshDesk SDK, it looks like most of the functionality is handled by one central class `Mobihelp`, with a few other POJO classes for passing data to and from the service. This is a good candidate for a comprehensive Codename One API.

Before proceeding, you also need to look at the iOS API to see if there are any features that aren't included. While naming conventions in the iOS API are a little different than those in the Android API, it looks like they're functionally the same.

The chosen approach therefore creates a class hierarchy and API that mirrors the Android SDK.

61.12.3. Step 3: The architecture and internal APIs

A Codename One library that wraps a native SDK, will consist of the following:

1. **Public Java API**, consisting of pure Java classes that are intended to be used by the outside world.
2. **Native Interface(s)**. The Native Interface(s) act as a conduit for the public Java API to communicate to the native SDK. Parameters in native interface methods are limited to primitive types, arrays of primitive types, and Strings, as are return values.
3. **Native code**. Each platform must include an implementation of the Native Interface(s). These implementations are written in the native language of the platform (for example: Java for Android, and Objective-C for iOS).
4. **Native dependencies**. Any 3rd party libraries required for the native code to work, need to be included for each platform. On android, this may mean bundling.jar files,.aar files, or.andlib files. On iOS, this may mean bundling `.h` files, `.a` files, `.framework`, and `.bundle` files.

5. **Build hints**. Some libraries will require you to add some extra build hints to your project. For example: On Android you may need to add permissions to the manifest, or define services in the `<Application>` section of the manifest. On iOS, this may mean specifying more core frameworks for inclusion, or adding build flags for compilation.

The following diagram shows the dependencies in a native library:

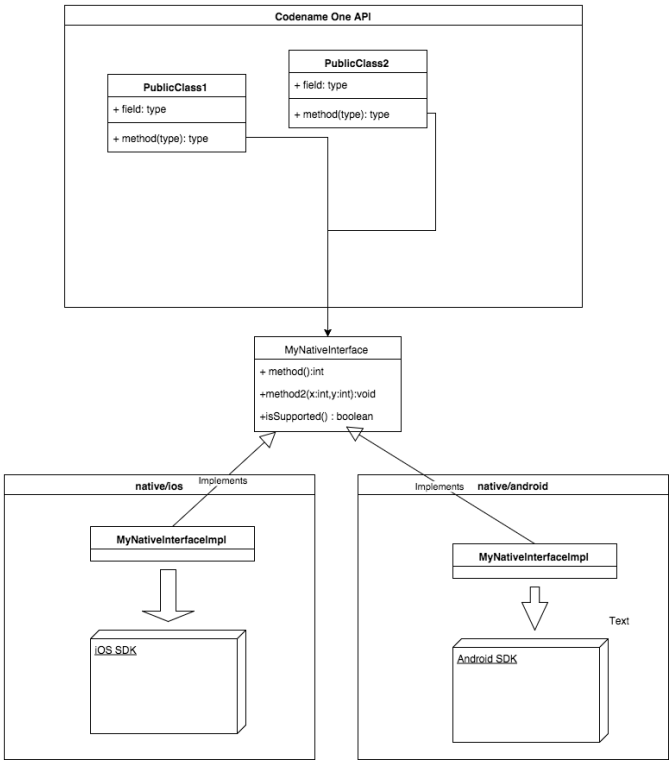


Figure 337. Relationship between native & Codename One API UML Diagram

In the specific case of your FreshDesk API, the public API and classes will look like:

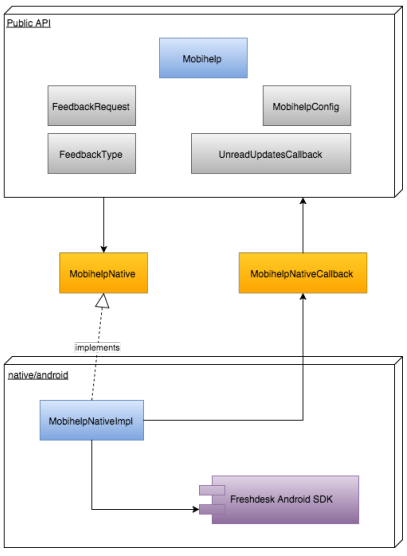


Figure 338. Freshdesk API Integration

Things to notice

1. The public API consists of the main class (`Mobihelp` [https://github.com/shannah/cn1-freshdesk/blob/master/cn1-freshdesk-demo/src/com/codename1/freshdesk/Mobihelp.java]), and a few supporting classes (`FeedbackRequest` [https://github.com/shannah/cn1-freshdesk/blob/master/cn1-freshdesk-demo/src/com/

codename1/freshdesk/FeedbackRequest.java], `FeedbackType` [https://github.com/shannah/cn1-freshdesk/blob/master/cn1-freshdesk-demo/src/com/codename1/freshdesk/FeedbackType.java], `MobihelpConfig` [https://github.com/shannah/cn1-freshdesk/blob/master/cn1-freshdesk-demo/src/com/codename1/freshdesk/MobihelpConfig.java], `MobihelpCallbackStatus` [https://github.com/shannah/cn1-freshdesk/blob/master/cn1-freshdesk-demo/src/com/codename1/freshdesk/MobihelpCallbackStatus.java]), which were copied almost directly from the Android SDK.

2. The way for the public API to communicate with the native SDK is through the `MobihelpNative` [https://github.com/shannah/cn1-freshdesk/blob/master/cn1-freshdesk-demo/src/com/codename1/freshdesk/MobihelpNative.java] interface.
3. You introduced the `MobihelpNativeCallback` [https://github.com/shannah/cn1-freshdesk/blob/master/cn1-freshdesk-demo/src/com/codename1/freshdesk/MobihelpNativeCallback.java] class to ease native code calling back into the public API. This was necessary for a few methods that used asynchronous callbacks.

61.12.4. Step 4: Implement the public API and native interface

You've already looked at the final product of the public API in the previous step. Back up and walk through the process step-by-step.

The goal is to model your API around the Android API. The central class that includes all the functionality of the SDK is the com.freshdesk.mobihelp.Mobihelp class [http://developer.freshdesk.com/mobihelp/android/api/reference/com/freshdesk/mobihelp/Mobihelp.html], so you begin there.

You will start by creating your own package (`com.codename1.freshdesk`) and your own `Mobihelp` class inside it.

Adapting method signatures

The `Context` parameter

In a first glance at the com.freshdesk.mobihelp.Mobihelp API [http://developer.freshdesk.com/mobihelp/android/api/reference/com/freshdesk/mobihelp/Mobihelp.html] you see that many of the methods take a parameter of type `android.content.Context` [http://developer.android.com/reference/android/content/Context.html]. This class is part of the core Android SDK, and won't be accessible to any pure Codename One APIs. Your public API therefore can't include any such references. You will be able to access a suitable context in the native layer, so you will omit this parameter from your public API, and inject them in your native implementation.

Hence, the method signature `public static final void setUserFullName (Context context, String name)` will become `public static final void setUserFullName (String name)` in your public API.

Non-Primitive parameters

Although your public API isn't constrained by the same rules as your Native Interfaces about parameter and return types, you need to be cognizant of the fact that parameters you pass to your public API will be funnelled through your native interface. You should pay attention to any parameters or return types that can't be passed directly to a native interface, and start forming a strategy for them. For example: consider the following method signature from the Android `Mobihelp` class:

```
public static final void showSolutions (Context activityContext, ArrayList<String> tags)
```

You've already decided to omit the `Context` parameter in your API, so that's a non-issue. However, what about the `ArrayList<String>` tags parameter? Passing this to your public API is no problem, but when you implement the public API, how will you pass this `ArrayList` to your native interface, since native interfaces don't allow you to arrays of strings as parameters?

One of three strategies applies in such cases:

1. Encode the parameter as either a single `String` (for example: using JSON or some other parseable format) or a byte[] array (in some known format that can be parsed in native code).
2. Store the parameter on the Codename One side and pass some ID or token that can be used on the native side to retrieve the value.
3. If the data structure can be expressed as a finite number of primitive values, then design the native interface method to take the individual values as parameters instead of a single object. For example: If there is a User [https://www.codenameone.com/javadoc/com/codename1/facebook/User.html] class with properties `name` and `phoneNumber`, the native interface can have `name` and `phoneNumber parameters rather than a single `user` parameter.

In this case, because an array of strings is such a simple data structure, the right approach is a variation on strategy number 1: Merge the array into a single string with a delimiter.

In any case, you don't have to come up with the specifics right now, as you're still on the public API, but it will pay dividends later if you think this through ahead of time.

Callbacks

It's often the case that native code needs to call back into Codename One code when an event occurs. This may be connected directly to an API method call (for example: as the result of an asynchronous method invocation), or due to something initiated by the operating system or the native SDK on its own (for example: a push notification, a location event, etc.).

Native code will have access to both the Codename One API and any native APIs in your app, but on some platforms, accessing the Codename One API may be a little tricky. For example: on iOS you'll be calling from Objective-C back into Java which requires knowledge of Codename One's java-to-goal C conversion process. In general, the easiest way to ease callbacks is to provide abstractions that involve static Java methods (in Codename One space) that accept and return primitive types.

For your `Mobihelp` class, the following method hints at the need to have a "callback plan":

```
public static final void getUnreadCountAsync (Context context, UnreadUpdatesCallback callback)
```

The interface definition for `UnreadUpdatesCallback` is:

```
public interface UnreadUpdatesCallback {
    //This method is called once the unread updates count is available.
```

```
    void onResult(MobihelpCallbackStatus status, Integer count);
}
```

The callback status enum is also mirrored in common code:

```
public enum MobihelpCallbackStatus {
    SUCCESS,
    FAILURE
}
```

In other words: if you were to implement this method (planned for a later section), you need to have a way for the native code to call the `callback.onResult()` method of the passed parameter.

You've two issues that will need to be solved here:

1. How to pass the `callback` object through the native interface.
2. How to **call** the `callback.onResult()` method from native code at the right time.

For the first issue, you will use strategy #2 that you mentioned earlier: (Store the parameter on the Codename One side and pass some ID or token that can be used on the native side to retrieve the value).

For the second issue, you will create a static method that can take the token generated to solve the first issue, and call the stored `callback` object's `onResult()` method. You abstract both sides of this process using the `MobihelpNativeCallback` class [https://github.com/shannah/cn1-freshdesk/blob/master/cn1-freshdesk-demo/src/com/codename1/freshdesk/MobihelpNativeCallback.java]:

```
public class MobihelpNativeCallback {
    private static int nextId = 0;
    private static Map<Integer, UnreadUpdatesCallback> callbacks = new HashMap<>();

    static int registerUnreadUpdatesCallback(UnreadUpdatesCallback callback) {
        callbacks.put(nextId, callback);
        return nextId++;
    }

    public static void fireUnreadUpdatesCallback(int callbackId, final int status, final int count) {
        final UnreadUpdatesCallback cb = callbacks.get(callbackId);
        if (cb != null) {
            callbacks.remove(callbackId);
            Display.getInstance().callSerially(new Runnable() {

                public void run() {
                    MobihelpCallbackStatus status2 = MobihelpCallbackStatus.values()[status];
                    cb.onResult(status2, count);
                }

            });
        }
    }
```

```
}
```

Things to notice here:

1. This class uses a static `Map<Integer,UnreadUpdatesCallback>` member to keep track of all callbacks, mapping a unique integer ID to each callback.
2. The `registerUnreadUpdatesCallback()` method takes an `UnreadUpdatesCallback` object, places it in the `callbacks` map, and returns the integer **token** that can be used to fire the callback later. This method would be called by the public API inside the `getUnreadCountAsync()` method implementation to convert the `callback` into an integer, which can then be passed to the native API.
3. The `fireUnreadUpdatesCallback()` method would be called later from native code. Its first parameter is the token for the callback to call.
4. You wrap the `onResult()` call inside a `Display.callSerially()` invocation to ensure that the callback is called on the EDT. This is a general convention that's used throughout Codename One, and you'd be well-advised to follow it. **Event handlers** should be run on the EDT unless there is a good reason not to - and in that case your documentation and naming conventions should make this clear to avoid stepping into multithreading hell!

Initialization

Most Native SDKs include some sort of initialization method where you pass your developer and application credentials to the API. After filling in FreshDesk's web-based form to create a new application, it generated an application ID, an app "secret," and a "domain." The SDK requires you to pass all three of these values to its `init()` method through the `MobihelpConfig` class.

```
public class MobihelpConfig {
    private String appSecret;
    private String appId;
    private String domain;
    private boolean autoReplyEnabled;
    private boolean enhancedPrivacyModeEnabled;
    private FeedbackType feedbackType;
    private int launchCountForReviewPrompt;
    private boolean prefetchSolutions;

    public void setAppSecret(String appSecret) {
        this.appSecret = appSecret;
    }

    public void setAppId(String appId) {
        this.appId = appId;
    }

    public void setDomain(String domain) {
        this.domain = domain;
    }
```

```
    public String getAppSecret() {
        return appSecret;
    }

    public String getAppId() {
        return appId;
    }

    public String getDomain() {
        return domain;
    }

    public void setAutoReplyEnabled(boolean autoReplyEnabled) {
        this.autoReplyEnabled = autoReplyEnabled;
    }

    public boolean isAutoReplyEnabled() {
        return autoReplyEnabled;
    }

    public void setEnhancedPrivacyModeEnabled(boolean enhancedPrivacyModeEnabled) {
        this.enhancedPrivacyModeEnabled = enhancedPrivacyModeEnabled;
    }

    public boolean isEnhancedPrivacyModeEnabled() {
        return enhancedPrivacyModeEnabled;
    }

    public void setFeedbackType(FeedbackType feedbackType) {
        this.feedbackType = feedbackType;
    }

    public FeedbackType getFeedbackType() {
        return feedbackType;
    }

    public void setLaunchCountForReviewPrompt(int launchCountForReviewPrompt) {
        this.launchCountForReviewPrompt = launchCountForReviewPrompt;
    }

    public int getLaunchCountForReviewPrompt() {
        return launchCountForReviewPrompt;
    }

    public void setPrefetchSolutions(boolean prefetchSolutions) {
        this.prefetchSolutions = prefetchSolutions;
    }

    public boolean isPrefetchSolutions() {
        return prefetchSolutions;
    }
```

```
    public enum FeedbackType {
        POSITIVE,
        NEGATIVE
    }
}
```

Note, but, that FreshDesk (and most other service provides that have native SDKs) requires you to create different Apps for each platform. This means that your App ID and App secret will be different on iOS than they will be on Android.

Your public API therefore needs to enable you to provide many credentials in the same app, and your API needs to know to use the correct credentials depending on the device that the app is running on.

Many solutions to this problem exist, but the chosen approach provides two different `init()` methods:

```
public final static void initAndroid(MobihelpConfig config) {
    if ("and".equals(Display.getInstance().getPlatformName())) {
        init(config);
    }
}
```

and

```
public final static void initIOS(MobihelpConfig config) {
    if ("ios".equals(Display.getInstance().getPlatformName())) {
        init(config);
    }
}
```

Then you can set up the API with code like:

```
public class MobihelpUsage {
    public void configure() {
        MobihelpConfig config = new MobihelpConfig();
        config.setAppSecret("xxxxxxx");
        config.setAppId("freshdeskdemo-2-xxxxxx");
        config.setDomain("codenameonetest1.freshdesk.com");
        MobihelpInitMethods.initIOS(config);

        config = new MobihelpConfig();
        config.setAppSecret("yyyyyyyy");
        config.setAppId("freshdeskdemo-1-yyyyyyyy");
        config.setDomain("https://codenameonetest1.freshdesk.com");
        MobihelpInitMethods.initAndroid(config);
```

```
    }
}
```

The resulting public API

```
public class Mobihelp {

    private static char[] separators = new char[]{',','|','/','@','#','%','!','^','&','*','=','+','*','<'};
    private static MobihelpNative peer;

    public static boolean isSupported() {
        return peer != null;
    }

    public static void setPeer(MobihelpNative peer) {
        Mobihelp.peer = peer;
    }

    //Attach the given custom data (key-value pair) to the conversations/tickets.
    public final static void addCustomData(String key, String value) {
        // ...
    }
    //Attach the given custom data (key-value pair) to the conversations/tickets with the ability to flag sensitive data.
    public final static void addCustomData(String key, String value, boolean isSensitive) {
        // ...
    }
    //Clear all breadcrumb data.
    public final static void clearBreadCrumbs() {
        // ...
    }
    //Clear all custom data.
    public final static void clearCustomData() {
        // ...
    }
    //Clears User information.
    public final static void clearUserData() {
        // ...
    }
    //Retrieve the number of unread items across all the conversations for the user synchronously i.e.
    public final static int getUnreadCount() {
        return 0;
    }

    //Retrieve the number of unread items across all the conversations for the user asynchronously
    public final static void getUnreadCountAsync(UnreadUpdatesCallback callback) {
        // ...
    }
    //Initialize the Mobihelp support section with necessary app configuration.
    public final static void initAndroid(MobihelpConfig config) {
        // ...
    }

    public final static void initIOS(MobihelpConfig config) {
        // ...
    }

    public void showSupport() {
        // ...
    }
}
```

The native interface

The final native interface is identical to your public API, except in cases where the public API included non-primitive parameters:

```
public interface MobihelpNative extends NativeInterface {
    void config_setAppId(String appId);
    void config_setAppSecret(String appSecret);
    void config_setAutoReplyEnabled(boolean enabled);
    void config_setDomain(String domain);
    void config_setEnhancedPrivacyModeEnabled(boolean enabled);
    void config_setFeedbackType(int type);
    void config_setLaunchCountForReviewPrompt(int count);
    void config_setPrefetchSolutions(boolean prefetch);
    void initNative();
    void setUserFullName(String name);
    void showSupportWithTags(String tags, String separator);
    void getUnreadCountAsync(int callbackId);
}
```

Notice also, that the native interface includes a set of methods with names prefixed with `config__`. This naming convention identifies methods that map to the `MobihelpConfig` class. A separate native interface for these would have been possible, but keeping all the native stuff in one class is simpler and easier to maintain.

Connecting the public API to the native interface

You've a public API, and you have a native interface. The idea is that the public API should be a thin wrapper around the native interface to smooth out rough edges that are likely to exist due to the strict set of rules involved in native interfaces. You will, so, use delegation inside the `Mobihelp` class to provide it a reference to an instance of `MobihelpNative`:

```
public class Mobihelp {

    private static char[] separators = new char[]{',','|','/','@','#','%','!','^','&','*','=','+','*','<'};
    private static MobihelpNative peer;

    public static boolean isSupported() {
        return peer != null;
    }

    public static void setPeer(MobihelpNative peer) {
        Mobihelp.peer = peer;
    }

    //Attach the given custom data (key-value pair) to the conversations/tickets.
    public final static void addCustomData(String key, String value) {
        // ...
    }
    //Attach the given custom data (key-value pair) to the conversations/tickets with the ability to flag sensitive data.
    public final static void addCustomData(String key, String value, boolean isSensitive) {
        // ...
    }
    //Clear all breadcrumb data.
```

```
    public final static void clearBreadCrumbs() {
        // ...
    }
    //Clear all custom data.
    public final static void clearCustomData() {
        // ...
    }
    //Clears User information.
    public final static void clearUserData() {
        // ...
    }
    //Retrieve the number of unread items across all the conversations for the user synchronously i.e.
    public final static int getUnreadCount() {
        return 0;
    }

    //Retrieve the number of unread items across all the conversations for the user asynchronously
    public final static void getUnreadCountAsync(UnreadUpdatesCallback callback) {
        // ...
    }
    //Initialize the Mobihelp support section with necessary app configuration.
    public final static void initAndroid(MobihelpConfig config) {
        // ...
    }

    public final static void initIOS(MobihelpConfig config) {
        // ...
    }

    public void showSupport() {
        // ...
    }
}
```

You will initialize this `peer` inside the `init()` method of the `Mobihelp` class. Notice, though that `init()` is `private` since you've provided distinct abstractions for the Android and iOS apps:

```
class MobihelpInitMethods {
    //Initialize the Mobihelp support section with necessary app configuration.
    public final static void initAndroid(MobihelpConfig config) {
        if ("and".equals(Display.getInstance().getPlatformName())) {
            init(config);
        }
    }

    public final static void initIOS(MobihelpConfig config) {
        if ("ios".equals(Display.getInstance().getPlatformName())) {
            init(config);
        }
    }

    private static void init(MobihelpConfig config) {
        MobihelpNative peer = (MobihelpNative) NativeLookup.create(MobihelpNative.class);
        peer.config_setAppId(config.getAppId());
        peer.config_setAppSecret(config.getAppSecret());
        peer.config_setAutoReplyEnabled(config.isAutoReplyEnabled());
        peer.config_setDomain(config.getDomain());
        peer.config_setEnhancedPrivacyModeEnabled(config.isEnhancedPrivacyModeEnabled());
```

```
        if (config.getFeedbackType() != null) {
            peer.config_setFeedbackType(config.getFeedbackType().ordinal());
        }
        peer.config_setLaunchCountForReviewPrompt(config.getLaunchCountForReviewPrompt());
        peer.config_setPrefetchSolutions(config.isPrefetchSolutions());
        peer.initNative();
    }
}
```

Things to Notice:

1. The `initAndroid()` and `initIOS()` methods include a check to see if they're running on the correct platform. They both call `init()`.
2. The `init()` method, uses the NativeLookup [https://www.codenameone.com/javadoc/com/codename1/system/NativeLookup.html] class to instantiate your native interface.

Implementing the glue between public API and native interface

For most of the methods in the `Mobihelp` class, you can see that the public API will be a thin wrapper around the native interface. For example: the public API implementation of `setUserFullName(String)` is:

```
public final static void setUserFullName(String name) {
    peer.setUserFullName(name);
}
```

For some other methods, the public API needs to break apart the parameters into a form that the native interface can accept. For example: the `init()` method, shown above, takes a `MobihelpConfig` object as a parameter, but it passed the properties of the `config` object individually into the native interface.

Another example, is the `showSupport(ArrayList<String> tags)` method. The corresponding native interface method that's wraps is `showSupport(String tags, `String separator)` - i.e., it needs to merge all tags into a single delimited string, and pass then to the native interface along with the delimiter used. The implementation is:

```
public final static void showSupport(ArrayList<String> tags) {
    String separator = findUnusedSeparator(tags);
    StringBuilder sb = new StringBuilder();
    for (String tag : tags) {
        sb.append(tag).append(separator);
    }
    peer.showSupportWithTags(sb.toString().substring(0, sb.length() - separator.length()), separator);
}
```

The other non-trivial wrapper is the `getUnreadCountAsync()` method that you discussed before:

```
public final static void getUnreadCountAsync(UnreadUpdatesCallback callback) {
```

```
    int callbackId = MobihelpNativeCallback.registerUnreadUpdatesCallback(callback);
    peer.getUnreadCountAsync(callbackId);
}
```

61.12.5. Step 5: Implementing the native interface in Android

Now that you've set up your public API and your native interface, it's time to work on the native side of things. You can generate stubs for all platforms in your IDE (Netbeans in your case), by right clicking on the `MobihelpNative` class in the project explorer and selecting `Generate Native Access`.

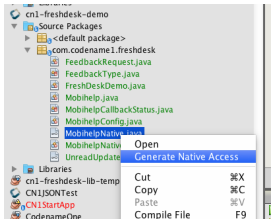


Figure 339. Generate Native Access Menu Item

This will generate a separate directory for each platform inside your project's `native` directory:

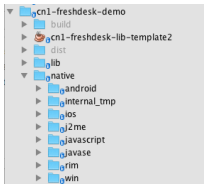


Figure 340. Native generated sources directory view

Inside the `android` directory, this generates a `com/codename1/freshdesk/MobihelpNativeImpl` class with stubs for each method.

Your implementation will be a thin wrapper around the native Android SDK. See the source here [https://github.com/shannah/cn1-freshdesk/blob/master/cn1-freshdesk-demo/native/android/com/codename1/freshdesk/MobihelpNativeImpl.java].

Some highlights:

1. `Context` : The native API requires you to pass a **context** object as a parameter on many methods. This should be the context for the current activity. It will allow the FreshDesk API to know where to return to after it has done its thing. Codename One provides a class called `AndroidNativeUtil` that allows you to retrieve the app's Activity (which includes the Context). You will wrap this with a convenience method in your class as follows:

   ```
   public final static void getUnreadCountAsync(UnreadUpdatesCallback callback) {
       int callbackId = MobihelpNativeCallback.registerUnreadUpdatesCallback(callback);
       peer.getUnreadCountAsync(callbackId);
   }
   ```

   This will enable you to wrap the freshdesk native API. For example:

```
private static Context context() {
    return AndroidNativeUtil.getActivity().getApplicationContext();
}
```

2. `runOnUiThread()` - Many of the calls to the FreshDesk API may have been made from the Codename One EDT. For example, Android has its own event dispatch thread that should be used for interacting with native Android UI. Any API calls that look like they start some sort of native Android UI process should therefore be wrapped inside Android's `runOnUiThread()` method which is like Codename One's `Display.callSerially()` method. For example: see the `showSolutions()` method:

```
public void showSolutions() {
    activity().runOnUiThread(new Runnable() {
        public void run() {
            com.freshdesk.mobihelp.Mobihelp.showSolutions(context());
        }
    });

}
```

   (Note here that the `activity()` method is another convenience method to retrieve the app's current `Activity` from the `AndroidNativeUtil` class).

3. **Callbacks**. You discussed, in detail, the mechanisms you put in place to enable your native code to perform callbacks into Codename One. You can see the native side of this by viewing the `getUnreadCountAsync()` method implementation:

```
public void getUnreadCountAsync(final int callbackId) {
    activity().runOnUiThread(new Runnable() {
        public void run() {
            com.freshdesk.mobihelp.Mobihelp.getUnreadCountAsync(context(), new com.freshdesk.mobihelp
.UnreadUpdatesCallback() {
                public void onResult(com.freshdesk.mobihelp.MobihelpCallbackStatus status, Integer count) {
                    MobihelpNativeCallback.fireUnreadUpdatesCallback(callbackId, status.ordinal(), count);
                }
            });
        }
    });

}
```

61.12.6. Step 6: Bundling the native SDKs

The last step (at least on the Android side) is to bundle the FreshDesk SDK. For Android, there are a few different scenarios you'll run into for embedding SDKs:

1. **The SDK includes Java classes** - NO XML UI files, assets, or resources that aren't included inside a simple.jar file. In this case, you can place the.jar file inside your project's `native/android` directory.

2. **The SDK includes some XML UI files, resources, and assets.** In this case, the SDK is distributed as an Android project folder that can be imported into an Eclipse or Android studio workspace. In general, in this case, you would need to zip the entire directory and change the extension of the resulting.zip file to `.andlib`, and place this in your project's `native/android` directory.
3. **The SDK is distributed as an `.aar` file** - In this case you can copy the `.aar` file into your `native/android` directory.

The FreshDesk SDK

The FreshDesk (aka Mobihelp) SDK is distributed as a project folder (that's: scenario 2 from the above list). Your procedure is therefore to download the SDK (download link [https://s3.amazonaws.com/assets.mobihelp.freshpo.com/sdk/mobihelp_sdk_android.zip]), and rename it from `mobihelp_sdk_android.zip` to `mobihelp_sdk_android.andlib`, and copy it into your `native/android` directory.

Dependencies

In this case there is a catch. The Mobihelp SDK includes a dependency:

> Mobihelp SDK depends on AppCompat-v7 (Revision 19.0+) Library. You will need to update project.properties to point to the Appcompat library.

If you look inside the `project.properties` file (inside the Mobihelp SDK directory--- that's: you'd need to extract it from the zip to view its contents), you'll see the dependency listed:

```
android.library.reference.1=../appcompat_v7
```

In other words, the Mobihelp SDK expects to find the `appcompat_v7` library located in the same parent directory as the Mobihelp SDK project. After a little bit of research (if you're not yet familiar with the Android AppCompat support library), you find that the `AppCompat_v7` library is part of the Android Support library, which can installed into your local Android SDK using Android SDK Manager. Installation processed specified here [https://developer.android.com/tools/support-library/setup.html].

After installing the support library, you need to retrieve it from your Android SDK. You can find that.aar file inside the `ANDROID_HOME/sdk/extras/android/m2repository/com/android/support/appcompat-v7/19.1.0/` directory (for version 19.1.0). The contents of that directory on your system are:

```
appcompat-v7-19.1.0.aar     appcompat-v7-19.1.0.pom
appcompat-v7-19.1.0.aar.md5 appcompat-v7-19.1.0.pom.md5
appcompat-v7-19.1.0.aar.sha1    appcompat-v7-19.1.0.pom.sha1
```

There are two files of interest here:

1. appcompat-v7-19.1.0.aar - This is the actual library that you need to include in your project to meet the MobiSDK dependency.
2. appcompat-v7-19.1.0.pom - This is the Maven XML file for the library. It will show you any dependencies that the appcompat library has. You will also need to include these dependencies:

```
  <dependencies>
    <dependency>
      <groupId>com.android.support</groupId>
      <artifactId>support-v4</artifactId>
      <version>19.1.0</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
```

   In other words: you need to include the `support-v4` library version 19.1.0 in your project. This is also part of the Android Support library. If you back up a couple of directories to: `ANDROID_HOME/sdk/extras/android/m2repository/com/android/support`, you will see it listed there:

```
appcompat-v7            palette-v7
cardview-v7         recyclerview-v7
gridlayout-v7           support-annotations
leanback-v17            support-v13
mediarouter-v7          support-v4
multidex            test
multidex-instrumentation
```

   + And if you look inside the appropriate version directory of `support-v4` (in `ANDROID_HOME/sdk/extras/android/m2repository/com/android/support/support-v4/19.1.0`), you see:

```
support-v4-19.1.0-javadoc.jar       support-v4-19.1.0.jar
support-v4-19.1.0-javadoc.jar.md5   support-v4-19.1.0.jar.md5
support-v4-19.1.0-javadoc.jar.sha1  support-v4-19.1.0.jar.sha1
support-v4-19.1.0-sources.jar       support-v4-19.1.0.pom
support-v4-19.1.0-sources.jar.md5   support-v4-19.1.0.pom.md5
support-v4-19.1.0-sources.jar.sha1  support-v4-19.1.0.pom.sha1
```

   Looks like this library is pure Java classes, so you need to include the `support-v4-19.1.0.jar` file into your project. Checking the `.pom` file you see that there are no more dependencies you need to add.

To summarize your findings, you need to include the following files in your `native/android` directory:

1. appcompat-v7-19.1.0.aar
2. support-v4-19.1.0.jar

And since your Mobihelp SDK lists the appcompat_v7 dependency path as "../appcompat_v7" in its project.properties file, you're going to rename `appcompat-v7-19.1.0.aar` to `appcompat_v7.aar`.

When all is said and done, your `native/android` directory should contain the following:

```
appcompat_v7.aar    mobihelp.andlib
com         support-v4-19.1.0.jar
```

61.12.7. Step 7: Injecting Android manifest and proguard config

The final step on the Android side is to inject necessary permissions and services into the project's AndroidManifest.xml file.

You can find the manifest file injections required by opening the `AndroidManifest.xml` file from the MobiHelp SDK project. Its contents are as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.freshdesk.mobihelp"
    android:versionCode="1"
    android:versionName="1.0" >

     <uses-sdk
        android:minSdkVersion="10" />

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.READ_LOGS" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

    <application>
        <activity
            android:name="com.freshdesk.mobihelp.activity.SolutionArticleListActivity"
            android:configChanges="orientation|screenSize"
            android:theme="@style/Theme.Mobihelp"
            android:windowSoftInputMode="adjustPan" >
        </activity>
        <activity
            android:name="com.freshdesk.mobihelp.activity.FeedbackActivity"
            android:configChanges="keyboardHidden|orientation|screenSize"
            android:theme="@style/Theme.Mobihelp"
            android:windowSoftInputMode="adjustResize|stateVisible" >
        </activity>
        <activity
            android:name="com.freshdesk.mobihelp.activity.InterstitialActivity"
            android:configChanges="orientation|screenSize"
            android:theme="@style/Theme.AppCompat">
        </activity>
        <activity
            android:name="com.freshdesk.mobihelp.activity.TicketListActivity"
            android:parentActivityName="com.freshdesk.mobihelp.activity.SolutionArticleListActivity"
            android:theme="@style/Theme.Mobihelp" >
```

```
            <!-- Parent activity meta-data to support 4.0 and lower -->
            <meta-data
                android:name="android.support.PARENT_ACTIVITY"
                android:value="com.freshdesk.mobihelp.activity.SolutionArticleListActivity" />
        </activity>
        <activity
            android:name="com.freshdesk.mobihelp.activity.SolutionArticleActivity"
            android:parentActivityName="com.freshdesk.mobihelp.activity.SolutionArticleListActivity"
            android:theme="@style/Theme.Mobihelp"
            android:configChanges="orientation|screenSize|keyboard|keyboardHidden" >

            <!-- Parent activity meta-data to support 4.0 and lower -->
            <meta-data
                android:name="android.support.PARENT_ACTIVITY"
                android:value="com.freshdesk.mobihelp.activity.SolutionArticleListActivity" />
        </activity>
        <activity
            android:name="com.freshdesk.mobihelp.activity.ConversationActivity"
            android:parentActivityName="com.freshdesk.mobihelp.activity.SolutionArticleListActivity"
            android:theme="@style/Theme.Mobihelp"
            android:windowSoftInputMode="adjustResize|stateHidden" >

            <!-- Parent activity meta-data to support 4.0 and lower -->
            <meta-data
                android:name="android.support.PARENT_ACTIVITY"
                android:value="com.freshdesk.mobihelp.activity.SolutionArticleListActivity" />
        </activity>
        <activity
            android:name="com.freshdesk.mobihelp.activity.AttachmentHandlerActivity"
            android:configChanges="keyboardHidden|orientation|screenSize"
            android:parentActivityName="com.freshdesk.mobihelp.activity.SolutionArticleListActivity"
            android:theme="@style/Theme.Mobihelp" >

            <!-- Parent activity meta-data to support 4.0 and lower -->
            <meta-data
                android:name="android.support.PARENT_ACTIVITY"
                android:value="com.freshdesk.mobihelp.activity.SolutionArticleListActivity" />
        </activity>

        <service android:name="com.freshdesk.mobihelp.service.MobihelpService" />

        <receiver android:name="com.freshdesk.mobihelp.receiver.ConnectivityReceiver" >
            <intent-filter>
                <action android:name="android.net.conn.CONNECTIVITY_CHANGE" />
            </intent-filter>
        </receiver>
    </application>

</manifest>
```

You will need to add the `<uses-permission>` tags and all the contents of the `<application>` tag to your manifest file. Codename One provides the following build hints for these:

1. `android.xpermissions` - For your `<uses-permission>` directives. Add a build hint with name `android.xpermissions`, and for the value, paste the actual `<uses-permission>` XML tag.

2. `android.xapplication` - For the contents of your `<application>` tag.

Proguard config

For the release build, you will also need to inject some proguard configuration so that important classes don't get stripped out at build time. The FreshDesk SDK instructions state:

> If you use Proguard, please make sure you have the following included in your project's proguard-project.txt
>
> ```
> -keep class android.support.v4.** { *; }
> -keep class android.support.v7.** { *; }
> ```

Also, if you look at the `proguard-project.txt` file inside the Mobihelp SDK, you'll see the rules:

```
-keep public class * extends android.app.Service
-keep public class * extends android.content.BroadcastReceiver
-keep public class * extends android.app.Activity
-keep public class * extends android.preference.Preference
-keep public class com.freshdesk.mobihelp.exception.MobihelpComponentNotFoundException

-keepclassmembers class * implements android.os.Parcelable {
  public static final android.os.Parcelable$Creator *;
}
```

You will want to merge this and then paste them into the build hint `android.proguardKeep` of your project.

Troubleshooting Android stuff

If, after doing all this, your project fails to build, you can enable the "Include Source" option of the build server, then download the source project, open it in Eclipse or Android Studio, and debug from there.

61.13. Part 2: Implementing the iOS native code

Part 1 of this tutorial focused on the Android native integration. Now you will shift your focus to the iOS implementation.

After selecting "Generate Native Interfaces" for your "MobihelpNative" class, you'll find a `native/ios` directory in your project with the following files:

1. `com_codename1_freshdesk_MobihelpNativeImpl.h` [https://github.com/shannah/cn1-freshdesk/blob/master/cn1-freshdesk-demo/native/ios/com_codename1_freshdesk_MobihelpNativeImpl.h]
2. `com_codename1_freshdesk_MobihelpNativeImpl.m` [https://github.com/shannah/cn1-freshdesk/blob/master/cn1-freshdesk-demo/native/ios/com_codename1_freshdesk_MobihelpNativeImpl.m]

These files contain stub implementations corresponding to your `MobihelpNative` class.

You make use of the API docs [http://developer.freshdesk.com/mobihelp/ios/api/] to see how the native SDK needs to be wrapped. The method names aren't the same. For example: instead of a method `showFeedback()`, it has a message `-presentFeedback:`

You more-or-less follow the iOS integration guide [http://developer.freshdesk.com/mobihelp/ios/integration_guide/#getting-started] for wrapping the API. Some key points include:

1. Remember to import the `Mobihelp.h` file in your header file:

```
#import "Mobihelp.h"
```

2. Like your use of `runOnUiThread()` in Android, you will wrap all your API calls in either `dispatch_async()` or `dispatch_sync()` calls to ensure that you're interacting with the Mobihelp API on the app's main thread rather than the Codename One EDT.
3. Some methods/messages in the Mobihelp SDK require you to pass a `UIViewController` as a parameter. In Codename One, the entire application uses a single UIViewController: `CodenameOne_GLViewController`. You can get a reference to this using the `[CodenameOne_GLViewController instance]` message. You need to import its header file:

```
#import "CodenameOne_GLViewController.h"
```

   As an example, look at the `showFeedback()` method:

```
-(void)showFeedback{
    dispatch_async(dispatch_get_main_queue(), ^{
        [[Mobihelp sharedInstance] presentFeedback:[CodenameOne_GLViewController instance]];
    });
}
```

61.13.1. Using the MobihelpNativeCallback

You described earlier how you created a static method on the `MobihelpNativeCallback` class so that native code could fire a callback method. Now take a look at how this looks from the iOS side of the fence. Here is the implementation of `getUnreadCountAsync()`:

```
-(void)getUnreadCountAsync:(int)param{
    dispatch_async(dispatch_get_main_queue(), ^{
        [[Mobihelp sharedInstance]
            unreadCountWithCompletion:^(NSInteger count){
                com_codename1_freshdesk_MobihelpNativeCallback_fireUnreadUpdatesCallback___int_int_int(
                    CN1_THREAD_GET_STATE_PASS_ARG param, 3 /*SUCCESS*/, count);
            }];
    });
}
```

In your case the iOS SDK version of this method is `+unreadCountWithCompletion:` which takes a block (which is like an anonymous function) as a parameter.

The callback to your Codename One function occurs on this line:

```
com_codename1_freshdesk_MobihelpNativeCallback_fireUnreadUpdatesCallback___int_int_int(
    CN1_THREAD_GET_STATE_PASS_ARG param, 3 /*SUCCESS*/, count);
```

Some things worth mentioning here:

1. The method name is the result of taking the FQN (`MobihelpNativeCallback.fireUpdateUnreadUpdatesCallback(int, int, int)` in the package `com.codename1.freshdesk`) and replacing all `.` characters with underscores, suffixing two underscores after the end of the method name, then appending `_int` once for each of the `int` arguments.
2. You also need to import the header file for this class:

   ```
   #import "com_codename1_freshdesk_MobihelpNativeCallback.h"
   ```

61.13.2. Bundling native iOS SDK

Now that you've implemented your iOS native interface, you need to bundle the Mobihelp iOS SDK into your project. A few different scenarios you may face when looking to include a native SDK:

1. The SDK includes `.bundle` resource files. In this case, copy the `.bundle` file(s) into your `native/ios` directory.
2. The SDK includes `.h` header files. In this case, copy the `.h` file(s) into your `native/ios` directory.
3. The SDK includes `.a` files. In this case, copy the `.a` file(s) into your `native/ios` directory.
4. The SDK includes `.framework` files. In this case, you'll need to zip up the framework, and copy it into your `native/ios` directory. For example: If the framework is named, `MyFramework.framework`, then the zip file should be named MyFramework.framework.zip, and should be located at `native/ios/MyFramework.framework.zip`.

The FreshDesk SDK doesn't include any `.framework` files, so you don't need to worry about that last scenario. You download the iOS SDK [https://s3.amazonaws.com/assets.mobihelp.freshpo.com/sdk/mobihelp_sdk_ios.zip], copy the `libFDMobihelpSDK.a`, `Mobihelp.h`. `MHModel.bundle`, `MHResources.bundle`, and `MHLocalization/en.proj/MHLocalizable.strings` into `native/ios`.

61.13.3. Troubleshooting iOS

If you run into problems with the build, you can select "Include Sources" in the build server to download the resulting Xcode Project. You can then debug the Xcode project locally, make changes to your iOS native implementation files, and copy them back into your project once it's building.

61.13.4. Adding required core libraries and frameworks

The iOS integration guide for the FreshDesk SDK lists the following core frameworks as dependencies:

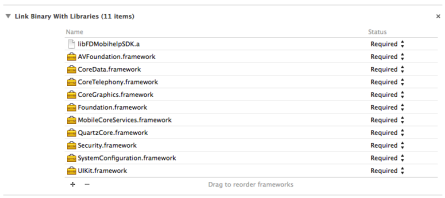


Figure 341. IOS link options

You can add these dependencies to your project using the `ios.add_libs` build hint. For example:

Figure 342. iOS's "add libs" build hint

In short: you list the framework names separated by semicolons. Notice that your list in the above image doesn't include all the frameworks that they list because many of the frameworks are already included by default (I obtained the default list by building the project with "include sources" checked, then looked at the frameworks that were included).

61.14. Part 3 : Packaging as a cn1lib

During the initial development, you find it easier to use a regular Codename One application project so that you can run and test as you go. However, once it's stabilized and you want to distribute the library to other developers, transfer it over to a Codename One library project. This general process involves:

1. Generate a Maven cn1lib project using the `cn1lib-archetype` (see Section 1.5, "Codename One libraries").
2. Copy the Java files from your original application project into the new library project's `common/src/main/java` directory.
3. Copy any native source directories from your original project into the matching platform module (`ios/src/main/objectivec`, `android/src/main/java`, etc.).
4. Copy the **relevant** build hints from the original project's `codenameone_settings.properties` file into the library project's `common/codenameone_library_appended.properties` file.

For the FreshDesk.cn1lib, the original project's build script was modified to generate and build a library project automatically. However, that's beyond the scope of this tutorial.

61.15. Building your own Layout manager

A Layout [https://www.codenameone.com/javadoc/com/codename1/ui/layouts/Layout.html] contains all the logic for positioning Codename One components. It essentially traverses a Codename One Container [https://www.codenameone.com/javadoc/com/codename1/ui/Container.html] and positions components based on internal logic.

When you build the layout you need to take margin into consideration and make sure to add it into the position/size calculations. Building a layout manager involves two simple methods: `layoutContainer` & `getPreferredSize`.

`layoutContainer` is invoked whenever Codename One decides the container needs rearranging, Codename One tries to avoid calling this method and invokes it at the last possible moment. Since this method is expensive (imagine the recursion with nested layouts). Codename One marks a flag indicating layout is "dirty" when something important changes and tries to avoid "reflows."

`getPreferredSize` allows the layout to determine the size desired for the container. This might be a difficult call to make for some layout managers that try to provide both flexibility and simplicity.

Most of `FlowLayout` bugs stem from the fact that this method is impossible to implement efficiently for all the use cases of a nested `FlowLayout`. The size of the final layout won't necessarily match the requested size (it probably won't) but the requested size is taken into consideration, when scrolling and also when sizing parent containers.

This is a layout manager that arranges components in a center column aligned to the middle. You then show the proper usage of margin to create a stair like effect with this layout manager:

```
class CenterLayout extends Layout {
    public void layoutContainer(Container parent) {
        int components = parent.getComponentCount();
        Style parentStyle = parent.getStyle();
        int centerPos = parent.getLayoutWidth() / 2 + parentStyle.getMargin(Component.LEFT);
        int y = parentStyle.getMargin(Component.TOP);
        boolean rtl = parent.isRTL();
        for (int iter = 0; iter < components; iter++) {
            Component current = parent.getComponentAt(iter);
            Dimension d = current.getPreferredSize();
            Style currentStyle = current.getStyle();
            int marginRight = currentStyle.getMarginRight(rtl);
            int marginLeft = currentStyle.getMarginLeft(rtl);
            int marginTop = currentStyle.getMarginTop();
            int marginBottom = currentStyle.getMarginBottom();
            current.setSize(d);
            int actualWidth = d.getWidth() + marginLeft + marginRight;
            current.setX(centerPos - actualWidth / 2 + marginLeft);
            y += marginTop;
            current.setY(y);
            y += d.getHeight() + marginBottom;
        }
    }

    public Dimension getPreferredSize(Container parent) {
        int components = parent.getComponentCount();
        Style parentStyle = parent.getStyle();
        int height = parentStyle.getMargin(Component.TOP) + parentStyle.getMargin(Component.BOTTOM);
        int marginX = parentStyle.getMargin(Component.RIGHT) + parentStyle.getMargin(Component.LEFT);
        int width = marginX;
        for (int iter = 0; iter < components; iter++) {
            Component current = parent.getComponentAt(iter);
            Dimension d = current.getPreferredSize();
            Style currentStyle = current.getStyle();
```

```
            width = Math.max(d.getWidth() + marginX + currentStyle.getMargin(Component.RIGHT)
                    + currentStyle.getMargin(Component.LEFT), width);
            height += currentStyle.getMargin(Component.TOP) + d.getHeight()
                    + currentStyle.getMargin(Component.BOTTOM);
        }
        Dimension size = new Dimension(width, height);
        return size;
    }
}

public class CenterLayoutDemo {
    public static Form createForm() {
        Form hi = new Form("Center Layout", new CenterLayout());
        for (int iter = 1; iter < 10; iter++) {
            Label l = new Label("Label: " + iter);
            l.getUnselectedStyle().setMarginUnit(Style.UNIT_TYPE_DIPS);
            l.getUnselectedStyle().setMarginLeft(iter * 4);
            l.getUnselectedStyle().setMarginRight(0);
            hi.add(l);
        }
        hi.add(new Label("Really Wide Label Text!!!"));
        return hi;
    }

    public void show() {
        createForm().show();
    }
}
```

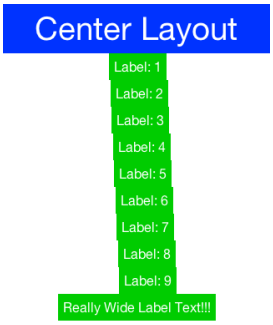


Figure 343. Center layout staircase effect with margin

61.15.1. Porting a Swing/AWT Layout manager

The GridBagLayout [https://www.codenameone.com/javadoc/com/codename1/ui/layouts/GridBagLayout.html] was ported to Codename One considering the complexity of that specific layout manager. Here are some tips you should consider when porting a Swing/AWT layout manager:

1. Codename One doesn't have `Insets`, but adds some support for them to port GridBag but components in Codename One have a margin they need to consider instead of the `Insets` (the padding is in the preferred size and is thus hidden from the layout manager).
2. AWT layout managers also synchronize a lot on the AWT thread. This is no longer necessary since Codename One is single threaded, like Swing.

3. AWT considers the top left position of the `Container` to be 0,0 whereas Codename One considers the position based on its parent `Container`. The top left position in Codename One is `getX()`, `getY()`.

Other than those things it's fixing method and import statements, which are slightly different. Pretty trivial stuff.

61.16. Port a language to Codename One

As you may have already read, you've added support for Kotlin in Codename One. This is something that you can achieve without the help of Codename One. You could port a 3rd party language like Scala, Ruby, Python etc. to Codename One.

61.16.1. What's a JVM language

A JVM Language is any programming language that can be compiled to byte-codes that will run on the JVM (Java Virtual Machine). Java was the original JVM language, but many others have sprung up over the years. Kotlin [https://kotlinlang.org/], Scala [https://www.scala-lang.org/], Groovy [http://groovy-lang.org/], and JRuby [http://jruby.org/] come to mind as well-established and mature languages, but there are many others [https://en.wikipedia.org/wiki/List_of_JVM_languages].

61.16.2. How hard is it to port a JVM language to Codename One

The difficulty of porting a particular language to Codename One will vary depending on such factors as:

1. Does it require a runtime library?
    a. How complex is the runtime library? (For example: Does it require classes that aren't offered in Codename One's subset of the java standard libraries?)
2. Does it need reflection?
    a. Codename One doesn't support reflection because it would result in a large application size. If a JVM language requires reflection to get off the ground then adding it to Codename one would be tricky.
3. Does it perform any runtime byte-code manipulation?
    a. Some dynamic languages may perform byte-code manipulation at runtime. This is problematic on iOS (and possibly other platforms) which prohibits such runtime behavior.

Step 1: Assess the language

The more similar a language, and its build outputs are to Java, the easier it will be to port (probably). Most JVM languages have two parts:

1. A compiler, which compiles source files to JVM byte-code (usually as `.class` files).
2. A runtime library.

One language (other than Java) doesn't require a runtime library: Mirah [http://www.mirah.org/].



Codename One also supports Mirah [https://www.codenameone.com/blog/mirah-for-codename-one.html]

Assessing the Byte-Code

The first thing to do is take a look at the byte-code that's produced by the compiler. Use `javap` to print out a nice version.

Consider a sample Kotlin class that extends `Form`, creates a `Label` and `Button`, and updates the label from the button's listener.

Take a look at the bytecode that Kotlin produced for this class:

```
$ javap -v com/codename1/hellokotlin2/KotlinForm.class
```

```
  Last modified 10-Jul-2017; size 1456 bytes
  MD5 checksum 1cb00f6e63b918bb5a9f146ca8b0b78e
  Compiled from "KotlinForm.kt"
public final class com.codename1.hellokotlin2.KotlinForm extends com.codename1.ui.Form
  SourceFile: "KotlinForm.kt"
  InnerClasses:
       static final #31; //class com/codename1/hellokotlin2/KotlinForm$1
  RuntimeVisibleAnnotations:
    0: #56(#57=[I#58,I#58,I#59],#60=[I#58,I#61,I#58],#62=I#58,#63=[s#64],#65=[s#55,s#66,s#6,s#67])
  minor version: 0
  major version: 50
  flags: ACC_PUBLIC, ACC_FINAL, ACC_SUPER
Constant pool:
   #1 = Utf8               com/codename1/hellokotlin2/KotlinForm
   #2 = Class              #1             //  com/codename1/hellokotlin2/KotlinForm
   #3 = Utf8               com/codename1/ui/Form
   #4 = Class              #3             //  com/codename1/ui/Form
   #5 = Utf8               <init>
   #6 = Utf8               ()V
   #7 = Utf8               Hello Kotlin
   #8 = String             #7             //  Hello Kotlin
   #9 = Utf8               com/codename1/ui/layouts/BoxLayout
  #10 = Class              #9             //  com/codename1/ui/layouts/BoxLayout
  #11 = Utf8               y
  #12 = Utf8               ()Lcom/codename1/ui/layouts/BoxLayout;
  #13 = NameAndType        #11:#12        //  y:()Lcom/codename1/ui/layouts/BoxLayout;
  #14 = Methodref          #10.#13        //  com/codename1/ui/layouts/BoxLayout.y:()Lcom/codename1/ui/layouts/BoxLayout;
  #15 = Utf8               com/codename1/ui/layouts/Layout
  #16 = Class              #15            //  com/codename1/ui/layouts/Layout
  #17 = Utf8               (Ljava/lang/String;Lcom/codename1/ui/layouts/Layout;)V
  #18 = NameAndType        #5:#17         //  "<init>":(Ljava/lang/String;Lcom/codename1/ui/layouts/Layout;)V
  #19 = Methodref          #4.#18         //
com/codename1/ui/Form."<init>":(Ljava/lang/String;Lcom/codename1/ui/layouts/Layout;)V
  #20 = Utf8               com/codename1/ui/Label
  #21 = Class              #20            //  com/codename1/ui/Label
  #22 = Utf8               (Ljava/lang/String;)V
  #23 = NameAndType        #5:#22         //  "<init>":(Ljava/lang/String;)V
  #24 = Methodref          #21.#23        //  com/codename1/ui/Label."<init>":(Ljava/lang/String;)V
  #25 = Utf8               com/codename1/ui/Button
  #26 = Class              #25            //  com/codename1/ui/Button
  #27 = Utf8               Click Me
  #28 = String             #27            //  Click Me
  #29 = Methodref          #26.#23        //  com/codename1/ui/Button."<init>":(Ljava/lang/String;)V
  #30 = Utf8               com/codename1/hellokotlin2/KotlinForm$1
```

```
  #31 = Class              #30            //  com/codename1/hellokotlin2/KotlinForm$1
  #32 = Utf8               (Lcom/codename1/hellokotlin2/KotlinForm;Lcom/codename1/ui/Label;)V
  #33 = NameAndType        #5:#32         //  "<init>":(Lcom/codename1/hellokotlin2/KotlinForm;Lcom/codename1/ui/Label;)V
  #34 = Methodref          #31.#33        //
com/codename1/hellokotlin2/KotlinForm$1."<init>":(Lcom/codename1/hellokotlin2/KotlinForm;Lcom/codename1/ui/Label;)V
  #35 = Utf8               com/codename1/ui/events/ActionListener
  #36 = Class              #35            //  com/codename1/ui/events/ActionListener
  #37 = Utf8               addActionListener
  #38 = Utf8               (Lcom/codename1/ui/events/ActionListener;)V
  #39 = NameAndType        #37:#38        //  addActionListener:(Lcom/codename1/ui/events/ActionListener;)V
  #40 = Methodref          #26.#39        //
com/codename1/ui/Button.addActionListener:(Lcom/codename1/ui/events/ActionListener;)V
  #41 = Utf8               com/codename1/ui/Component
  #42 = Class              #41            //  com/codename1/ui/Component
  #43 = Utf8               add
  #44 = Utf8               (Lcom/codename1/ui/Component;)Lcom/codename1/ui/Container;
  #45 = NameAndType        #43:#44        //  add:(Lcom/codename1/ui/Component;)Lcom/codename1/ui/Container;
  #46 = Methodref          #2.#45         //
com/codename1/hellokotlin2/KotlinForm.add:(Lcom/codename1/ui/Component;)Lcom/codename1/ui/Container;
  #47 = Utf8               com/codename1/ui/Container
  #48 = Class              #47            //  com/codename1/ui/Container
  #49 = Methodref          #48.#45        //
com/codename1/ui/Container.add:(Lcom/codename1/ui/Component;)Lcom/codename1/ui/Container;
  #50 = Utf8               clickMe
  #51 = Utf8               Lcom/codename1/ui/Button;
  #52 = Utf8               label
  #53 = Utf8               Lcom/codename1/ui/Label;
  #54 = Utf8               this
  #55 = Utf8               Lcom/codename1/hellokotlin2/KotlinForm;
  #56 = Utf8               Lkotlin/Metadata;
  #57 = Utf8               mv
  #58 = Integer            1
  #59 = Integer            6
  #60 = Utf8               bv
  #61 = Integer            0
  #62 = Utf8               k
  #63 = Utf8               d1
  #64 = Utf8
                           \n\n\20¢¨
  #65 = Utf8               d2
  #66 = Utf8               Lcom/codename1/ui/Form;
  #67 = Utf8               HelloKotlin2
  #68 = Utf8               KotlinForm.kt
  #69 = Utf8               Code
  #70 = Utf8               LocalVariableTable
  #71 = Utf8               LineNumberTable
  #72 = Utf8               SourceFile
  #73 = Utf8               InnerClasses
  #74 = Utf8               RuntimeVisibleAnnotations
{
  public com.codename1.hellokotlin2.KotlinForm();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=5, locals=3, args_size=1
         0: aload_0
         1: ldc           #8                  // String Hello Kotlin
         3: invokestatic  #14                 // Method
com/codename1/ui/layouts/BoxLayout.y:()Lcom/codename1/ui/layouts/BoxLayout;
         6: checkcast     #16                 // class com/codename1/ui/layouts/Layout
         9: invokespecial #19                 // Method
com/codename1/ui/Form."<init>":(Ljava/lang/String;Lcom/codename1/ui/layouts/Layout;)V
        12: new           #21                 // class com/codename1/ui/Label
        15: dup
        16: ldc           #8                  // String Hello Kotlin
        18: invokespecial #24                 // Method com/codename1/ui/Label."<init>":(Ljava/lang/String;)V
```

```
        21: astore_1
        22: new           #26                 // class com/codename1/ui/Button
        25: dup
        26: ldc           #28                 // String Click Me
        28: invokespecial #29                 // Method com/codename1/ui/Button."<init>":(Ljava/lang/String;)V
        31: astore_2
        32: aload_2
        33: new           #31                 // class com/codename1/hellokotlin2/KotlinForm$1
        36: dup
        37: aload_0
        38: aload_1
        39: invokespecial #34                 // Method
com/codename1/hellokotlin2/KotlinForm$1."<init>":(Lcom/codename1/hellokotlin2/KotlinForm;Lcom/codename1/ui/Label;)V
        42: checkcast     #36                 // class com/codename1/ui/events/ActionListener
        45: invokevirtual #40                 // Method
com/codename1/ui/Button.addActionListener:(Lcom/codename1/ui/events/ActionListener;)V
        48: aload_0
        49: aload_1
        50: checkcast     #42                 // class com/codename1/ui/Component
        53: invokevirtual #46                 // Method add:(Lcom/codename1/ui/Component;)Lcom/codename1/ui/Container;
        56: aload_2
        57: checkcast     #42                 // class com/codename1/ui/Component
        60: invokevirtual #49                 // Method
com/codename1/ui/Container.add:(Lcom/codename1/ui/Component;)Lcom/codename1/ui/Container;
        63: pop
        64: return
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
           32      32     2 clickMe   Lcom/codename1/ui/Button;
           22      42     1 label   Lcom/codename1/ui/Label;
            0      65     0  this   Lcom/codename1/hellokotlin2/KotlinForm;
      LineNumberTable:
        line 13: 0
        line 14: 12
        line 15: 22
        line 16: 32
        line 21: 48
}
```

That's a big mess of stuff, but it's pretty easy to pick through it when you know what you're looking for. The layout of this output is pretty straight forward. The beginning shows that this is a class definition:

```
public final class KotlinForm extends Form {
    //...
}
```

Even comparing this line with the class definition from the source file you've learned something about the Kotlin compiler. It has made the class `final` by default. That observation shouldn't affect your assessment here, but it's kind of interesting.

After the class definition, it shows the internal classes:

```
InnerClasses:
     static final #31; //class com/codename1/hellokotlin2/KotlinForm$1
```

The Constant Pool

And the constants that are used in the class:

```
Constant pool:
   #1 = Utf8               com/codename1/hellokotlin2/KotlinForm
   #2 = Class              #1             //  com/codename1/hellokotlin2/KotlinForm
   #3 = Utf8               com/codename1/ui/Form
   #4 = Class              #3             //  com/codename1/ui/Form
   #5 = Utf8               <init>
   #6 = Utf8               ()V
   #7 = Utf8               Hello Kotlin
   #8 = String             #7             //  Hello Kotlin
   #9 = Utf8               com/codename1/ui/layouts/BoxLayout
   ... etc.
```

The constant pool will consist of class names, and strings. You'll want to peruse this list to see if the compiler has added any classes that aren't in the source code. In the example above, it looks like Kotlin is pretty faithful to the original source's dependencies. It didn't inject any classes that aren't in the original source.

Even if the compiler does inject other dependencies into the bytecode, it might not be a problem. It's a problem if those classes aren't supported by Codename One. Watch for anything in the `java.lang.reflect` package or unsolicited use of `java.net`, `java.nio`, or any other package that aren't part of the Codename One standard library. If you're not sure if a class or package is available in the Codename One standard library, check the javadocs [https://www.codenameone.com/javadoc/].

The ByteCode Instructions:

After the constant pool, you see each of the methods of the class written out as a list of bytecode instructions. For example:

```
public com.codename1.hellokotlin2.KotlinForm();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=5, locals=3, args_size=1
       0: aload_0
       1: ldc           #8                  // String Hello Kotlin
       3: invokestatic  #14                 // Method
com/codename1/ui/layouts/BoxLayout.y:()Lcom/codename1/ui/layouts/BoxLayout;
       6: checkcast     #16                 // class com/codename1/ui/layouts/Layout
       9: invokespecial #19                 // Method
com/codename1/ui/Form."<init>":(Ljava/lang/String;Lcom/codename1/ui/layouts/Layout;)V
      12: new           #21                 // class com/codename1/ui/Label
      15: dup
      16: ldc           #8                  // String Hello Kotlin
      etc.
```

In the above snippet, the first instruction is `aload_0` (which adds `this` to the stack). The 2nd instruction is `ldc`, (which loads constant #8 — the string "Hello Kotlin" to the stack). The 3rd instruction is `invokestatic` which calls the static method define by Constant #14 from the constant pool, with the two parameters that had been added to the stack.



You don’t need to understand what all these instructions do. You need to look for instructions that may be problematic.

The instruction that **might** be problematic is "invokedynamic." All other instructions should work fine in Codename One. (It isn’t certain that invokedynamic won’t work - it might not work on some platforms.)

Summary of Byte-code Assessment

To summarize, the byte-code assessment phase, you’re basically looking to make sure that the compiler doesn’t tend to add dependencies to parts of the JDK that Codename One doesn’t support. And you want to make sure that it doesn’t use invokedynamic.

If you find that the compiler does use invokedynamic or add references to classes that Codename One doesn’t support, don’t give up yet. You might be able to create your own "porting" runtime library that will provide these dependencies at runtime.

Assessing the runtime library

The process for assessing the runtime library is pretty like the process for the bytecodes. You’ll want to get your hands on the language’s runtime library, and use `javap` to inspect the `.class` files. You’re looking for the same things as you were looking for in the compiler’s output: "invokedynamic" and classes that aren’t supported in Codename One.

Step 2: Convert the runtime library into a CN1Lib

Once you’ve assessed the language and are optimistic that it’s a good candidate for porting, you can proceed to port the runtime library into Codename One. That language’s runtime library will be distributed in.jar format. You need to convert this into a cn1lib so that it can be used in a Codename One project. If you can get your hands on the source code for the runtime library then the best approach is to paste the source files into a Maven cn1lib project generated with the `cn1lib-archetype` (see Section 1.5, “Codename One libraries”) and try to build it. This has the advantage that it will check the source during compile to ensure that it doesn’t depend on any classes that Codename One doesn’t support.

If you can’t find the sources of the runtime library or they don’t seem to be "buildable," then the next best thing is to get the binary distribution’s jar file and convert it to a cn1lib. This is what you did for the Kotlin runtime library [https://github.com/shannah/codenameone-kotlin].

This procedure exploits the fact that a cn1lib file is a zip file with a specific file structure inside it. The cross-platform Java `.class` files are all contained inside a file named `main.zip`, inside the zip file. This is the **mandatory** file that must be inside a cn1lib.

To make the library easier to use the cn1lib file can also contain a file named "stubs.zip" which includes stubs of the Java sources. When you build a cn1lib using a Maven cn1lib project, it will automatically generate stubs of the source so that the IDE will have access to nice things like Javadoc when using the library. The Kotlin distribution includes a separate jar file with the runtime sources, named `kotlin-runtime-sources.jar`, so you used this as the "stubs." It contains full sources, which isn’t necessary, but it also doesn’t hurt.

Now that you had your two jar files: kotlin-runtime.jar and kotlin-runtime-sources.jar, create a new empty directory and copy them inside. Rename the jars "main.zip" and "stubs.zip" respectively. Then zip up the directory and rename the zip file `kotlin-runtime.cn1lib`.



Building cn1libs manually in this way is a ** bad habit, as it bypasses the API verification step that occurs when building a library project. It's possible, even likely, that the jar files that you convert depend on classes that aren't in the Codename One library, so your library will fail at runtime in unexpected ways. The reason you could do this with kotlin's runtime (with some confidence) is because the bytecodes were already analyzed to ensure that they didn't include anything problematic.

Step 3: Hello world

For your "Hello World" test you will need to create a separate project in your JVM language and produce class files that you will **manually** copy into an appropriate location of your project. You will want to use the **normal** tools for the language and not worry about how it integrates with Codename One. For Kotlin, the steps follow the getting started tutorial on the Kotlin site to create a new Kotlin project in IntelliJ. When Steve ported Mirah, he used a text editor and the mirahc command-line compiler to create your Hello World class. The tools and process will depend on the language.

Create a minimal "hello world" class in the external language runtime you are evaluating.

After building this, the build produces a directory that contains `com/mycompany/myapp/HelloKotlin.class`.

It also produced a.jar file that contains this class.

The easiest way to integrate external code into a Codename One project is to wrap it as a cn1lib and consume it as a Maven dependency. Generate a cn1lib project with the `cn1lib-archetype` (see Section 1.5, "Codename One libraries"), drop the external class files into the `common` module, install it locally with `mvn install`, then add a `pom`-typed `<dependency>` entry to your application's `common/pom.xml`. Using the same procedure as you used to create the kotlin-runtime.cn1lib, wrap your hellokotlin.jar as a cn1lib that exposes the same classes to the application project.



If you're maintaining a legacy Ant project, you can still drop the `.cn1lib` file into the project's `lib` directory and select "Codename One" → "Refresh CN1Libs" to pick it up - but new projects should use the Maven dependency mechanism described above.

Finally, call your library from the start() method of your app:

```
public final class KotlinForm extends Form {
    //...
}
```

If you run this in the Simulator, it should print "Hello from Kotlin" in the output console. If you get

an error, then you can dig in and try to figure out what went wrong using your standard debugging techniques. **EXPECT** an error on the first run. The error will probably be a missing import or something simple.

Step 4: A more complex hello world

For Kotlin, the hello world example app would actually run without the runtime library because it was so simple. It was necessary to add a more complex example to prove the need for the runtime library. It doesn't matter what you do with your more complex example, as long as it doesn't require classes that aren't in Codename One.

If you want to use the Codename One inside your project, you should add the CodenameOne.jar (found inside any Codename One project) to your classpath so that it will compile.

Step 5: Automation and integration

At this point you already have a manual process for incorporating files built with your alternate language into a Codename One project. The process looks like:

1. Use standard tools for your JVM language to write your code.
2. Use the JVM language's standard build tools (for example: command-line compiler, etc.) to compile your code so that you've`.class` files (and optionally a `.jar` file).
3. Wrap your `.class` files in a cn1lib by dropping them into a Maven cn1lib project generated from the `cn1lib-archetype` (see Section 1.5, "Codename One libraries").
4. Install the cn1lib locally with `mvn install` and add it as a `pom`-typed dependency to your Codename One application's `common/pom.xml`.
5. Use your library from the Codename One project.

When Steve first developed Mirah support he automated this process using an ANT script [https://github.com/shannah/CN1MirahNBM/blob/master/src/ca/weblite/codename1/mirah/build.xml]. He also automatically generated some bootstrap code so that he could develop the whole app in Mirah and he wouldn't have to write any Java. For example, this level of integration has limitations.

For example, with this approach alone, you couldn't have two-way dependencies between Java source and Mirah source. Yes, Mirah code could use Java libraries (and it did depend on CodenameOne.jar), and your Java code could use your Mirah code. For example, Mirah **source** code couldn't depend on the Java **source** code in your project. This has to do with the order in which code is compiled. It's a bit of a chicken-and-egg issue. If you're building a project that has Java source code and Mirah source code, you're using two different compilers: mirahc to compile the Mirah files, and javac to compile the Java files. If you're starting from a clean build, and you run mirahc first, then the `.java` files haven't yet been compiled to `.class` files - and thus mirahc can't **reference** them - and any mirah code that depends on those uncompiled Java classes will fail. If you compile the `.java` files first, then you have the opposite problem.

Steve worked around this problem in Mirah by writing your own pseudo-compiler [https://github.com/shannah/mirah-ant/blob/master/src/ca/weblite/asm/JavaExtendedStubCompiler.java] that produced stub class files for the java source that would be referenced by mirahc when compiling the Mirah files. In this way he was able to have two-way dependencies between Java and Mirah in the same project.

Kotlin also supports two-way dependencies, probably using a similar mechanism.

How seamless can you make it

For both the Kotlin and Mirah support, you wanted integration to be seamless. You didn't want users to have to create a separate project for their Kotlin/Mirah code. You wanted them to add a Kotlin/Mirah file into their project and have it * work*. Achieving this level of integration in Kotlin was easy, since they provide an ANT plugin [https://kotlinlang.org/docs/reference/using-ant.html] that essentially allowed you to add one tag inside your `<javac/>` tags:

```
<withKotlin/>
```

And it would automatically handle Kotlin and Java files together: Seamlessly. A few places in a Codename One's build.xml file where you call "javac" so you needed to inject these tags in those places. This injection is performed automatically by the Codename One IntelliJ plugin.

For Mirah, Steve developed his own ANT plugins [https://github.com/shannah/mirah-ant] and Netbeans module [https://github.com/shannah/mirah-nbm] that do something similar in Netbeans.

61.17. Update framework

When Codename One launched in 2012 there was a need to ship updates and fixes faster than the plugin update system, leading to the client lib update system. There was also a need to update the designer tool (resource editor), the GUI builder, and the skins, as well as the built-in builder code (`CodeNameOneBuildClient.jar`).

The Update Framework [https://github.com/codenameone/UpdateCodenameOne] solves many problems in the old systems:

- Download once - if you have many projects the library will download once to the `.codenameone` directory. All the projects will update from local storage
- Skins update automatically - this is hugely important. When you change a theme you need to update it in the skins and if you don't update the skin you might see a difference between the simulator and the device
- Update of designer tools without an IDE plugin update - The IDE plugin update process is slow and tedious. This way a GUI builder fix can ship without a new IDE plugin release. The standalone Settings artifact is versioned with the Maven plugin instead.

This framework should be seamless. You should no longer see the "downloading" message whenever you push an update after your build client is updated. Your system would poll for a new version daily and update when new updates are available.

To force an update check, run `mvn cn1:update` from the project root.

61.17.1. How does it work

You can see the full code here [https://github.com/codenameone/UpdateCodenameOne] the gist of it is

simple. You create a jar called `UpdateCodenameOne.jar` under `~/.codenameone` (`~` represents the users home directory).

An update happens by running this tool with a path to a Codename One project for example:

```
java -jar ~/.codenameone/UpdateCodenameOne.jar path_to_my_codenameone_project
```

For example:

```
java -jar ~/.codenameone/UpdateCodenameOne.jar ~/dev/AccordionDemo
Checking: JavaSE.jar
Checking: CodeNameOneBuildClient.jar
Checking: CLDC11.jar
Checking: CodenameOne.jar
Checking: CodenameOne_SRC.jar
Checking: designer_1.jar
Checking: guibuilder_1.jar
Updating the file: /Users/shai/dev/AccordionDemo/JavaSE.jar
Updating the file: /Users/shai/dev/AccordionDemo/CodeNameOneBuildClient.jar
Updating the file: /Users/shai/dev/AccordionDemo/lib/CLDC11.jar
Updating the file: /Users/shai/dev/AccordionDemo/lib/CodenameOne.jar
Updated project files
```

Notice that no download happened since the files were up-to-date. You can also force a check against the server by adding the force argument as such:

```
java -jar ~/.codenameone/UpdateCodenameOne.jar path_to_my_codenameone_project
```

The way this works under the hood is thought a `Versions.properties` within your directory that lists the versions of local files. That way you know what should be updated.



Exclude `Versions.properties` from Git

Under the `~/.codenameone` directory you have a more detailed `UpdateStatus.properties` file that includes versions of the locally downloaded files. Notice you can delete this file and it will be recreated as all the jars get downloaded over again.

61.17.2. What isn't covered

You will notice 3 big things that aren't covered in this unified framework:

- You don't update cn1libs - whether you would like to update them automatically is unclear
- Versioned builds - a lot of complexity exists in the versioned build system. This might be something you address in the future but for now the framework stays simple.

[1] The old Codename One VM

Chapter 62. Security

Security is a "big word." It implies many things and that's also true for the mobile app development so before you get started lets try to define the scope of security.

You will deal with application security and its communication mechanisms while ignoring everything beyond that scope. Start with a simple fact:

> Codename One applications are secure on the devices by the nature of the mobile OS security.

For most intents and purposes this will be enough, unless you're specifically concerned about security this section isn't for you. Mobile OSes isolate applications from one another so it's hard for an application to damage the OS or even damage/spy on a different app.

The restrictions laid on apps are here to make them extra secure and on top of that Codename One lays a few big advantages in security:

- Codename One code is compiled (unlike for example, PhoneGap/Cordova)
- Codename One obfuscates by default which makes the binaries harder to reverse engineer
- Codename One compiles the UI to native code too which means typical reverse engineering code will have a harder time following
- Codename One disables debug flags so a hacker won't be able to debug your production app on the device

Despite that you still need to keep in mind that the binary could still be reverse engineered and so it's important to avoid storing keys in the client side code. For example, if you have an API key to access a service (for example, Google Cloud key) it needs to be stored in your server and not as a constant in your app!

Each section below discusses some attack vectors against applications and how they can be stopped. Pretty much all these attacks require a sophisticated attacker which will exist for high value targets (for example, bank apps, government etc.).

62.1. Constant obfuscation

One of the first things a hacker will do when compromising an app is look at it. For example, an attacker wanting to exploit a bank's login UI would look at the label next to the login and then search for it in the decompiled code. If the UI has the String "enter username and password" they can search for that.

It won't lead directly to a hack or exploit, but it will narrow down the area of the code where you should look and it makes the first step that much easier. Obfuscation helps as it removes descriptive method names but it can't hide the Strings you use in constants. If an app has a secret key within obfuscating it can make a difference (albeit a slight difference).

Notice that this is a temporary roadblock as any savvy hacker would compile the app and connect a

debugger eventually (although this is blocked in release builds) and would be able to inspect values of variables/flow. But the road to reverse engineering the app would be harder even with a simple xor obfuscation.



This isn't encoding or encryption — it's a simple obfuscation of the data.

There are two simple methods in the `Util` class:

They use a simple xor based obfuscation to make a String less readable. For example, if you've code like this:

```
private static final String SECRET = "Don't let anyone see this/* omitted */";
```

You might be concerned about the secret, then this would make it slightly harder to find out:

```
// Don't let anyone see this/* omitted */
```

Notice that this **isn't secure**, if you have a crucial value that must not be found you need to store it in the server. No alternative exists because everything that's sent to the client can be compromised by a determined hacker.



Use the comment to help you find the string in the code

Codename One's built-in user-specific constants are obfuscated with this method. For example, an app built with Codename One carries some internal data such as the user who built the app, which is now obfuscated. The following small app encodes strings so you can copy and paste them into your app:

```
Form hi = new Form("Encoder", BoxLayout.y());
TextField bla = new TextField("", "Type Text Here", 20, TextArea.ANY);
TextArea encoded = new TextArea();
SpanLabel decoded = new SpanLabel();
hi.addAll(bla, encoded, decoded);
bla.addDataChangedListener((a, b) -> {
    String s = bla.getText();
    String e = Util.xorEncode(s);
    encoded.setText(e);
    decoded.setText(Util.xorDecode(e));
    hi.getContentPane().animateLayout(100);
});

hi.show();
```

This allows you to type in the first text field and the second text area shows the encoded result. A text area is used so copy/paste is easy.

For your convenience this app can be accessed here: https://www.codenameone.com/demos/StringEncoder/index.html

62.2. Storage encryption

Codename One had support for bouncy castle encryption for quite a while but it isn't as intuitive as you'd like it to be. This makes securing/encrypting your app more painful than it should.

Codename One supports full encryption of the `Storage` (notice the distinction, `Storage` isn't `FileSystemStorage`). This is available by installing the bouncy castle cn1lib from the extensions menu then using one line of code



You would want that code within your `init(Object)` method

Notice that you can't use storage or preferences to store this data as it would be encrypted (`Preferences` uses `Storage` internally). You can use a password for this key and it would make it way more secure **but** if a user changes his password you might have a problem. In that case you might need the old password to migrate to a new password.

This works through a new mechanism in storage where you can replace the storage instance with another instance using:

You can leverage that knowledge to change the encryption password on the encryption storage using pseudo-code like this:



It isn't a good idea to replace storage objects when an app is running so this is purely for this special case...

If you use preferences it might be a good idea to set their built-in location to a different path using something like `Preferences.setPreferencesLocation("EncryptedPreferences");`.

This is useful as it prevents the encrypted preferences from colliding with the regular preferences.

62.3. Secure storage and secrets

Some values are too sensitive for `Storage` or `Preferences`, which are plaintext on disk: auth tokens, refresh tokens, and API keys. Codename One gives you two complementary tools for them.

62.3.1. SecureStorage (the platform keychain)

`com.codename1.security.SecureStorage` keeps small string values in the operating system's secure keychain (iOS Keychain Services and the Android Keystore) so they're protected by the platform's secure enclave instead of being written to app storage. The quiet, no-prompt API is three calls:

On a platform without a keychain the base implementation returns `false` / `null`, so treat a `null` read as "not available" and fall back accordingly.

62.3.2. Secrets (fetched from the cloud, never shipped in the binary)

Baking an API key (for example a Google Maps key) into your app is unsafe: anyone can extract it from the binary, and rotating it means shipping a new build. `com.codename1.security.Secrets` solves this. You define the secret once in the Codename One Cloud vault, and the app fetches it at runtime and caches it in `SecureStorage`:

`get` returns the keychain-cached value when present, otherwise it fetches from the vault over TLS, caches it, and returns it. `refresh(name)` forces a fresh fetch (after you rotate the value server-side) and `clear(name)` drops the cached copy.

To publish a secret, open the **Secrets** page of the build console and add it under the app-readable namespace (the `app.` prefix; you reference it by its bare name from `Secrets.get`). Only secrets in that namespace are ever served to a device, so your server-only credentials (such as the store API keys used by commerce validation) stay in the vault and can't be fetched from a client. Authentication uses the build key every cloud build already carries, so there's nothing to wire up; on the simulator or a local build there's no build key, so `get` returns the cached value if any and otherwise `null`.

Build hint	Default	Description
`secrets.cloud.endpoint`	*(Codename One cloud)*	Base URL of the secrets service. Override only if you self-host.

62.4. Disabling screenshots

One of the common security features some apps expect is the ability to block a screenshot. In the past apps like snapchat required that you touch the screen to view a photo to block the ability to grab a screenshot (on iOS). This no longer works…

Blocking screenshots is implemented by classifying the app window as secure on Android. You can enable this via the build hint `android.disableScreenshots=true`. Once that's added screenshots should no longer work for the app, this might impact other things as well such as the task view which will no longer show the screenshot either.

On iOS, you can't prevent a user from taking a static screenshot, but you can block screen capture/recording while it's active. This uses `UIScreen.isCaptured` and its change notification to hide the app contents while a capture session is active. Enable this behavior with the build hint `ios.disableScreenshots=true`.

62.5. Blocking copy & paste

Blocking copy & paste is useful for cases where a device might have spyware installed that monitors the clipboard. This also prevents a user from using a password manager (which rely on the clipboard), those managers could be compromised and thus if you're building a secure app this might be necessary.

You can block copy & paste on Android & iOS. Blocking of copy & paste can be implemented globally or on a specific field.

To block copy & paste globally use:

```
Display.getInstance().setProperty("blockCopyPaste", "true");
```

To block copy & paste on a specific field do:



Notice that the inverse of using `false` might not work as expected

62.6. Blocking jailbreak

iOS & Android are walled gardens which is both a blessing and a curse. Looking at the bright side the walled garden aspect of locked down devices means the devices are more secure by nature. For example, on a PC that was compromised you can detect the banking details of a user logging into a bank. But on a phone it would be much harder due to the deep process isolation.

This isn't true for jailbroken or rooted devices. In these devices security has often been compromised with good intentions (opening up the ecosystem) but it can also be used as a step in a serious attack on an application!

For obvious reasons it's hard to accurately detect a jailbroken or rooted device but when possible if you have a high security app you might want to block the functionality or even raise a "silent alarm" in such a case. To detect this you can use the `isJailbrokenDevice` method as such:

```
if(Display.getInstance().isJailbrokenDevice()) {
    // probably jailbroken or rooted
} else {
   // probably not
}
```

Notice that this isn't right, you can't be 100% sure as there are no official ways to detect jailbreak. That's why it's crucial to encrypt everything and assume the device was compromised to begin with when dealing with sensitive data. Still it's worthwhile to use these APIs to make the life of an attacker a little bit harder.

62.7. Device integrity, RASP & attestation

The `isJailbrokenDevice` check above is a single, coarse signal. For high-security apps (banking, payments, healthcare) Codename One exposes a richer, cross-platform device-integrity API in `com.codename1.security.DeviceIntegrity`. It groups three families of runtime self-protection (RASP) primitives, each available both as a **zero-code build hint** (an automatic launch-time guard) and as a **runtime API** (so your app can make granular, per-transaction decisions). The build hints are wired into the build server, so enabling a feature is just a settings entry away.



Every on-device check is bypassable on a sufficiently compromised device — that's an industry truth, not a Codename One limitation. Use the on-device checks for defense-in-depth and UX (warn, degrade, require step-up auth), but gate genuinely

> sensitive server actions on the **server-verified attestation token** described below, never on a boolean computed on the device itself.

62.7.1. Detecting rooted / jailbroken / tampered devices (RASP)

Codename One ships several launch-time hard gates that terminate the app on a compromised device. They're enabled through build hints and need no code:

- `android.rootCheck=true` — bundles the RootBeer library and exits at launch on a rooted device (`su` binaries, test-keys / custom ROM signals, Magisk indicators, dangerous system properties).
- `android.fridaDetection=true` — detects the Frida dynamic-instrumentation toolkit (the core RASP threat) and exits.
- `ios.detectJailbreak=true` — exits at launch on a jailbroken iOS device (MobileSubstrate / Liberty / FridaGadget dylibs, Cydia/apt paths, sandbox-escape probes).

When you'd rather respond in-app than be hard-killed — for example to block only a single risky operation, raise a silent alarm, or require additional authentication — use the non-exiting runtime report instead:

```
if (DeviceIntegrity.isDeviceCompromised()) {
    // root / jailbreak / Frida / emulator detected -- do not hard-exit,
    // instead degrade: disable the transfer button, require step-up auth, etc.
    for (String reason : DeviceIntegrity.getCompromiseReasons()) {
        // reason is a machine-readable code: "root", "frida", "emulator", "jailbreak"
        Log.p("Device compromise signal: " + reason);
    }
}
```

`isDeviceCompromised()` aggregates the same root/jailbreak/Frida signals as the launch gates plus emulator heuristics, but never terminates the app. On the simulator and unsupported ports it simply returns `false`.



> If your security policy mandates a specific commercial Mobile Threat Defense (MTD) / RASP vendor (Zimperium, Promon SHIELD, etc.), you integrate that vendor's SDK through the standard `android.gradleDep` / `ios.pods` build hints — Codename One's built-in checks complement, but don't claim to replace, a dedicated MTD product.

62.7.2. Platform attestation: Google Play Integrity & Apple App Attest

The trustworthy way to verify device and app integrity is hardware-backed attestation: Google Play Integrity on Android and Apple App Attest on iOS. Both produce a **signed token** that your backend verifies with Google / Apple. This is the mechanism to use when you must, for example, block an outward transfer to a newly added beneficiary on a device that fails integrity.

Enable it with a build hint, which bundles the SDK (Android) or links the entitlement (iOS):

- `android.playIntegrity=true` — bundles the Play Integrity SDK and enables the runtime token API. Optionally add `android.playIntegrity.verifyUrl=https://your-backend/integrity` to **also** attest at launch on a background thread and exit if your backend rejects the token. `android.playIntegrityVersion` overrides the bundled SDK version.
- `ios.appAttest=true` — enables App Attest (DeviceCheck.framework) and the runtime token API. `ios.appAttest.environment` is `development` (debug builds) or `production` (release) by default.

The runtime API is identical on both platforms — request a token bound to a fresh server nonce, then POST it to your backend for verification:

`DeviceIntegrity.isAttestationSupported()` reports whether attestation is available and was bundled into this build. On the simulator, on unsupported ports, or when the build hint is off, `requestIntegrityToken()` completes with an error rather than throwing.



The block-the-transfer decision lives in **your backend**, gated on the verified verdict. Codename One's job is to reliably deliver the attested token; the policy enforcement — for example blocking new-beneficiary transfers when integrity fails and biometric controls aren't enabled — is application + server logic that consumes this API.

`DeviceIntegrity` hands you the raw platform blob and leaves verification, policy, replay protection, key management and rotation to you. If you'd rather not build that, Chapter 63, *App Shield* is the managed layer on top of exactly this API: the attestation is verified server-side against Apple and Google, evaluated against a policy you control, and turned into a short-lived signed token your backend checks with a few lines of middleware. Both remain available and this API keeps working unchanged.

62.7.3. Defending against accessibility-service abuse (Android)

A common Android malware technique is to abuse the accessibility services API for overlays, remote control, and on-screen text extraction (including reading text destined for text-to-speech). Codename One lets you detect untrusted accessibility services and lock down sensitive screens.

The `android.accessibilityGuard=true` build hint checks the enabled accessibility services at launch and, by default, exits when an untrusted one is active. Tune it with:

- `android.accessibilityGuard.allow=com.google.android.marvin.talkback,com.example.trusted` — a comma-separated allow-list of accessibility-service **packages** the user legitimately relies on (such as a screen reader). Any enabled service outside this list trips the guard.
- `android.accessibilityGuard.mode=exit|warn` — `exit` (default) terminates the app; `warn` only logs, leaving the decision to your runtime code.

At runtime you get the same information for granular control:

```
// Enumerate the enabled accessibility services (component ids "pkg/.Service").
String[] services = DeviceIntegrity.getEnabledAccessibilityServices();

// True if any enabled service is outside your trusted allow-list.
```

```
if (DeviceIntegrity.hasUntrustedAccessibilityService("com.google.android.marvin.talkback")) {
    // Likely accessibility-abusing malware -- block the sensitive action.
}
```

For the screens that actually matter (PIN entry, transfer confirmation), mark them secure so the OS blocks screenshots, screen recording, and accessibility screen scraping while they're displayed:

```
// Entering a sensitive screen (Android FLAG_SECURE; no-op on iOS):
DeviceIntegrity.setSecureScreen(true);
// null leaving it:
DeviceIntegrity.setSecureScreen(false);
```



Accessibility-service enumeration and `setSecureScreen()` are Android concepts; on iOS they're no-ops (iOS has no equivalent third-party accessibility-service threat model). Use `ios.disableScreenshots=true` to block screen capture on iOS as described in Section 62.4, "Disabling screenshots".

62.8. Strong Android certificates

When Android launched RSA1024 with SHA1 was considered strong enough for the foreseeable future, this hasn't changed completely but the recommendation today is to use stronger ciphers for signing & encrypting as those can be compromised.

APKs are signed as part of the build process when you upload an app to the Google Play Store. This process seems redundant as you generate the signature/certificate ourselves (unlike Apple which generates it for you). But, this is a crucial step as it allows the device to verify upgrades and make sure a new update is from the same original author!

This means that if a hacker takes over your account on Google Play, he still won't be able to ship fake updates to your apps without your certificate. That's important since if a hacker would have access to your certificate he could create an app update that would send him all the users private information for example, if you're a bank this could be a disaster.

Android launched with RSA1024/SHA1 as the signing certificates. This was good enough at the time and is still pretty secure. But, these algorithms are eroding and it's conceivable that within the 10-15 year lifetime of an app they might be compromised using powerful hardware. That's why Google introduced support for stronger cryptographic signing in newer versions of Android and you can use that.

62.8.1. The bad news

A downside exists…

Google introduced that capability in Android 4.3 so using these new keys will break compatibility with older devices. If you're building a highly secure app this is probably a tradeoff you should accept. If not this might not be worth it for some theoretical benefit.

Furthermore, if your app is already shipping you're out of luck. Due to the obvious security implications once you shipped an app the certificate is final. Google doesn't provide a way to update the certificate of a shipping app. Thus this feature applies to apps that aren't yet in the play store.

62.8.2. The good

If you're building a new app this is pretty easy to integrate and requires no changes on your part. Just a new certificate. You can generate the new secure key using instructions in articles like this one [https://guardianproject.info/2015/12/29/how-to-migrate-your-android-apps-signing-key/].

If you're using Codename One Setting you can check the box to generate an SHA512 key which will harden the security for the APK.

62.9. Certificate pinning

When you connect to HTTPS servers your networking code checks the certificate on the server. If the certificate was issued by a trusted certificate authority then the connection goes through, otherwise it fails. Imagine you're sitting in a coffee shop connected to the local Wi-Fi, and you try to connect to Gmail to check your email. Since you use HTTPS to Google you trust the connection is secure.



What if the coffee shop was hacked and the router is listening in on everything?

HTTPS is encrypted and the way encryption works is through the certificate. The server sends you a certificate and you can use that to send encrypted data to it.



What if the router grabs the server's certificate and communicates with Google on your behalf?

This won't work since the data you send to the server is encrypted with the certificate from the server.



What if the router sends its own "fake certificate"?

That won't work either. All certificates are signed by a "certificate authority" indicating that a google.com certificate is valid.



What if an attacker could get a fake certificate authorized by a real certificate authority?

That's a problem!

It's obviously hard to do but if someone was able to do this he could execute a "man in the middle" attack as described above. People were able to fool certificate authorities in the past and gain fake certificates using various methods so this is possible and probably doable for any government level attacker.

62.9.1. Certificate pinning

This is the attack certificate pinning (or SSL pinning) aims to prevent. You code into your app the "fingerprint" of the certificate that's "good" and thus prevent the app from working when the certificate is changed. This might break the app if you replace the certificate at some point but that might be reasonable in such a case.

To do this you have a cn1lib [https://github.com/codenameone/SSLCertificateFingerprint/]. It fetches the certificate fingerprint from the server so you can check the fingerprint against a list of authorized keys. Run `mvn cn1:settings`, open **Extensions**, and install `SSLCertificateFingerprint`. Then use something like this to verify your server:



Fingerprints coded into the app are the part that hurts: the certificate has to be replaced eventually, and a hardcoded pin turns that into an app store release with every un-updated install broken until it lands. Chapter 63, *App Shield* pins the public key rather than the whole certificate, delivers the pin set over the air with the attestation token, and refuses to publish a set that would brick the app — which is the same protection without the release cycle.

Notice that once connection is established you don't need to verify again for the current application run.

62.10. Cryptographic primitives

The `com.codename1.security` package gives you the cryptographic building blocks you need for typical mobile-app concerns — hashing, message authentication, symmetric and asymmetric encryption, digital signatures, JSON Web Tokens, and time/counter-based one-time passwords. The supporting `com.codename1.components.OtpField` component covers the UI side of OTP entry.

The hash and HMAC primitives are implemented in portable Java so they produce identical output on every platform without depending on any external library. Symmetric and asymmetric encryption, signing, and the secure RNG go through each platform's native crypto provider via Codename One's implementation layer:

- The simulator (JavaSE) and Android route through the JRE's `java.security` and `javax.crypto`.
- iOS routes through Apple's CommonCrypto and the Security framework (via `Ports/iOSPort/nativeSources/CN1Crypto.{h,m}` and the corresponding `IOSImplementation` overrides).

Application code never sees these differences — you call the same `com.codename1.security` classes regardless of the platform you are running on.



Cryptographic operations only throw `com.codename1.security.CryptoException`, an unchecked exception. You can let it bubble up or catch it; you don't have to declare it.

62.10.1. Hashing

`com.codename1.security.Hash` implements MD5, SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512. SHA-256 is the recommended default for new code; MD5 and SHA-1 are exposed only for compatibility with older protocols since both are broken for collision resistance:

The algorithm identifier strings (`Hash.MD5`, `Hash.SHA256`, etc.) are also accepted with or without dashes and in any case (`"sha256"` and `"SHA-256"` are equivalent).

62.10.2. Message authentication (HMAC)

`com.codename1.security.Hmac` implements HMAC (RFC 2104) on top of any of the hash algorithms above. Use HMAC whenever a message authentication code is needed — API request signing, session cookie tamper detection, JWT signing with the `HS` family, or TOTP token generation:

When comparing authentication tags, use `Hmac.constantTimeEquals(a, b)` instead of `Arrays.equals(a, b)` — the latter short-circuits on the first mismatch and is vulnerable to timing attacks.

62.10.3. Secure random numbers

`com.codename1.security.SecureRandom` exposes the platform's cryptographically secure RNG (`SecRandomCopyBytes` on iOS, `/dev/urandom` on Linux/Android, etc.). Use it for keys, nonces, salts, password reset tokens, and any other security-sensitive value. Never use `java.util.Random` or `Math.random()` for these purposes:

```
byte[] iv  = SecureRandom.bytes(12);      // 12-byte AES-GCM nonce
int    pin = SecureRandom.intBelow(1_000_000);   // bias-free 6-digit code
long   id  = SecureRandom.longBelow(1L << 53);
```

62.10.4. Symmetric encryption (AES)

`com.codename1.security.Cipher` exposes AES through the platform's native AES implementation. AES-GCM is the recommended default because it's authenticated — a single tag-mismatch failure detects any tampering of the ciphertext or the associated data:

The `Cipher` class exposes the standard JCE-style transformation strings as constants: `Cipher.AES_GCM`, `Cipher.AES_CBC_PKCS5`, `Cipher.AES_CBC`, and `Cipher.AES_ECB_PKCS5`. For GCM, the 16-byte authentication tag is appended to the ciphertext (matching the JCE convention). ECB is exposed only for interop with legacy systems — it leaks structure and should never be used for new designs.

62.10.5. Asymmetric encryption and digital signatures (RSA, ECDSA)

`com.codename1.security.Cipher` also covers RSA encryption, and `com.codename1.security.Signature` covers digital signatures (RSA and ECDSA). Keys are represented by `com.codename1.security.PublicKey` and `com.codename1.security.PrivateKey`, which wrap X.509 SubjectPublicKeyInfo and PKCS#8 DER blobs respectively — the same encodings used by OpenSSL:

To use a key that was generated outside the app, feed the DER bytes to the static factory methods:

The `Signature` class exposes the JCE algorithm strings as constants: `SHA256_WITH_RSA`, `SHA384_WITH_RSA`, `SHA512_WITH_RSA`, `SHA256_WITH_ECDSA`, `SHA384_WITH_ECDSA`, and `SHA512_WITH_ECDSA`.

62.10.6. JSON Web Tokens

`com.codename1.security.Jwt` provides JWT (RFC 7519) signing and verification. The `HS` family (HMAC-SHA-2) is pure Java so it works on every platform; the `RS` family (RSA-PKCS#1 v1.5 with SHA-2) and the `ES` family (ECDSA with SHA-2) route through the platform's native crypto:

```
Map<String, Object> claims = new LinkedHashMap<String, Object>();
claims.put("sub", "user-123");
claims.put("exp", System.currentTimeMillis() / 1000 + 3600);

// Sign with a shared secret (HS256)
String token = Jwt.signHs256(claims, "secret".getBytes("UTF-8"));

// Parse + verify
Jwt parsed = Jwt.parse(token);
if (!parsed.verifyHs256("secret".getBytes("UTF-8"))) {
    throw new SecurityException("bad signature");
}
String subject = (String) parsed.getClaim("sub");
```

For RSA-signed tokens (RS256/384/512) and ECDSA-signed tokens (ES256/384/512), pass the key to the dynamic overloads:



`Jwt.parse(token)` doesn't verify the signature — you must call one of the `verify*` methods afterwards. Tokens with an `alg` header of `none` are rejected by `verify` unless you explicitly call `setVerifyAllowNoneAlgorithm(true)` on the parsed JWT. Accepting unsigned tokens is a critical security bug in almost every deployment, so leave it off unless you have decided that the transport is trusted.

62.10.7. HOTP / TOTP one-time passwords

`com.codename1.security.Otp` implements counter-based HOTP (RFC 4226) and time-based TOTP (RFC 6238). The output is identical to standard authenticator apps (Google Authenticator, Microsoft Authenticator, 1Password, Authy, etc.) — you can use this class to generate codes inside your app, to validate codes sent by an authenticator on a sign-in flow, or for SMS-based two-factor flows.

Shared secrets are commonly distributed as Base32 strings (the format embedded in QR codes by authenticator apps). `com.codename1.security.Base32` handles the encoding:

For HOTP, supply the counter explicitly and increment it after every successful authentication:

62.10.8. Walkthrough: Adding two-factor authentication to a sign-in flow

The typical TOTP-based two-factor authentication flow has two distinct phases. The first is *enrolment*, where the server provisions a shared secret to the user's authenticator app; the second is *verification*, which happens on every subsequent sign-in.



The TOTP secret should be generated and stored on your **server**, not on the Codename One client. The client only ever sees the secret long enough to display the enrolment QR code, then forgets it. Verification on every subsequent sign-in is also a server-side concern (the client just sends the code the user types in). The code samples below put the cryptography on the client only to keep the example self-contained — in production, the server does the actual generation and verification.

Phase 1 — Enrollment. When the user opts into 2FA, generate a 20-byte random secret and show it to their authenticator app as a QR code that encodes a standard `otpauth://totp/…` URI:

```
// 1. Generate (or fetch from your server) a 20-byte secret -- 160 bits is the
//    de-facto standard for authenticator compatibility.
byte[] secret = SecureRandom.bytes(20);

// 2. Build the otpauth URI. The authenticator app stores the secret against
//    the "Acme Bank: alice@example.com" label and tags it with the issuer
//    so it can show "Acme Bank" next to the rotating code.
String uri = Otp.otpauthUri("Acme Bank", "alice@example.com", secret);

// 3. Render `uri` as a QR code and show it on screen for the user to scan.
//    See "Rendering the QR code" below.
```



The maximum-compatibility settings — 6 digits, 30 second step, SHA-1 — are the defaults of `Otp.otpauthUri(issuer, accountName, secret)`. Pass digits / step / algorithm explicitly via the full overload only if you have a specific reason; some authenticator apps don't recognize anything except the defaults.

Rendering the QR code. Codename One core doesn't currently ship a QR-code generator. Pick whichever option fits your deployment:

- **Server-side render** (simplest). Send `uri` to your backend over HTTPS and have it return a PNG. The Google Charts API endpoint `https://chart.googleapis.com/chart?cht=qr&chs=300x300&chl={url-encoded-uri}` and the open-source `https://api.qrserver.com/v1/create-qr-code/?data={url-encoded-uri}` both work; the latter doesn't require Google API setup. Render the resulting PNG with `URLImage` or `EncodedImage.create(bytes)`.
- **Use a QR cn1lib.** Community cn1libs such as `QRMaker` provide native generation. Add as a normal cn1lib dependency.
- **Display the secret as a typed value.** If you can't render a QR code, show the user the Base32-encoded secret (the value after `secret=` in `uri`) and ask them to type it manually into their authenticator's "set up by hand" flow. Less convenient but works on every platform.

Phase 2 — Verification. On every sign-in after enrolment, ask the user for the current six-digit code and verify it against the stored secret. Use [com.codename1.components.OtpField] for the input:

Server-side verification. When your backend does the verification, the corresponding Java looks the same — `Otp.verifyTotp(storedSecret, codeFromClient, 1)`. Make sure you rate-limit verification attempts (for example, lock the account after 5 wrong codes within a minute) and log failures, since the 6-digit search space is small enough to be brute-forced without that.

62.10.9. OTP input widget

`com.codename1.components.OtpField` is a segmented input — one box per character, that advances to the next box as the user types and steps back on backspace. This is the standard pattern for SMS confirmation and authenticator-app entry screens:

Style the boxes through the UIIDs `OtpField` (the container) and `OtpDigit` (each box). The component supports alphanumeric input via the two-argument constructor (`new OtpField(8, false)`), and pasting a full code into any one box distributes the characters across the remaining boxes automatically.

62.10.10. Algorithm identifier reference

The string constants exposed by these classes are listed below so you can pick them out at a glance.

Class	Constant	Algorithm string
`Hash`	`MD5`, `SHA1`, `SHA224`, `SHA256`, `SHA384`, `SHA512`	`MD5`, `SHA-1`, `SHA-224`, `SHA-256`, `SHA-384`, `SHA-512`
`Cipher`	`AES_GCM`	`AES/GCM/NoPadding`
`Cipher`	`AES_CBC_PKCS5`	`AES/CBC/PKCS5Padding`
`Cipher`	`AES_CBC`	`AES/CBC/NoPadding`
`Cipher`	`AES_ECB_PKCS5`	`AES/ECB/PKCS5Padding` (legacy)
`Cipher`	`RSA_OAEP_SHA256`	`RSA/ECB/OAEPWithSHA-256AndMGF1Padding`
`Cipher`	`RSA_PKCS1`	`RSA/ECB/PKCS1Padding` (legacy)
`Signature`	`SHA256_WITH_RSA`, `SHA384_WITH_RSA`, `SHA512_WITH_RSA`	`SHA256withRSA`, `SHA384withRSA`, `SHA512withRSA`
`Signature`	`SHA256_WITH_ECDSA`, `SHA384_WITH_ECDSA`, `SHA512_WITH_ECDSA`	`SHA256withECDSA`, `SHA384withECDSA`, `SHA512withECDSA`
`Jwt`	`HS256`, `HS384`, `HS512`, `RS256`, `RS384`, `RS512`, `ES256`, `ES384`, `ES512`	Same as the JWT `alg` header value

62.10.11. Key sizes and recommended defaults

- **AES**: 256 bits (`KeyGenerator.aes(256)`) for new code. 128 bits is also acceptable and faster; pick 192 only if you have a specific reason.

- **RSA**: 2048 bits minimum (`KeyGenerator.rsa(2048)`). Use 3072 or 4096 if you need a longer security margin — key generation gets slow above 2048 so run it on a background thread.
- **HMAC**: a key at least as long as the hash output (32 bytes for HMAC-SHA-256). `KeyGenerator.hmac(256)` returns a key of the requested bit length.
- **TOTP** / **HOTP**: 160 bits (20 bytes) is the de-facto standard for authenticator-app compatibility.

Chapter 63. App Shield

The security chapter covers a lot of ground that ends at the same place: every check runs on the device, and a device the attacker fully controls is a device where every check can be patched out. Root detection returns false. The pinning code is skipped. The jailbreak probe is stubbed. None of it reaches your server, which is where the money is.

App Shield closes that loop. The app obtains a short-lived, Codename One-signed **attestation token** backed by a server-side verification of Apple's App Attest and Google's Play Integrity statements, and your own API refuses to serve a request that doesn't carry a valid one. The statement that matters is made by a party the attacker doesn't control, and your backend can check it in about twenty lines of middleware.



App Shield doesn't make an app unhackable, and no product does. What it changes is the cost: scripted abuse of your API goes from "curl in an afternoon" to "reverse-engineer a native binary, and start again after every rotation," and your backend gets a cryptographically verifiable statement it can act on. Be careful not to promise more than that, internally or in marketing.

This is an **Enterprise** feature. The public API in `com.codename1.security.shield` ships in the open-source core and is safe to call in any build — in a build without the attestation engine it degrades to a documented no-op rather than failing — but tokens are only ever minted for entitled accounts.

63.1. How it fits together

Three parties are involved: the app, the Codename One attestation service, and your own API. A protected request goes through five steps.

1. The app asks the attestation service for a challenge, and the service answers with a nonce it generated. The nonce is what stops a token captured once from being replayed later.
2. The app asks the platform to attest itself against that nonce. Apple's App Attest or Google's Play Integrity produces a statement signed by hardware the app can't forge.
3. The app sends that statement to the attestation service, together with any runtime signals it has collected.
4. The service verifies the statement with Apple or Google, applies the policy you configured, and answers with a short-lived ES256 JWT and the current certificate pin set.
5. The app sends that token to **your** API in the `X-CN1-Attest` header, and your middleware verifies its signature against the public keys the service publishes.

Steps 1 to 4 are the framework's job and happen on the network thread. Step 5 is automatic for every `ConnectionRequest` to a host you registered. The decision — what to do about a device the service declined — is yours, and it belongs in step 5, on your server.

63.2. Turning it on

Two things have to be true: the account has to be entitled, and the build has to ask for it.

Build hint	Default	Description
`shield.enabled`	`false`	Master switch. The build server injects the hardening engine and wires it into the app's startup. A build that asks for it without an Enterprise subscription, or on a server where the engine is unavailable, **fails with an explanation** rather than producing an unprotected app and saying nothing.
`shield.policyVersion`	*(service default)*	The policy generation this build was compiled against.
`shield.pins`	*(none)*	The initial pin set, so the first launch on a new install is pinned before it has ever reached the service. Comma-separated base64 SHA-256 SPKI digests.
`shield.jwksPin`	*(none)*	Pins the key that signs the tokens, so a compromised DNS answer can't introduce a signing key the app will trust.
`shield.attestationVerifyUrl`	*(Codename One cloud)*	Override for a private deployment of the verification service.
`shield.hardFail`	`false`	The hardening engine's own failure policy: `false` reports a problem inside the engine as a signal and lets the app run, `true` makes it fatal. Leave the default unless support advises otherwise.
`ios.appAttest.environment`	`development` for debug builds, `production` for release	The App Attest environment written into the entitlement.



The engine is injected at build time and isn't present in a source checkout, in the simulator, or in an unentitled build. That's deliberate: the detection logic is the part an attacker most wants to read. Everything in this chapter that's **API** is open source; only the engine behind it isn't.

63.3. Protecting hosts

Registration is the whole configuration. A host you don't register is never touched — no header, no certificate check, no behaviour change of any kind.

```
AppShield.init(new ShieldConfig()
        // Money moves here, so refuse the request when no token can be obtained.
        .protect("api.mybank.example", HostPolicy.ENFORCED)
```

```
        // Everything else under the domain gets a token when one is available.
        .protect("*.mybank.example", HostPolicy.PROTECTED));

// From here on nothing changes at the call sites. A request to a protected
// host is given its token and its certificate check on the network thread.
ConnectionRequest r = new ConnectionRequest("https://api.mybank.example/transfer", true);
NetworkManager.getInstance().addToQueueAndWait(r);
```

`HostPolicy` is two independent decisions plus what to do when the shield can't deliver:

Policy	Token	Pins	When no token can be obtained
`HostPolicy.UNPROTECTED`	no	no	Not applicable. This is what an unregistered host gets.
`HostPolicy.PROTECTED`	yes	yes	**Fail open.** The request goes out without a token and your backend decides. Right for most hosts.
`HostPolicy.ENFORCED`	yes	yes	**Fail closed.** `ShieldException` is thrown and nothing is sent. Right for the handful of endpoints that move money or read personal data.

Fail-closed is a promise that a request without a valid token never leaves the device, so it applies to the awkward cases too: a protected host reached over plain `http` is refused rather than handed the bearer token in clear text, and a shield that was interrupted before it finished starting refuses rather than guessing.

If most of your hosts want the same failure mode, set it once and register them the short way:

```
AppShield.init(new ShieldConfig()
        .defaultFailureMode(FailureMode.CLOSED)
        // Registered the short way, so it picks up the default above.
        .protect("api.mybank.example")
        // And this one opts back out, because an outage here should degrade
        // rather than block the app.
        .protect("images.mybank.example", HostPolicy.PROTECTED));
```



The default failure mode is read when a host's policy is **resolved**, not when it's registered, so `.protect(h).defaultFailureMode(CLOSED)` and the same two calls in the other order mean the same thing.

63.3.1. Wildcards and hosts discovered at runtime

A pattern may be an exact host or a leading `.` wildcard covering its subdomains — `.example.com` matches `api.example.com` and `a.b.example.com`, but not the apex `example.com` and not `example.com.somewhere-else.test`. The most specific rule wins, so an exact registration is never relaxed by a wildcard.

Backends the app only learns about after it starts — a regional endpoint chosen at login, a tenant-specific host — are registered the same way, with the same matching:

```
// Exact host or a "*." wildcard, resolved the way the configured ones are:
// the most specific rule wins, and a runtime registration beats a configured
// one for the same pattern.
AppShield.addProtectedHost(region + ".api.mybank.example", HostPolicy.ENFORCED);
AppShield.addProtectedHost("*.cdn.mybank.example", HostPolicy.PROTECTED);
```

63.4. What happens to a request

Once `init()` has run, `ConnectionRequest`, `Rest`, `RequestBuilder` and anything else built on `NetworkManager` are covered without further code. For a request to a protected host, on the network thread:

- It waits for initialization to finish if a cold start is still in progress, rather than treating "not ready yet" as "not protected."
- Any token the shield attached earlier is taken back off before the request is reused — a redirect reuses the same request object, and a bearer token that follows a redirect off a protected host has been handed to whoever the redirect points at. A header your own `onRedirect()` installed under the same name is left alone: the shield removes its token only while the value is still the token it attached.
- The token is added under the token header, unless the URL isn't `https`, in which case the host's failure mode decides.
- The certificate chain is collected and checked against the pin set, before any request body is written.

The response side is just as narrow: the guard sees the status code and the headers it asked for, which is how a backend tells the app its **token** was refused as opposed to the user's credentials.

63.4.1. The token header

The default is `X-CN1-Attest`. It isn't `Authorization`, by design: that slot belongs to your own user authentication, and the two answer different questions — who the user is, versus whether this is a genuine unmodified app on an uncompromised device. Your backend needs both, so they have to compose.

`ShieldConfig.tokenHeader(String)` changes it, and refuses names that can't carry a token rather than accepting them and failing out of sight later:

- Anything that isn't a legal HTTP field name. A name with a space or a separator in it isn't the header you asked for by the time it reaches the server — and it also slips past every check below, because `Cookie ` isn't `Cookie`.
- `Content-Type`, which `ConnectionRequest` routes to the request's media type rather than the header map, so the token would replace the content type and survive the redirect cleanup.
- Your app's cookie header, whatever it has been renamed to, because the cookie string is written

after the request's own headers and would overwrite the token after `attach()` reported success.

- Headers the transport owns — framing, routing, and the hop-by-hop names a correctly behaving proxy consumes and doesn't forward.

63.5. Status, and what to do about it

The most important distinction in the API is between "the shield couldn't be reached" and "the shield looked at this device and said no." Reacting to the first the way you react to the second is how an app locks a paying customer out on a train.

Status	Transient	Meaning
`OK`	-	A valid token is in hand.
`UNPROTECTED`	no	This build has no attestation engine. Nothing is enforced.
`NOT_INITIALIZED`	no	`init()` hasn't run, or was interrupted.
`NO_NETWORK`, `POOR_NETWORK`	yes	The service couldn't be reached.
`SERVICE_DOWN`	yes	The service answered with an error.
`RATE_LIMITED`	yes	This device is being throttled. Retry after a backoff.
`REJECTED`	no	The service evaluated this device and declined to mint a token.
`PIN_MISMATCH`	no	A pinned host presented a chain matching none of its pins. Nothing was sent.

`ShieldStatus` is a constant **object** rather than an enum on purpose: the vocabulary is visible on the wire and the service must be able to add a status without breaking apps built against an older release. Compare with `equals`, branch on `isSuccess()` and `isTransient()`, and treat an id you don't recognise as a non-success.

Listeners are delivered on the EDT, so they may touch the UI directly. The framework never shows a dialog and never terminates the app over a shield event — what the user sees is entirely your decision:

```
AppShield.addListener(new ShieldListener() {
    public void statusChanged(ShieldStatus status) {
        if (status.isSuccess()) {
            return;
        }
        if (status.isTransient()) {
            // The service could not be reached. This is a bad connection, not a
            // bad device -- treating it like a rejection is how an app locks out
            // users on a train.
            Log.p("AppShield: attestation unavailable (" + status.getId() + ")");
            return;
        }
        // REJECTED or PIN_MISMATCH: the service evaluated this device and
```

```
        // declined, so degrade rather than retry.
        disableTransfers();
    }

    public void signalRaised(ShieldSignal signal) {
        // Informational. The server decides what a signal means -- an emulator
        // signal is normal on a developer's machine.
        Log.p("AppShield signal " + signal.getId() + " severity "
                + signal.getSeverity());
    }

    public void tokenRefreshed(ShieldToken token) {
    }
});
```

63.6. Getting a token yourself

For anything the guard doesn't cover — a third-party HTTP client, a payload you sign yourself, a handshake — ask for the token directly. `fetchToken()` is asynchronous and returns immediately, including during a cold start:

```
AsyncResource<ShieldToken> pending = AppShield.fetchToken();
pending.ready(new SuccessCallback<ShieldToken>() {
    public void onSucess(ShieldToken token) {
        sendToMyBackend(token.getValue());
    }
});
pending.except(new SuccessCallback<Throwable>() {
    public void onSucess(Throwable error) {
        ShieldStatus status = ((ShieldException) error).getStatus();
        Log.p("AppShield: no token (" + status.getId() + ")");
    }
});
```

Pass a binding value to tie the token to one specific request, so a token lifted from it can't be replayed onto another:

```
// The digest travels in the token, so a token lifted off this request and
// replayed on another one does not verify against the other one's body.
AppShield.fetchToken(sha256(payload)).ready(new SuccessCallback<ShieldToken>() {
    public void onSucess(ShieldToken token) {
        sendToMyBackend(token.getValue());
    }
});
```

`AppShield.attach(ConnectionRequest)` does the same work synchronously for a request you are about to send. It blocks on a network round trip, so it must not be called on the EDT:

```
ConnectionRequest r = new ConnectionRequest("https://api.mybank.example/transfer", true);
try {
    // Blocks on the attestation round trip, so never on the EDT. In normal use
    // the guard does this for you on the network thread.
    AppShield.attach(r);
} catch (ShieldException err) {
    if (ShieldStatus.PIN_MISMATCH.equals(err.getStatus())) {
        // The chain presented for a pinned host matched no pin. Nothing was
        // sent, and this is the one status worth surfacing to the user: it
        // usually means the connection is being intercepted.
        warnAboutInterception();
        return;
    }
    // A fail-closed host with no token. Retry later if the status is transient.
    Log.e(err);
    return;
}
NetworkManager.getInstance().addToQueueAndWait(r);
```

Token lifetime is measured from a monotonic clock, never from `exp` against the device clock, because on a rooted device the wall clock is whatever the attacker wants it to be. `getCachedToken()` returns the current token without any network access, and `invalidateToken()` discards it so the next request re-attests — which is what to call when your backend tells you the token itself was refused:

```
ConnectionRequest r = new ConnectionRequest("https://api.mybank.example/transfer", true) {
    @Override
    protected void readHeaders(Object connection) throws IOException {
        // Your middleware sets this header when the ATTESTATION was the problem,
        // as opposed to the user's own credentials. Without it a 401 or 403 is
        // ambiguous, and an app re-attests its way through a wrong password.
        if (getHeader(connection, AppShield.REJECT_HEADER) != null) {
            AppShield.invalidateToken();
        }
    }
};
NetworkManager.getInstance().addToQueueAndWait(r);
```



`AppShield.REJECT_HEADER` is `X-CN1-Attest-Reject`. Have your middleware set it only when the **token** was the problem: a protected API normally carries your own user authentication too, and an app that reads every 401 as an attestation failure re-attests its way through a wrong password.

63.7. Certificate pinning

Pins are published by the service and refreshed with the token, so rotating a certificate is a console action rather than an app store release. They're pinned over the **subject public key info**, so renewing a certificate on the same key pair doesn't invalidate the pin, and a chain matches if any

certificate in it matches any pin for the host — which is what makes it safe to pin an issuing CA as the backup.

Pinning is the one part of the shield that can take an app offline for a reason the developer can't fix remotely, so the failure behaviour is asymmetric by design:

Condition	Behaviour
Host has no pins	Never enforced. Covers first run, a cold start with no network, and any host the service hasn't published pins for.
Pin fetch failed	Never fails a request. The last known set is kept.
Set past its soft expiry	Still enforced; a refresh is attempted.
Set past its hard expiry	Dropped, and enforcement stops. A device offline for weeks loses pinning, not the app.
Chain matches a pin	Request proceeds.
Host has pins, chain matches none	**Fail closed.** `ShieldException` with `PIN_MISMATCH`, before any body is written.

```
PinSet pins = AppShield.getPinSet();
if (!pins.isEnforcedFor("api.mybank.example")) {
    // No published pins for this host yet, or the set has hard-expired on a
    // long-offline device. Pinning is not enforced, and the app still works.
    Log.p("AppShield: no pins in force (version " + pins.getVersion() + ")");
}
```



A pin mismatch surfaces as an exception on the request, not as a silent empty response. If you swallow `IOException` at your call sites you won't see it, and an intercepted connection is the one failure worth telling the user about.

63.8. Runtime signals

The framework's own detections — root, jailbreak, hooking frameworks, emulators, debuggers, repackaging, untrusted accessibility services — are reported to `ShieldSignals` rather than acted on locally, and ride along with the next token fetch. There is no separate beacon to block and no extra battery cost.

This is a deliberate inversion of the older `exit(0)` style checks. Terminating the app is trivially patched out, destroys the telemetry that makes the feature useful, and turns a detection into a crash report. **The client reports; the service decides.** An emulator signal is normal on a developer's machine; ten high-severity signals from one device in an hour is a device the service can keep flagging on its next clean-looking attestation.

Your own code can report what only it can notice — a server-side consistency check that failed, a sequence of actions no human performs:

```
// Severity is 0-100. The service applies the policy; the client only reports.
ShieldSignals.add("serverStateMismatch", 60,
        "balance disagreed with the ledger after a retry");

// The framework's own detections arrive the same way.
for (ShieldSignal s : AppShield.getSignals()) {
    if (ShieldSignal.HOOK.equals(s.getId())) {
        requireStepUpAuth();
    }
}
```

The bus is bounded and repeat reports of an id collapse onto the existing entry, so a detector that trips on every frame can't exhaust memory. An identical repeat refreshes the entry without notifying listeners again; a changed severity or detail is a new observation and is announced.

63.9. Transports the guard doesn't cover

Coverage matters more here than anywhere else in this chapter, because a gap is invisible at the call site: the code compiles, the request succeeds, and no token was ever attached.

Transport	Coverage
`ConnectionRequest`, `Rest`, `RequestBuilder`, `NetworkManager`	Token and pinning, automatically.
`WebSocket`	Token via `header()` on the opening handshake — and only on Android, desktop, Windows and Linux. iOS and the browser hand the handshake to a platform socket that exposes no way to add a header, which drops it without reporting anything. Pinning isn't reachable through the WebSocket implementations at all.
`BrowserComponent`	The initial navigation only, through `setURL(url, headers)`. Requests the loaded page makes for itself are invisible to the framework: no token, no pinning.
A native or third-party HTTP client	Nothing automatic. Use `fetchToken()` and set the header yourself.

```
// BrowserComponent: the INITIAL navigation only. Requests the loaded page makes
// itself are invisible to the framework and cannot be given a token or pinned.
Hashtable headers = AppShield.headersFor("https://api.mybank.example/statement");
browser.setURL("https://api.mybank.example/statement", headers);

// WebSocket: the handshake carries the token. Emitted on Android, desktop,
// Windows and Linux; silently dropped on iOS and in the browser, whose platform
// sockets expose no way to add a header.
WebSocket.build("wss://api.mybank.example/stream")
        .header("X-CN1-Attest", token)
```

```
        .connect();
```

63.10. Composing with your own network guard

`NetworkManager` holds exactly one guard and seals the slot after the first one is installed, so an app that needs its own interception has to delegate to the shield's rather than displace it:

```
// NetworkManager holds exactly one guard and seals the slot, so an app with its
// own has to delegate rather than replace -- calling attach() by hand instead
// would restore the token and silently drop the certificate callbacks that
// enforce pinning.
final NetworkGuard shield = AppShield.getNetworkGuard();
NetworkManager.setNetworkGuard(new NetworkGuard() {
    public void beforeRequest(ConnectionRequest r) throws IOException {
        r.addRequestHeader("X-My-Trace", newTraceId());
        shield.beforeRequest(r);
    }

    public boolean isCertificateCheckRequired(String url) {
        return shield.isCertificateCheckRequired(url);
    }

    public void checkCertificates(ConnectionRequest r, SSLCertificate[] chain)
            throws IOException {
        shield.checkCertificates(r, chain);
    }

    public String[] interestingResponseHeaders() {
        return shield.interestingResponseHeaders();
    }

    public void afterResponse(ConnectionRequest r, int code, String[] headers) {
        shield.afterResponse(r, code, headers);
    }
});
```



Don't "work around" the single slot by dropping the shield's guard and calling `attach()` by hand. That restores the token and drops the certificate callbacks with no error, so pinning stops being enforced while everything still looks like it's working — the half of the shield whose absence looks exactly like success.

63.11. Verifying the token in your backend

The token is a compact ES256 JWT with `typ` of `cn1-attest+jwt`, signed by a key published at `/.well-known/cn1-attest-jwks.json` and named by the header's `kid`. Verification is ordinary JWT verification against a JWKS URL — `NimbusJwtDecoder.withJwkSetUri` in Spring, `jose` in Node, `PyJWT[crypto]` in Python — and the console renders a copy-paste quickstart for each, including a

Cloudflare Worker for the zero-code-change case.

Beyond the standard `iss`, `aud`, `sub`, `iat`, `exp` and `jti`, the claims your middleware acts on are:

- `pkg` and `plat` — the package name and platform the token was minted for. Check them against the app you expect.
- `vrd` — the verdict: `attested`, `monitored` or `unverified`.
- `dec` — the decision under the current policy: `allow`, or `would_deny` while the app is still in monitor mode.
- `sig` — the signals the device reported.
- `pol` — the policy version the decision was made under.
- `ph` — the payload binding, when the request asked for one.

Roll it out in two stages, and keep the two sides independent. While the service is in **monitor** mode it mints a token for everyone and marks the ones it would have refused, so you can watch the would-deny rate against real traffic before anything breaks. Your middleware has its own switch: log and pass through, or reject. Move one at a time.



Reject with an opaque `403`, not a `401`. A `401` invites the client to retry with fresh credentials, which is exactly the loop you don't want, and a detailed reason is an oracle for whoever is probing you.

63.12. iOS App Attest specifics

Apple's model is **one attestation, then many assertions**: the first token carries a hardware-backed public key and its attestation, and every later token is a small assertion naming the key. The framework manages the key, the state and the retry policy in the keychain, including recovery when iOS invalidates the key — a restore to a new device, or a reinstall — and Apple's rate limits, which are strict enough that attesting per request gets an app throttled.

The one thing your code has to do is close the loop when the backend has stored the key, because a registration call that was lost leaves the device asserting against a key the server has never seen:

```
// iOS App Attest is attest-once, assert-many: the FIRST token carries the public
// key, every later one is an assertion naming it. Call this once your backend
// has stored that key, or a device whose registration call was lost keeps
// sending assertions for a key the server never saw.
DeviceIntegrity.confirmAttestation(keyId);
```

`DeviceIntegrity.resetAttestation()` throws the key away and starts again, which is the recovery path for a backend that no longer knows the key.



App Attest doesn't work in Apple's iOS simulator — it needs the Secure Enclave. Test the iOS path on a real device, and use the simulator menu below for everything else.

63.13. Testing it in the simulator

None of this is testable against a real service on a developer's desktop, so the simulator ships a simulation engine behind **Simulate** > **App Shield**. Switching on any item installs it; leaving the menu alone leaves the shield inert.

Menu item	What it makes reachable
Attestation Supported	Unchecking it simulates a platform with no attestation at all.
Attestation Result	`PASS`, or each failure in turn: `FAIL_REJECTED`, `FAIL_NO_NETWORK`, `FAIL_SERVICE_DOWN`, `FAIL_RATE_LIMITED`, `UNSUPPORTED`. This is how you exercise the transient-versus-final branch in your listener.
Device signal toggles	Rooted / jailbroken, hooking framework, emulator, debugger, repackaged, untrusted accessibility service — each raising the signal your code reacts to.
Serve Expired Token	The refresh path, without waiting out a real token's lifetime.
Force Pin Mismatch On Next Request	The most valuable item here: a fail-closed pinning branch is otherwise only reachable by mis-pinning a live host on purpose. It's a one-shot that disarms itself once spent.
Fail Pin Fetch	Confirms that a failed pin refresh doesn't fail requests.
Show Shield Status…	What the simulation is currently set to, and whether the simulated engine is registered.

The simulated token is stamped so it can't be mistaken for a real one, and the simulator's engine never contacts the service. A backend that accepts a simulated token is a backend that isn't verifying signatures.

63.14. When there is no engine

Every method in this chapter is safe to call in a build without the engine — an open-source build, an unentitled account, the simulator with the menu untouched. The degradation contract is explicit, and it's the reason you don't need to guard your call sites:

- `init()` succeeds and logs one line. `isProtected()` returns `false`.
- `fetchToken()` always completes, with `UNPROTECTED`, and never hangs.
- `attach()` does nothing. No header, no exception.
- **No request is ever blocked**, including one to a host registered as `ENFORCED`. Fail-closed enforcement requires an engine to fail; with no engine there is nothing to enforce and the app must keep working.
- `getSignals()` still returns real `DeviceIntegrity` data, so the free-tier detections described in the security chapter keep working.

Chapter 64. Biometric Authentication

Codename One ships first-class biometric authentication (Touch ID, Face ID, Android `BiometricPrompt`) under the `com.codename1.security` package. A single entry point exposes the platform prompt, a typed enum reports the modalities currently enrolled, and a typed error enum lets callers react to failures without string matching.

A sibling `SecureStorage` API stores secret strings (auth tokens, refresh tokens, encryption keys) behind biometric authentication using the iOS keychain or the Android Keystore.

64.1. Quick start: Authenticate

```
Biometrics b = Biometrics.getInstance();
if (!b.canAuthenticate()) {
    // Fall back to password
    return;
}
b.authenticate("Unlock your account").onResult((success, err) -> {
    if (err != null) {
        BiometricError code = ((BiometricException) err).getError();
        switch (code) {
            case USER_CANCELED: /* user dismissed the prompt */ break;
            case LOCKED_OUT:    /* too many bad attempts */ break;
            case NOT_ENROLLED:  /* prompt the user to enrol in Settings */ break;
            default:            /* generic failure */ break;
        }
    } else {
        // Authenticated. Continue with the gated action.
    }
});
```

Always gate the call on `canAuthenticate()` and never on `isSupported()` alone. `isSupported()` only checks for hardware, while `canAuthenticate()` also requires at least one enrolled credential and that the device isn't in a locked-out state.

64.2. Quick start: Secure storage

Entries are bound to the current set of enrolled biometrics. If the user enrols a new fingerprint or face after writing, the next `get` fails with `BiometricError.KEY_REVOKED` and the application must re-prompt for the original value and write it again.

64.3. Platform behaviour

Platform	Implementation	Notes
iOS	`LocalAuthentication.framework` (`LAContext`) + `Security.framework` keychain	Add the `ios.NSFaceIDUsageDescription` build hint when targeting Face ID hardware.
Android API 29+	`BiometricPrompt`	Face / iris / fingerprint per `PackageManager` features.
Android API 23-28	`FingerprintManager`	Fingerprint only.
JavaSE simulator	`Simulate → Biometric Simulation` submenu	Toggle hardware availability, per-modality enrolment, and the next-call outcome.
All other ports	Non-supporting fallback	`canAuthenticate()` returns `false`; `authenticate()` completes with `BiometricError.NOT_AVAILABLE`.

Application code never needs a platform `if`. The API itself returns the right thing on each port. On platforms without biometric hardware, the fallback reports `NOT_AVAILABLE` immediately.

64.4. Build hints

The Codename One Maven plugin and the build daemon both **automatically inject** the platform configuration when the app's bytecode references `com.codename1.security`:

- **iOS:** `LocalAuthentication.framework` is linked into the generated Xcode project.
- **Android:** `<uses-permission android:name="android.permission.USE_BIOMETRIC" />` (and `USE_FINGERPRINT` with `maxSdkVersion=28` for the legacy path) is injected into `AndroidManifest.xml`.

Apps that never touch the API pay nothing: no extra framework, no extra permission.

One thing the plugin won't inject: the iOS Face ID usage description. Apple rejects placeholder text, so you must supply the build hint yourself:

```
ios.NSFaceIDUsageDescription=Authenticate to securely access your account
```

The JavaSE simulator automatically sets this hint to a default the first time biometrics is invoked, so projects developed in the simulator have a working default; you should overwrite the text before shipping.

To share keychain entries with iOS App Extensions, set both the build hint and the runtime access group:

```
ios.keychainAccessGroup=TEAMID123.group.com.example.app
```

```
SecureStorage.getInstance().setKeychainAccessGroup("TEAMID123.group.com.example.app");
```

The Team ID prefix is required; omitting it produces `errSecMissingEntitlement`.

64.5. Configuring the prompt

Use `AuthenticationOptions` to control prompt copy and security policy:

```
Biometrics.getInstance().authenticate(new AuthenticationOptions()
    .setReason("Authorize transfer")               // iOS localizedReason; Android title fallback
    .setTitle("Confirm payment")                   // Android only
    .setSubtitle("Stripe charge $25.00")           // Android only
    .setNegativeButtonText("Cancel")               // Android only
    .setBiometricOnly(true)                        // reject PIN / passcode fallback
    .setSensitiveTransaction(true)                 // request class-3 ("strong") biometric (Android 30+)
);
```

Unrecognised options are ignored, so callers can set the union without platform-checking.

64.6. Typed error handling

`BiometricException.getError()` returns a `BiometricError`:

Code	Meaning
`NOT_AVAILABLE`	No biometric hardware or the platform doesn't support it
`NOT_ENROLLED`	Hardware present but the user hasn't enrolled
`LOCKED_OUT`	Too many failed attempts; temporary
`PERMANENTLY_LOCKED_OUT`	User must unlock with passcode before biometrics can be used again
`PASSCODE_NOT_SET`	Device has no PIN / pattern configured (can't use biometrics)
`USER_CANCELED`	User dismissed the prompt
`SYSTEM_CANCELED`	OS dismissed the prompt (app backgrounded, system pre-empted)
`AUTHENTICATION_FAILED`	Prompt completed but the user wasn't recognised
`KEY_REVOKED`	`SecureStorage` entry can no longer be decrypted (re-enrolment occurred)
`UNKNOWN`	Anything not covered above

64.7. Cancelling an in-flight prompt

Call `Biometrics.getInstance().stopAuthentication()` to dismiss an active prompt; the pending `AsyncResource` then completes with `BiometricError.USER_CANCELED`. The method returns `true` if a prompt was cancelled, `false` if nothing was pending.

64.8. Listing enrolled biometrics

`getAvailableBiometrics()` returns a `List<BiometricType>` to drive UI affordances (icon, prompt copy):

- `FINGERPRINT`: iOS Touch ID; Android fingerprint sensor.
- `FACE`: iOS Face ID; Android face recognition (API 29+).
- `IRIS`: a handful of Samsung devices; never on iOS.
- `STRONG`: Android class-3 authenticator tier (false-acceptance < 1/100,000); only Android API 30+.
- `WEAK`: Android class-2 authenticator tier; only Android API 30+. Can't unlock Keystore-bound keys, so weak-only devices can't use `SecureStorage`.

Never use this list to decide whether to call `authenticate()`; always use `canAuthenticate()` for that decision.

64.9. Simulator workflow

The JavaSE simulator's `Simulate → Biometric Simulation` submenu lets you exercise every code path without a device:

- **Hardware Available**: toggles `isSupported()` and `canAuthenticate()`.
- **Face ID Enrolled / Touch ID Enrolled / Iris Enrolled**: populate `getAvailableBiometrics()`.
- **Next authenticate() Outcome**: radio button that picks the outcome of the next prompt: `SUCCEED`, `FAIL`, `CANCEL`, `LOCKED_OUT`, `PERMANENTLY_LOCKED_OUT`, `NOT_ENROLLED`, `PASSCODE_NOT_SET`. The selection is sticky; change it any time.

When `authenticate()` is invoked, a small modal Swing dialog mirrors the platform prompt so the developer can see the trigger fire and click through. State persists across simulator restarts via `java.util.prefs`.

`SecureStorage` round-trips against `java.util.prefs` in the simulator, so you can test the full `set → get → KEY_REVOKED` flow end-to-end.

64.10. Security notes

The Android backing for `authenticate()` doesn't simply forward the OS success callback; it binds the prompt to a single-use AES probe cipher in the Android Keystore (with `setUserAuthenticationRequired(true)` and, on API 24+, `setInvalidatedByBiometricEnrollment(true)`). On success the cipher performs a real `doFinal` operation. A hooked or spoofed success callback fails because the Keystore refuses the operation. This guards against the bypass class documented by

CodeQL's `java/android/insecure-local-authentication` rule.

Chapter 65. Authentication and Identity

This chapter covers Codename One's modern sign-in stack: OpenID Connect, Sign in with Apple, Google, Facebook, Microsoft Entra ID, Auth0 and Firebase Authentication.

The stack is rebuilt around two new primitives:

- `com.codename1.io.oidc.OidcClient` — a full OpenID Connect / OAuth 2.0 client with PKCE, discovery, refresh and token persistence.
- `com.codename1.io.oidc.SystemBrowser` — routes the sign-in step through the platform's hardened sign-in surface (`ASWebAuthenticationSession` on iOS, Android Custom Tabs, the user's default browser on JavaSE / Web).

Every provider-specific class is a thin layer on top of these primitives, so once you learn the underlying client you understand the whole stack.



The legacy `com.codename1.io.Oauth2` class is **deprecated**. It opens an in-app `WebBrowser`, which modern identity providers (Google, Apple, Microsoft, Facebook) now refuse to render — they detect the embedded view and block the page. The new `OidcClient` works the same on every supported platform without that limitation. See Section 65.10, "Migrating from `Oauth2`" for a migration recipe.

65.1. Why the change

Identity providers tightened their rules in 2023 and 2024:

- **Google** — the legacy `GoogleSignIn` SDK has been replaced by Google Identity Services (GIS). GIS expects PKCE and discourages embedded WebViews.
- **Apple** — Sign in with Apple, mandatory for App Store apps that offer any other social login, requires a native ASAuthorizationAppleIDProvider call on iOS 13+ or a web flow with `form_post response_mode`.
- **Microsoft** — Entra ID (formerly Azure AD) requires PKCE for public clients and forwards all browser flows through Microsoft Authenticator when installed.
- **Facebook** — Facebook Login still supports OAuth 2.0 but its embedded-WebView detection now blocks unknown user agents.

The old `Oauth2` class drove the entire flow through `com.codename1.components.WebBrowser`, which is an embedded view. That model no longer works on a Codename One device build for these providers.

65.2. Quick start — any OpenID Connect provider

If your identity provider publishes a standard `.well-known/openid-configuration` document (Google, Microsoft, Apple, Auth0, Okta, Keycloak, AWS Cognito, Authentik, and most others do) you can sign users in with about ten lines of code.

```
OidcClient.discover("https://accounts.google.com").ready(new SuccessCallback<OidcClient>() {
    public void onSucess(OidcClient client) {
        client.setClientId("YOUR_CLIENT_ID")
              .setRedirectUri("com.example.app:/oauth2redirect")
              .setScopes("openid", "email", "profile");
        client.authorize().ready(new SuccessCallback<OidcTokens>() {
            public void onSucess(OidcTokens tokens) {
                // tokens.getAccessToken() -- bearer for API calls
                // tokens.getIdToken()     -- JWT with user identity claims
                // tokens.getEmail()       -- convenience accessor
            }
        }).except(new SuccessCallback<Throwable>() {
            public void onSucess(Throwable err) {
                System.out.println("Sign-in failed: " + err.getMessage());
            }
        });
    }
});
```

That call:

1. Fetches `accounts.google.com/.well-known/openid-configuration`.
2. Generates a PKCE pair (S256), a `state`, and a `nonce`.
3. Opens the OS system-browser sheet at the discovered authorization endpoint.
4. Exchanges the code for tokens on the discovered token endpoint.
5. Verifies `state` and `nonce`, decodes the ID token, and persists the tokens via the default `TokenStore`.

65.2.1. Picking a redirect URI

For mobile apps, use a **custom scheme** unique to your app, for example `com.example.app:/oauth2redirect`. Register the scheme with the OS via the build hints below so the system browser can hand the redirect back to your app.

For web / simulator runs, use a normal HTTPS URL pointing at a page you control. The Codename One fallback `BrowserWindow` closes as soon as the URL matches.

iOS

Add the URL scheme to your Info.plist via the standard build hint:

```
ios.urlScheme=<key>CFBundleURLTypes</key>\n<array>\n  <dict>\n    <key>CFBundleURLSchemes</key>\n    <array>\n
<string>com.example.app</string>\n    </array>\n  </dict>\n</array>
```

`AuthenticationServices.framework` is added to the linker automatically whenever your app references anything in `com.codename1.io.oidc` or `com.codename1.social.AppleSignIn` — you don't need to add `ios.add_libs=AuthenticationServices.framework` manually.

Android

Add an intent filter for the redirect scheme to the application manifest via the `android.xintent_filter` build hint:

```
android.xintent_filter=<intent-filter>\n  <action android:name="android.intent.action.VIEW"/>\n  <category
android:name="android.intent.category.DEFAULT"/>\n  <category android:name="android.intent.category.BROWSABLE"/>\n  <data
android:scheme="com.example.app"/>\n</intent-filter>
```

The `androidx.browser:browser:1.8.0` Gradle dependency (for Custom Tabs) is added to your app's Gradle build automatically whenever it references anything in `com.codename1.io.oidc`. Override the version with build hint `android.customTabsVersion=1.x.y` if needed.

65.2.2. Saving and restoring tokens

`OidcClient` saves the response under a per-issuer + per-client-ID key using `com.codename1.io.oidc.TokenStore.DefaultStorageTokenStore` (which serializes to `com.codename1.io.Storage`). To restore on next launch:

If the saved access token is within 60 seconds of expiring and a refresh token is available, `refreshIfExpired` performs a refresh-token grant in the background and persists the new tokens.

65.2.3. Encrypting tokens at rest

The default token store uses `com.codename1.io.Storage`, which is sandboxed but not encrypted. For biometric-gated storage, implement `TokenStore` over `com.codename1.security.SecureStorage` (see the source of `DefaultStorageTokenStore` for the pattern — swap the `Storage.writeObject` / `readObject` calls for `SecureStorage.set` / `get`).

65.3. Sign in with Apple

`com.codename1.social.AppleSignIn` consolidates the previous `cn1-applesignin` cn1lib into core. On iOS 13+ it uses native ASAuthorizationAppleIDProvider; on Android, JavaSE and the Web port it uses the public Apple OIDC issuer through `OidcClient`.

65.3.1. iOS

The iOS port implements the `com.codename1.social.AppleSignInNative` interface. No build hints are required beyond enabling the `Sign in with Apple` capability:

```
ios.signinwithapple=true
```

```
AppleSignIn.getInstance().signIn("name email", new AppleSignInCallback() {
    public void onSuccess(AppleSignInResult result) {
        // result.getIdentityToken()  -- JWT to verify on your server
        // result.getUserId()         -- stable opaque user id, use as PK
        // result.getEmail()          -- may be a real or relay address
```

```
    }
    public void onError(String err) { }
    public void onCancel()          { }
});
```



Apple only returns the user's name and email on the **first** authorization. AppleSignIn persists those values in `Preferences` so the result is consistent on subsequent sign-ins.

65.3.2. Android, JavaSE, Web

Apple requires a separate **Services ID** (a "web client") for non-iOS environments and a `client_secret` JWT generated by your server. The recipe lives at Section 65.3.3, "One-time Apple setup".

65.3.3. One-time Apple setup

1. In the Apple Developer portal, create a **Services ID** alongside your App ID. Enable "Sign in with Apple" for both.
2. Configure the Services ID's **Web Authentication Configuration** with the primary App ID and your redirect URI.
3. Generate an **AuthKey** private key in the Apple developer portal (`.p8` file). Apple doesn't allow public mobile clients to mint the `client_secret` JWT themselves; your backend must do it. See Apple's docs [https://developer.apple.com/documentation/sign_in_with_apple/generate_and_validate_tokens] for the JWT recipe.

65.4. Google Sign-In

`com.codename1.social.GoogleConnect` now offers a modern `signIn(…)` method that runs entirely through `OidcClient`:

`signIn` works on iOS, Android, JavaSE and Web with the same code, against the **Web Application** OAuth 2.0 client ID from Google Cloud Console. The redirect URI you supply must match one of the **Authorized redirect URIs** configured for that client.

The legacy `doLogin()` / `setClientSecret(…)` path is still available for source compatibility but is no longer recommended.

65.4.1. Migrating from the legacy Google integration

If you currently rely on `GoogleConnect.doLogin()` with the Google Sign-In SDK on Android or `GIDSignIn` on iOS, you can switch to the modern flow without touching your build:

- On Android, the legacy code uses `com.google.android.gms.auth.api.Auth.GoogleSignInApi` — this is the part Google deprecated. New code via `signIn(…)` doesn't require the native SDK at all.
- On iOS, the legacy code uses `GIDSignIn` (Google Sign-In SDK). The new code uses `ASWebAuthenticationSession` via `SystemBrowser`, which is what Google now recommends for non-

broker apps.

65.5. Facebook Login

`com.codename1.social.FacebookConnect` exposes both the old SDK-based `doLogin()` and a new SDK-free `signIn(…)` that uses the system browser via `OidcClient`. Use the new method for the simulator, the web port, and for apps that don't want to bundle the Facebook SDK at all:

Facebook doesn't issue OpenID-Connect ID tokens for classic OAuth flows, so `getIdToken()` returns `null`. Read user profile fields via the Graph API.

65.6. Microsoft / Entra ID

`com.codename1.social.MicrosoftConnect` wraps the same OIDC client against `https://login.microsoftonline.com/{tenant}/v2.0`. The default tenant `common` accepts personal and work / school accounts; pass a tenant GUID, `organizations`, or `consumers` to restrict.



Apps that need broker (Microsoft Authenticator) support for Conditional Access can bundle the MSAL SDK manually and bypass `MicrosoftConnect`. For the vast majority of apps the system-browser flow is sufficient and lets the simulator and web port share the same code path.

65.7. Auth0

`Auth0Connect` is the simplest of the providers because Auth0 is a clean OIDC provider; everything beyond `withDomain` is optional.

65.8. Firebase Authentication

Firebase Auth isn't an OIDC provider — it issues Google-Identity-Toolkit-style tokens via REST endpoints. `com.codename1.social.FirebaseAuth` wraps those endpoints:

For federated sign-in (Google / Apple / Microsoft as Firebase providers), first obtain an ID token via the matching `*Connect` class, then swap it for a Firebase session:

Refresh the Firebase session at app launch:

65.9. Passkeys / WebAuthn

Codename One ships a portable WebAuthn client at `com.codename1.io.webauthn.WebAuthnClient`. It wraps the OS public-key credential APIs (`ASAuthorizationPlatformPublicKeyCredentialProvider` on iOS 16+, `androidx.credentials.CredentialManager` on Android API 28+) behind a JSON-friendly Java surface so you can talk to any relying-party server with the same code on every supported platform.

65.9.1. When you actually need this class

If you sign users in via `OidcClient` against Google, Apple, Microsoft Entra ID, Auth0 or Firebase, the **identity provider already supports passkeys**. The user's web browser at the IdP shows the passkey sheet, and OIDC just hands you the resulting ID / access tokens. You don't have to call `WebAuthnClient` at all — everything in the rest of this chapter that signs users in via `OidcClient` already gets passkey-backed sign-in transparently, with no additional code on your side.

Reach for `WebAuthnClient` when:

- Your app talks to **your own** relying-party server and you want passwordless sign-in / step-up auth backed by a passkey.
- You wire up a direct Auth0 or Firebase passkey ceremony (see Section 65.9.5, "Auth0 passkeys" and Section 65.9.6, "Firebase passkeys").
- You drive a third-party SDK that emits a WebAuthn challenge.

65.9.2. Quick start — own backend

Pseudo-code (the actual HTTP calls depend on your server library):

`PublicKeyCredential#toJson()` is in the W3C `RegistrationResponseJSON` / `AuthenticationResponseJSON` format, which every WebAuthn server library accepts as-is. On the server, verify the response with one of:

- Java — webauthn4j [https://webauthn4j.github.io/webauthn4j/en/]
- Node — @simplewebauthn/server [https://simplewebauthn.dev/]
- Rust — webauthn-rs [https://github.com/kanidm/webauthn-rs]
- Python — py_webauthn [https://github.com/duo-labs/py_webauthn]

Do **not** try to verify the attestation or assertion on the device. The relying-party server holds the credential record and the signature counter; only it can detect cloned authenticators.

65.9.3. Platform requirements

Platform	Notes
iOS	iOS 16+. The `AuthenticationServices.framework` library is linked automatically when your app references `com.codename1.io.webauthn.. Add an *Associated Domains` entitlement with `webcredentials:<rp.id>` and host an `apple-app-site-association` file at `https://<rp.id>/.well-known/apple-app-site-association` describing your bundle's `webcredentials` app IDs.
Android	API 28+ via `androidx.credentials:credentials` (added to your Gradle build automatically; override the version with build hint `android.credentialsVersion=1.x.y`). Publish a Digital Asset Links file at `https://<rp.id>/.well-known/assetlinks.json` declaring your app's SHA-256 certificate fingerprint.

Platform	Notes
JavaSE / desktop, Web port	`WebAuthnClient.isSupported()` returns `false`. Calls fail with `WebAuthnException.NOT_IMPLEMENTED`. Use this to gate the UI — present a fallback (password sign-in, email magic link) when passkeys are unavailable.

65.9.4. Build hints

iOS:

```
ios.entitlements.com.apple.developer.associated-domains=<array>\n  <string>webcredentials:example.com</string>\n</array>
```

Android: nothing beyond the dependency that gets injected automatically. The asset-links file lives on your server; the build doesn't touch it.

65.9.5. Auth0 passkeys

`Auth0Connect` exposes two passkey helpers driven by Auth0's WebAuthn grant (`urn:okta:params:oauth:grant-type:webauthn`). They drive `WebAuthnClient` under the hood, so the same iOS / Android platform requirements apply.

Enable **Passkeys** in the Auth0 dashboard (Authentication → Database → your connection → Authentication Methods → Passkey), and add **WebAuthn** to the application's **Grant Types**.

65.9.6. Firebase passkeys

`FirebaseAuth` exposes `registerPasskey` (enroll a passkey for the signed-in user) and `signInWithPasskey` (sign in with an existing passkey). Both use Firebase Identity Platform's v2 passkey REST endpoints (`accounts/passkeyEnrollment:start|finalize`, `accounts/passkeySignIn:start|finalize`).

Passkey support requires the **Identity Platform** tier of Firebase Auth (the upgraded plan). The free Firebase Auth tier doesn't expose the passkey endpoints; the calls will fail with an HTTP 403. Check your project plan in the Firebase console under Build → Authentication → Settings → User account linking.

65.9.7. Mapping WebAuthn errors

`WebAuthnException.getError()` returns one of the W3C-defined error names:

Code	Meaning
`NotAllowedError`	User dismissed the OS sheet, no matching credential, or the authenticator denied the request.
`InvalidStateError`	The credential being created already exists on the authenticator.
`NotSupportedError`	Platform doesn't support the requested algorithm / transport combo.

Code	Meaning
`SecurityError`	Origin / RP-ID validation failed. Most often an Associated Domain (iOS) or asset-links (Android) misconfiguration.
`AbortError`	Caller cancelled the ceremony.
`ConstraintError`	A constraint (resident key, user verification) couldn't be satisfied.
`not_supported`	The platform has no native WebAuthn implementation at all. Caller should fall back to password sign-in.
`transport_error`	Network failure while POSTing to your relying-party server.
`invalid_options`	The options JSON received from the relying party couldn't be parsed.
`invalid_response`	The response returned by the authenticator couldn't be parsed.

65.10. Migrating from `Oauth2`

A typical legacy snippet:

```
Oauth2 oauth = new Oauth2(
    "https://provider.example.com/oauth2/authorize",
    "CLIENT_ID",
    "https://example.com/callback",
    "openid email",
    "https://provider.example.com/oauth2/token",
    "CLIENT_SECRET");
oauth.showAuthentication(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() instanceof AccessToken) {
            AccessToken t = (AccessToken) e.getSource();
            // null
        }
    }
});
```

becomes:

Or, if the provider exposes a discovery document:

`OidcTokens#toAccessToken()` returns a `com.codename1.io.AccessToken`, so callers that already deal in `AccessToken` (most subclasses of `com.codename1.social.Login`) can adopt `OidcClient` without changing their token type.

65.11. A universal sign-in demo

A complete sample is included under `samples/UniversalSignInDemo` in the repository. It renders one button per provider (Apple, Google, Microsoft, Facebook, Auth0, Firebase, generic OIDC) and dumps the resulting tokens to a `TextArea` so you can inspect them. See its `README.md` for the credentials

wiring.

Chapter 66. Deep-Link Routing

Codename One can dispatch a deep link — an iOS Universal Link, an Android App Link, a custom-scheme URL, a push-notification payload, or anything else the platform delivers as a URL — to a specific `Form` by class or by static factory method. The same route table powers in-app navigation through `Navigation.navigate("/path")`, so the application chooses declarative or imperative per call site without maintaining two parallel maps.

The application surface is two annotations plus a tiny `Navigation` class — five static methods and one value type.

66.1. Declare a route

Annotate the target `Form` class:

```
@Route("/users/:id")
public class ProfileForm extends Form {
    public ProfileForm(@RouteParam("id") String id) {
        setTitle("Profile " + id);
        // null
    }
}
```

Or annotate a static factory method:

Both forms are supported and a project can mix them. Path variables in the pattern (`:id`, `*`) are matched by name against parameters annotated with `@RouteParam`. Missing `@RouteParam` on a path-variable parameter fails the build.

66.2. Path syntax

Pattern	Matches	Bound
`/about`	`/about`	—
`/users/:id`	`/users/42`	`id = "42"`
`/files/*`	`/files/photo.png` (one segment)	`* = "photo.png"`
`/files/**`	`/files/a/b/c` (catch-all)	`* = "a/b/c"`

Query string parameters are bound the same way as path variables. The build prefers a path-variable match for a given name and falls back to the query string when the pattern doesn't include that name.

66.3. Navigate from app code

The `Navigation` class is the imperative counterpart to `@Route`. Calling `Navigation.navigate("/users/42")` from anywhere in the app resolves the URL through the same

generated route table that handles incoming deep links, builds the matching `Form`, pushes it onto the navigation stack, and shows it.

The five-method surface (`navigate`, `back`, `getCurrent`, `getStack`, `popTo`) is everything the application sees — the rest of the URL-to- form machinery is generated. Applications that prefer raw `new MyForm().show()` keep working unchanged; only URL-driven calls update the `Navigation` stack.

66.4. How it works

Codename One scans the project's compiled bytecode at build time and generates an internal dispatcher class from every `@Route` declaration. The dispatcher is bound into the per-platform application stub before `Display.init(…)`, so deep links delivered during launch see the route table immediately. The validation gate catches every problem in a single build run:

- `@Route` declared on a class that doesn't extend `Form`
- Pattern with no leading `/` or empty value
- Path variable with no matching `@RouteParam` on the constructor / factory
- Duplicate pattern declared on two different targets
- `@Route` on an abstract class or non-`public static` method

66.5. iOS Universal Links

Host an `apple-app-site-association` JSON file at `https://your.domain/.well-known/apple-app-site-association` over HTTPS without redirects. The plugin's `AasaBuilder` produces the payload:

Tell iOS which domains your app claims by setting the `ios.associatedDomains` build hint — a comma-separated list of `applinks:` entries. The iOS builder writes the Associated Domains entitlement so the resulting `.ipa` is signed for those domains automatically.

```
codename1.arg.ios.associatedDomains=applinks:example.com,applinks:www.example.com
```

66.6. Android App Links

Host an `assetlinks.json` file at `https://your.domain/.well-known/assetlinks.json`. The plugin's `AssetLinksBuilder` produces the payload:

Tell Android which URLs to intercept by setting the `android.xintent_filter` build hint with a verified intent filter for your domain. The Android builder injects the filter into the manifest.

The SHA-256 fingerprint comes from `keytool -list -v -keystore …`, or from the Play Console under **Setup** > **App integrity** when using Play App Signing.

66.7. Testing

66.7.1. iOS Simulator

```
xcrun simctl openurl booted "https://example.com/users/42"
```

66.7.2. Android Emulator

```
adb shell am start -a android.intent.action.VIEW \
    -d "https://example.com/users/42" com.example.app
```

66.7.3. Desktop Simulator

Pass `--cn1-arg=https://example.com/users/42` on the run command, or use the simulator's URL-injection helper.

66.8. Back-press guard

Independent of deep linking, individual forms can intercept back / pop attempts so they can confirm before discarding unsaved work:

The guard fires for the toolbar back button, the Android hardware back key, the iOS edge-swipe gesture, and any programmatic back navigation. `PopReason` distinguishes the trigger so a guard can be selective.

Chapter 67. JSON / XML Object Mapping

Codename One includes a build-time POJO mapping framework that turns a plain Java class — or a `PropertyBusinessObject` — into JSON or XML without any runtime reflection. The Maven plugin scans the project's compiled bytecode at build time, generates a typed `Mapper` next to each `@Mapped` class, and ships it alongside the source. The runtime entry point is two methods on `com.codename1.mapping.Mappers`.

The framework sits next to the existing `JSONParser`, `XMLParser` and `PropertyIndex.toJSON` / `fromJSON` helpers — it doesn't replace them. Use the imperative APIs when you want to read a foreign JSON shape without modelling it; use the annotation framework when you have a model class and want a one-liner round-trip.

67.1. Annotate the model

Apply `@Mapped` to the class. The processor accepts POJOs with public fields, `PropertyBusinessObject`-style classes with `Property` fields, or any mix:

```
@Mapped
@XmlRoot("user")
public class User {

    @JsonProperty("first_name")
    @XmlElement("first")
    public String firstName;

    public int age;

    // Renders as <user role="admin"/> in XML; omitted from JSON.
    @XmlAttribute
    @JsonIgnore
    public String role;

    public User() { }                                   ①
}
```

① Every `@Mapped` class must declare a public no-arg constructor — the generated mapper calls it from `fromJson` / `fromXml`.

For `PropertyBusinessObject`-style classes the same annotations work on `Property<T>` fields. The processor reads through `Property#get` / `Property#set` so subscribers (`addChangeListener`) still see the mutation:

67.1.1. Field-level annotations

Annotation	Purpose
`@JsonProperty("k")`	Rename the JSON key. Default: the field name.

Annotation	Purpose
`@JsonIgnore`	Omit the field from JSON.
`@XmlElement("name")`	Rename the XML element. Default: the field name.
`@XmlAttribute("name")`	Lift the field into an XML attribute rather than a nested child element. Only valid for scalar or `Property<scalar>` fields.
`@XmlTransient`	Omit the field from XML.

67.1.2. Supported field types

Kind	Detail
Scalars	`String`, all primitives, boxed primitives, `java.util.Date`, `byte[]` (base64 in JSON / XML).
`Property<T, ?>`	`T` must be a scalar or another `@Mapped` type.
`ListProperty<T, ?>`	Same rules as `Property`.
`java.util.List<T>`	Read from the field's generic signature.
Nested `@Mapped` ref.	Recursive serialization through the registered nested mapper.

67.2. Round-trip

`Mappers.fromJson` and `Mappers.fromXml` both accept either a `String` or a `java.io.Reader`, so streamed responses (network, file, resource) don't need to be fully buffered first.

67.3. How the plumbing works

At build time:

1. `cn1:process-annotations` (PROCESS_CLASSES phase) scans `target/classes` for every class carrying `@Mapped`.
2. The processor validates each one (concrete, public no-arg constructor, supported field types) and fails the build on the first offender — every offending class shows up in the same error report.
3. A `<SimpleName>Cn1Mapper` class is generated for every `@Mapped` type in the **same package as the source class** (e.g. `com.example.UserCn1Mapper` next to `com.example.User`). Each generated mapper exposes a public `static register()` hook.
4. A single `cn1app.MapperBootstrap` is generated whose constructor calls `UserCn1Mapper.register()`, `ItemCn1Mapper.register()`, … for every accepted `@Mapped` class.

At app start:

- On **iOS** / **Android**, the build server probes the project zip for `cn1app/MapperBootstrap.class`. When found it splices `new cn1app.MapperBootstrap();` into the per-build application stub right before `Display.init`. ParparVM rename and R8 obfuscation rewrite the call site and the

generated class together, so the direct symbol reference stays valid after the pass.

- On the **JavaSE simulator and desktop run**, `JavaSEPort#postInit` loads the bootstrap via `Class.forName("cn1app.MapperBootstrap")`. The classloader is the legitimate path here — JavaSE runs unobfuscated and the same `Class.forName` pattern is used by the @Route dispatcher.

Projects that ship no `@Mapped` classes produce no bootstrap, the build server probe falls through, and `JavaSEPort` catches the `ClassNotFoundException` — the registry stays empty and `Mappers.toJson` throws a clear "no mapper registered" error.

The runtime registry is keyed on `Class#getName()`. The map keys survive obfuscation because both the registration site and the lookup site see the same renamed name within a single execution — the keys are never persisted across builds.

67.4. Custom mappers

Sometimes a class lives in a third-party JAR the build can't annotate. Hand-write a `Mapper<T>` and register it at startup:

Hand-written mappers take precedence over generated ones for the same class.

Chapter 68. Component Binding

The component binding framework wires a model field to the matching component in a `Form` or `Container` by name. The Maven plugin generates a `<SimpleName>Cn1Binder` per `@Bindable` class at build time, so the wiring happens through direct symbol references — no reflection, no listener bookkeeping in application code.

It's a thin alternative to the imperative `UiBinding` API (`com.codename1.properties.UiBinding`). Both can be used together.

68.1. Annotate the model

```
@Bindable
public class LoginModel {

    @Bind(name = "userField", attr = BindAttr.TEXT)
    private String user;
    public String getUser()              { return user; }
    public void   setUser(String u)      { this.user = u; }                ①

    @Bind(name = "rememberMe", attr = BindAttr.SELECTED)
    public boolean remember;                                               ②

    @Bind(name = "banner", attr = BindAttr.UIID, twoWay = false)
    public String bannerStyle;

    @Bind(name = "fullName",
          attr = BindAttr.TEXT,
          getter = "computeFullName",
          setter = "applyFullName")                                        ③
    private String fullName;
    public String computeFullName()      { return fullName.toUpperCase(); }
    public void   applyFullName(String f){ this.fullName = f.trim(); }
}
```

① JavaBeans `getUser` / `setUser` are detected automatically when the field is private. The processor instruments `setUser` to fire a change notification for two-way bindings.

② Public field still works — direct field access falls back when no accessor matches the JavaBeans convention.

③ Explicit `getter` / `setter` override the JavaBeans search. Use when the convention doesn't apply (renamed accessors, transforms, fluent setters).

Every `@Bind(name=…)` is looked up against `Component#getName()`, walking the container's children recursively until a match is found. The default lookup matches GUI-builder names exactly.

68.1.1. Attribute kinds

BindAttr	Mirrors
TEXT	getText / setText on Label, Button, TextField, TextArea, SpanLabel, SpanButton. Two-way for TextArea / TextField.
UIID	getUIID / setUIID. One-way.
VISIBLE	isVisible / setVisible. One-way.
HIDDEN	isHidden / setHidden. One-way.
ENABLED	isEnabled / setEnabled. One-way.
SELECTED	isSelected / setSelected on CheckBox, RadioButton. Two-way.
ICON_NAME	Calls Resources.getGlobalResources().getImage(name) and writes the result via Label#setIcon. One-way.
NAME	setName. One-way.

68.2. Bind at runtime

Binding is the handle the binder returns:

Method	Purpose
refresh()	Push the current model values into every bound component. Use after mutating the model outside the form, or to re-sync after commit().
commit()	Pull current component values back into the model. Useful before validating / submitting when none of the bindings is two-way.
disconnect()	Remove every listener the binder installed and unregister the binding from the change-notification fan-out.

68.3. Two-way bindings and the change-notification contract

A binding declared twoWay = true flows in both directions:

1. **Component → model.** The generated binder installs a listener on the editable component (TextField, CheckBox, etc.). When the user mutates the component, the binder calls the model's setter (or assigns the public field). The setter's instrumented exit calls Binders.notifyChanged(this).
2. **Model → component.** Application code that mutates the model through a setter causes Binders.notifyChanged(this) to fire, which walks every live binding for that model and pushes the new value into the matching component.

The two paths together would loop forever if left alone — the model setter fires a change event, which refreshes the component, which fires its own change event, which calls the model setter again, … To break the loop, every framework-initiated mutation runs inside an **update region**

guarded by a thread-local depth counter. While the counter is positive, `notifyChanged` is a no-op and component listeners short-circuit.

Concretely:

- `Binders.bind(model, container)` enters an update region for the initial model → component push.
- `Binding#refresh()` enters an update region for every subsequent model → component push.
- `Binding#commit()` enters an update region for the component → model pull.
- The generated component listener enters an update region before calling the setter.

68.3.1. Setter instrumentation

For every two-way `@Bind` field whose write accessor is a method (whether you wrote `@Bind(setter="setName")` or the processor detected `setName(String)` via the JavaBeans convention), the `cn1:process-annotations` Mojo reads the original `.class` file with ASM and inserts the equivalent of:

The injection lands before every `return` point of the setter. The instrumented setter is written back to `target/classes`, replacing the original. Application code that calls `model.setName(…)` — from any thread, any code path — now triggers the binding fan-out automatically.

68.3.2. Limitation — cross-field setter chains

When a setter synchronously mutates a **second** bound field (`setFirstName` also calls `setFullName`, for example), the second field's `notifyChanged` lands inside the same update region as the first and is suppressed. The bound component for `fullName` won't catch up until something exits the region and a separate event drives the refresh. If you need the cross-field update to propagate, call `binding.refresh()` explicitly after the setter chain completes, or restructure the model so the cross-field mutation happens outside the setter's call frame — for example via a `Display.callSerially`.

68.4. POJOs versus Property objects

A `String`, `int`, or `boolean` field is read and written through the resolved accessors. A `Property<T, ?>` field is read through `get()` and written through `set()` — so existing `PropertyChangeListener` subscribers fire as expected. Both styles can sit on the same class.

68.5. Validation annotations

The same `@Bind` field can carry validation annotations that are wired into a `com.codename1.ui.validation.Validator`. The validator is built by the generated binder at the moment you call `Binders.bind(…)` and is reachable via `Binding#getValidator()` — so the same generated plumbing that pushes model values into components also installs the constraints that gate them.

Annotation	Maps to	Notes
`@Required`	`LengthConstraint(1)`	"Field must be non-empty."
`@Length(min = N)`	`LengthConstraint(N, message)`	Minimum string length.
`@Regex(pattern = …, message = …)`	`RegexConstraint(pattern, message)`	Same dialect as `com.codename1.util.regex.RE`.
`@Email`	`RegexConstraint.validEmail(message)`	The standard email regex (also accepts empty — stack with `@Required`).
`@Url`	`RegexConstraint.validURL(message)`	`http` / `https` / `ftp` / `file` schemes.
`@Numeric(decimal = …, min = …, max = …, message = …)`	`NumericConstraint(…)`	Bounds are inclusive; default range is unbounded.
`@ExistIn({…})`	`ExistInConstraint(values, caseSensitive, message)`	Whitelist of allowed string values.
`@Validate(MyConstraint.class)`	`new MyConstraint()`	Escape hatch — the class must have a public no-arg constructor and implement `Constraint`.

Multiple annotations on the same field are combined into a `GroupConstraint` (first failure wins), matching the behaviour of `Validator.addConstraint(Component, Constraint…)`.

After binding, drive validation through the returned handle:

`Binding#getValidator()` never returns `null` — when the model has no validation annotations the validator is empty and `isValid()` returns `true`. The validator owns the listener and emblem plumbing on the matching components; calling `addSubmitButtons` after `bind` is the typical pattern.

68.6. Build-time validation

In addition to the runtime constraints above, the annotation processor itself fails the build when:

- `@Bind` is applied to a field with no accessible read path — the field must be public, declare a JavaBeans `getX` / `isX` getter, or the annotation must name an explicit `getter`.
- A two-way `@Bind` field on a `TEXT` or `SELECTED` attr has no writable accessor. Set `twoWay=false` to keep the binding one-way, or add a setter / explicit `setter=`.
- `@Bind(name=…)` is empty.
- The field's static type isn't supported (raw collections, opaque references that aren't `@Bindable` themselves, …).
- `@Regex` is missing a pattern, or `@ExistIn` is missing values.

Errors are accumulated so a single build run reports every offending field at once.

68.7. How the plumbing works

`cn1:process-annotations` writes one `<SimpleName>Cn1Binder` per `@Bindable` class in the source class's package, plus a single `cn1app.BinderBootstrap` whose constructor calls `UserCn1Binder.register()`, `LoginModelCn1Binder.register()`, … for every accepted `@Bindable` class. At app start:

- On **iOS** / **Android** the build server probes the project zip for `cn1app/BinderBootstrap.class` and splices `new cn1app.BinderBootstrap();` into the per-build application stub before `Display.init`. ParparVM rename and R8 obfuscation rewrite the direct symbol reference together with the generated class so the binding still resolves after the pass.
- On **JavaSE** `JavaSEPort#postInit` loads the bootstrap via `Class.forName("cn1app.BinderBootstrap")` — the unobfuscated classloader path.

Projects with no `@Bindable` classes produce no bootstrap; the build server probe falls through and the registry stays empty.

The runtime registry is keyed on `Class#getName()`; obfuscation renames the call sites and the registered keys together within a single execution.

Chapter 69. SQLite ORM

The SQLite ORM framework turns a plain Java class into a typed data access object at build time. The Maven plugin reads `@Entity` / `@Id` / `@Column` annotations from the project's compiled bytecode, generates one `<SimpleName>Cn1Dao` per entity in the source class's package, and hands the dao to `com.codename1.orm.EntityManager` for typed retrieval. The dao issues prepared statements through `com.codename1.db.Database`, the same surface every cn1 SQLite port exposes.

The framework is a JPA-inspired, simplified alternative to the existing imperative `SQLMap`. It doesn't replace `SQLMap`; both can be used side by side.

69.1. Annotate the entity

```
@Entity(table = "users")
public class User {

    @Id(autoIncrement = true)
    public long id;

    @Column(name = "full_name", nullable = false)
    public String name;

    public int age;

    public java.util.Date createdAt;

    @DbTransient
    public String cacheKey;                                    ①

    public User() { }
}
```

① Excluded from the generated table.

`Property` fields work the same way; the dao calls `Property#get` and `Property#set` for read and write.

69.1.1. Field-level annotations

Annotation	Purpose
`@Id`	Marks the primary-key field. Exactly one per entity. `autoIncrement=true` (the default) emits `INTEGER PRIMARY KEY AUTOINCREMENT` and back-fills the field after `insert`.
`@Column(name)`	Rename the column. Default: the field name.
`@Column(type)`	Override the SQL type. Default: inferred from the Java type (`String` → `TEXT`, `int/long` → `INTEGER`, `float/double` → `REAL`, `boolean` → `INTEGER`, `java.util.Date` → `INTEGER` (epoch millis), `byte[]` → `BLOB`).

Annotation	Purpose
`@Column(nullable=false)`	Adds `NOT NULL` to the column declaration.
`@DbTransient`	Excludes the field from the table.

69.2. Use the dao

`EntityManager` is a thin façade over `Database`. The underlying connection is reachable through `em.database()` for raw SQL when the dao surface isn't enough. Transactions:

69.3. Relationships

Relationship annotations (`@OneToMany`, `@ManyToOne`, …) aren't yet supported. An entity field that references another entity or a `java.util.List<X>` of entities fails the build with a clear error — mark such fields `@DbTransient` and persist the foreign key (id) as a separate scalar column for now.

69.4. Validation

The annotation processor fails the build when:

- `@Entity` lands on an abstract class or interface.
- The class has no public no-arg constructor.
- No field carries `@Id`, or more than one does.
- A field's static type isn't supported (relationships, unsupported reference types).

Errors are accumulated so the first build run reports every offending entity at once.

69.5. How the plumbing works

`cn1:process-annotations` writes one `<SimpleName>Cn1Dao` per `@Entity` in the source class's package (`com.example.UserCn1Dao` next to `com.example.User`), plus a single `cn1app.DaoBootstrap` whose constructor calls `UserCn1Dao.register()`, `OrderCn1Dao.register()`, … for every accepted `@Entity` class. At app start:

- On **iOS** / **Android** the build server probes the project zip for `cn1app/DaoBootstrap.class` and splices `new cn1app.DaoBootstrap();` into the per-build application stub before `Display.init`. ParparVM rename and R8 obfuscation rewrite the call site and the generated dao together, so the direct symbol reference stays valid after the pass.
- On the **JavaSE simulator and desktop run** `JavaSEPort#postInit` loads the bootstrap via `Class.forName("cn1app.DaoBootstrap")`. Class-loading is the legitimate path here — JavaSE runs unobfuscated.

Projects with no `@Entity` classes produce no bootstrap; the build server probe falls through and the registry stays empty.

The runtime registry is keyed on `Class#getName()`; obfuscation renames the call sites and the registered keys together within a single execution. The keys are never persisted across builds, so the renaming has no observable effect on behavior.

Chapter 70. AI, Chat UI, and Speech

Codename One ships a portable LLM client, a streaming chat component, speech-to-text and text-to-speech APIs, built-in vision and language services, and a portable LiteRT inference API. The builders add native frameworks and dependencies when application bytecode references the corresponding entry point.

This chapter introduces every piece in turn:

- `com.codename1.ai` — provider-agnostic chat, embeddings, tool calls, and image generation.
- `com.codename1.components.ChatView` — a theme-aware chat surface that streams tokens in place.
- `com.codename1.media.SpeechRecognizer` and `com.codename1.media.TextToSpeech` — on-device speech-to-text and speech synthesis.
- `com.codename1.security.SecureStorage` non-prompting overloads — silent reads for secrets the network layer needs on every call, such as LLM API keys.
- `com.codename1.ai.vision` — OCR, barcode, face, labeling, pose, segmentation, document correction, and camera-frame pipelines.
- `com.codename1.ai.inference` — reusable LiteRT sessions for `.tflite` models.
- `com.codename1.ai.language` — language identification, translation, and smart reply.



Each async call in this chapter returns an `com.codename1.util.AsyncResource`. Use `.ready(…)` for the success path, `.except(…)` for errors, and `.cancel()` to abort an in-flight request. The streaming chat call also dispatches deltas through a `StreamingListener`; both listener callbacks and `ready` callbacks fire on the EDT.

70.1. The `com.codename1.ai` package

The `LlmClient` class is the single entry point. Static factories return a configured client for each supported provider. Every client speaks the same `ChatRequest` and `ChatResponse` value types, so the call site does not change when you swap providers:

```
// OpenAI (also drives Ollama, vLLM, llama.cpp, and other
// OpenAI-compatible endpoints).
LlmClient openai = LlmClient.openai(apiKey);

// Local Ollama on the default port (http://localhost:11434).
LlmClient ollama = LlmClient.ollama("llama3.2");

// Any OpenAI-compatible endpoint (Together, Groq, Fireworks, vLLM, null).
LlmClient together = LlmClient.localOpenAiCompatible(
        "https://api.together.xyz/v1",
        apiKey,
        "meta-llama/Llama-3.3-70B-Instruct-Turbo");

// Anthropic and Google Gemini, both via their OpenAI-compatible
```

```
// endpoints. The wire format is identical; only the base URL,
// default model, and auth differ.
LlmClient anthropic = LlmClient.anthropic(apiKey);
LlmClient gemini = LlmClient.gemini(apiKey);
```

All five factories return the same `LlmClient` API. OpenAI, Ollama, vLLM, and `llama.cpp` share the canonical wire format. Anthropic's `https://api.anthropic.com/v1/chat/completions` and Google's `https://generativelanguage.googleapis.com/v1beta/openai/chat/completions` implement the same shape, so the framework speaks to all three through one network layer. Default models pick a sensible production-grade target per provider (`gpt-4o-mini`, `claude-sonnet-4-5`, `gemini-2.0-flash`, `llama3.2`); override per request with `ChatRequest.builder().model(…)`.

70.1.1. A first chat

`ChatRequest` is a builder. The bare minimum is a model identifier and one user message:

`ChatResponse.getText()` concatenates every text part of the assistant message. `getFinishReason()` returns one of `"stop"`, `"length"`, `"tool_calls"`, `"content_filter"`, or `"error"`. `getUsage()` returns prompt, completion, and total token counts when the provider reports them; the fields return -1 when the provider omits them.

70.1.2. Streaming tokens

`chatStream(…)` opens an SSE connection and dispatches deltas through a `StreamingListener` on the EDT. The returned `AsyncResource` resolves to the final aggregated `ChatResponse` when the stream completes; calling `cancel()` on the resource closes the underlying socket:

The decoder reassembles fragmented SSE deltas before invoking the listener, so `onContentDelta` always receives a complete token chunk. Tool-call fragments are reassembled the same way and surfaced through `onToolCallDelta(index, id, name, argumentsFragment)`; `name` is non-null on the first fragment and `id` is present on the first call.

70.1.3. Tool calling

`Tool` describes a callable function with a JSON schema. Pass a `ToolHandler` to make it executable; the model can then call the tool and the handler returns the JSON result that gets fed back into the conversation:

`ToolChoice` covers the four standard modes:

- `ToolChoice.AUTO` — the model picks at will (default).
- `ToolChoice.NONE` — the model must not call any tool.
- `ToolChoice.REQUIRED` — the model must call exactly one tool.
- `ToolChoice.named("get_weather")` — force a specific tool.

70.1.4. Structured output

Constrain a response to JSON with `ResponseFormat.JSON_OBJECT`. The client adds the provider-specific

flag, and `ChatResponse.getText()` returns a string the runtime can hand to `JSONParser`:

70.1.5. Multi-modal messages

Attach an image to a user message by adding an `ImagePart`. The part accepts either inline bytes plus a MIME type or a remote HTTPS URL:

70.1.6. Embeddings

`embed(EmbeddingRequest)` returns one `Embedding` vector per input. Use the vectors for semantic search, deduplication, or downstream clustering:

70.1.7. Image generation

`ImageGenerator` mirrors the LLM client shape. The OpenAI factory drives DALL-E 3 today; the on-device factory routes through the optional `cn1-ai-stablediffusion` cn1lib (when present in the consumer project):

DALL-E 3 supports `count = 1` only. Larger batches require a different underlying model; pass it via `setModel(…)`.

70.1.8. Conversation persistence

`ConversationStore` wraps `Storage` to serialize a list of `ChatMessage` values to JSON under a named key:

`Tokenizer.estimateMessages(history)` returns a rough token count so you can trim the oldest turns before the conversation outgrows the model's context window.

70.1.9. Prompt templates

`PromptTemplate` does simple `{placeholder}` substitution. Unknown placeholders pass through unchanged so partially-filled templates are safe to log:

70.1.10. Retry policy

`RetryPolicy.exponentialBackoff()` returns a sensible default (four attempts, 500ms initial delay, 30s cap, jitter). Wrap any chat call in a retry loop yourself, or attach the policy to a higher-level wrapper. `LlmException.getRetryAfterSeconds()` returns the provider's `Retry-After` header value when present, or -1, so the policy can honor rate-limit hints.

70.1.11. Simulator redirect for offline development

The JavaSE simulator pings `localhost:11434` at startup. When Ollama is running, the simulator exports the flag `cn1.ai.ollamaDetected=true`. A second system property, `cn1.ai.simulatorRedirect`, controls whether the simulator transparently routes `LlmClient.openai(…)` calls to the local Ollama endpoint instead:

Value	Behavior
`disabled`	The default on a device. Calls go to the configured provider, even in the simulator.
`auto`	The default in the simulator. If Ollama is reachable, route OpenAI calls through it. Otherwise behave like `disabled`.
`ollama`	Force redirect to Ollama. Use this in offline-only development environments.

The redirect target is also configurable: `cn1.ai.ollamaUrl` defaults to `http://localhost:11434/v1`, and `cn1.ai.ollamaModel` defaults to `llama3.2`. Unchanged production code therefore runs offline against a local model without any conditional wiring at the call site.

70.1.12. Storing the API key



Never hard-code a provider API key in source, ship it in a bundled resource, or commit it to git. Mobile binaries are trivially reverse-engineered; any key embedded in the app is a key your users can extract. Fetch the key from a server endpoint that the user authenticates against, then cache it locally with the non-prompting `SecureStorage` overloads:

70.2. The `ChatView` component

`ChatView` is a scrollable, theme-aware chat surface that handles the list of bubbles, the streaming append, the typing indicator, and the input strip in one component. Drop it into a `Form` with a single `BorderLayout.CENTER`:

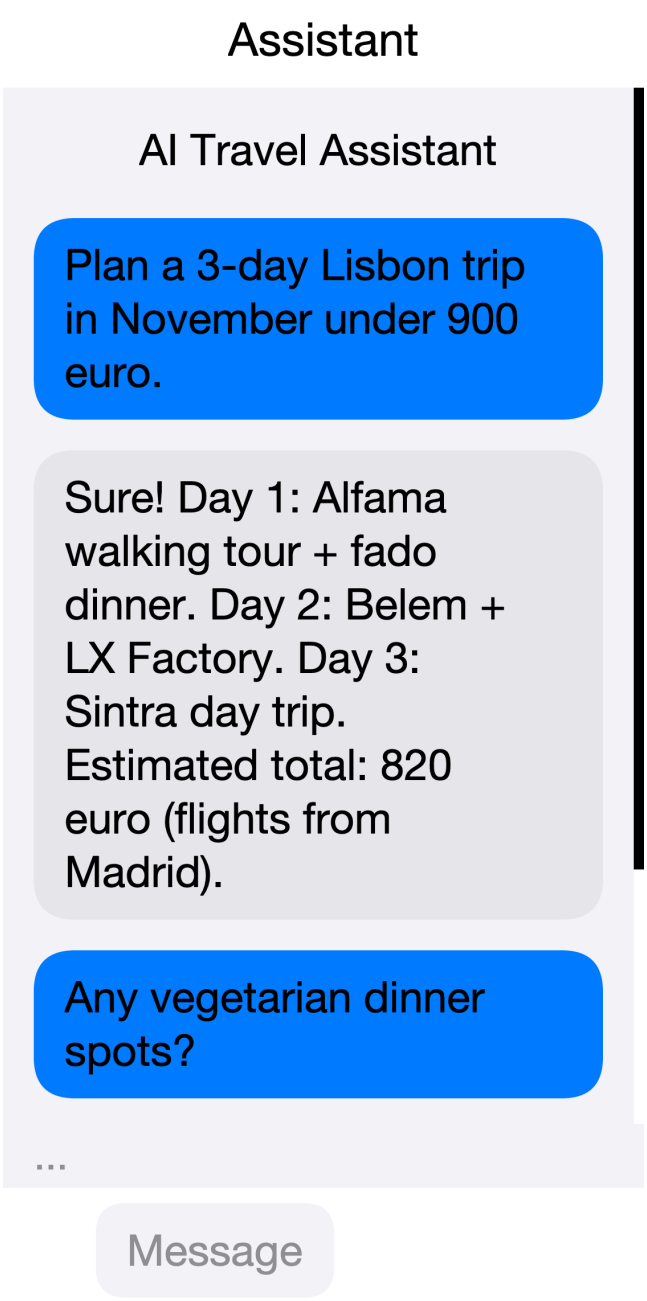


Figure 344. `ChatView` *rendered in the JavaSE simulator under the iOS Modern theme*

The component exposes thread-safe `addMessage`, `appendToLastMessage`, and `setTypingIndicatorVisible` methods, so background callbacks from `chatStream(…)` can mutate the view directly:

`ChatBubble` and `ChatInput` are public, so subclass them for custom rendering. Override `ChatView.createBubble(message)` to swap in a custom subclass when the view builds the message list.

70.2.1. Theming

`ChatView` exposes the following UIIDs out of the box:

UIID	Applies to
`ChatView`	The outer container.

UIID	Applies to
ChatViewMessages	The scrollable message column.
ChatBubbleUser	The container for a user message.
ChatBubbleAssistant	The container for an assistant message.
ChatBubbleSystem	The container for a system message.
ChatBubbleText	The inner TextArea of every bubble.
ChatTypingIndicator	The animated typing dots.
ChatInput	The input strip.
ChatInputField	The text field inside the strip.
ChatSendButton	The send button.
ChatAttachButton	The attach button.
ChatVoiceButton	The voice button.

Style them from theme.css to align with the rest of the app. Hide the voice or attach button by leaving its listener unset; the corresponding Button instance is hidden when no ActionListener is registered.

70.2.2. One-call binding to an LLM

LlmChatBinding.bind(view, client, baseRequest) wires the input bar directly to chatStream(…). Every send replays the conversation history, dispatches the response into the view, and updates the typing indicator. Use it for prototypes, or as a reference implementation when you need a custom send pipeline:

70.3. Speech recognition and TextToSpeech

The new media APIs route through Display and into the implementation hooks on CodenameOneImplementation, so the call surface is identical on every platform. The default implementation returns false from isSupported() and is a no-op for recognize/speak.

70.3.1. SpeechRecognizer

iOS uses SFSpeechRecognizer, Android uses android.speech.SpeechRecognizer, and the JavaSE simulator stays unsupported unless the optional cn1-ai-whisper cn1lib is on the classpath. Call stop() to end an active session early; partial results stop firing immediately.

70.3.2. TextToSpeech

```
if (TextToSpeech.isSupported()) {
    TtsOptions opts = new TtsOptions()
            .setLanguageTag("fr-FR")
            .setRate(1.0f);
    TextToSpeech.speak("Bonjour", opts);
```

```
}
```

iOS uses `AVSpeechSynthesizer`, Android uses `android.speech.tts.TextToSpeech`, and the JavaSE simulator falls back to `say` on macOS, `espeak` on Linux, and the platform `SAPI` bridge on Windows. `getAvailableVoices()` returns the platform-specific voice identifiers when the OS exposes them; `setVoiceId(…)` accepts any of those strings, or `null` to use the default voice for the configured language.

70.4. On-device vision

The vision API accepts JPEG, PNG, NV21, or RGBA8888 data through `VisionImage`. `VisionImage.fromCameraFrame(frame)` is normally the best way to connect a camera because it copies the frame's callback-owned buffers and preserves the frame rotation before analysis becomes asynchronous. For an encoded gallery or file image whose pixels need an EXIF display rotation, use `VisionImage.encoded(bytes, rotationDegrees)`; the one-argument overload assumes the stored pixels are already upright and doesn't inspect EXIF metadata. Each analyzer returns stable Codename One result types. Bounds and points use normalized coordinates in the range 0 through 1. `VisionOptions.minimumConfidence(…)` clamps finite and infinite values to that range and rejects NaN.

Android uses ML Kit. iOS and Mac native builds default to Apple Vision and Core Image. iOS can select ML Kit explicitly:

```
VisionOptions options = new VisionOptions()
        .backend(VisionBackends.mlKitTextRecognition())
        .minimumConfidence(0.5f);

new TextRecognizer(options)
        .process(VisionImage.encoded(jpegBytes))
        .ready(result -> Log.p(result.getText()))
        .except(error -> Log.e(error));
```

On an iOS simulator, the Apple barcode backend uses Vision barcode request revision 1. Newer simulator runtimes can otherwise complete a barcode request without reporting valid QR codes. Physical devices use the latest revision supplied by the OS. Test newer symbologies such as Micro QR on a device, or select the matching ML Kit backend.

Create and reuse an analyzer when processing a sequence. Call `close()` when the analyzer is no longer needed. Before showing a feature, call its `isSupported()` method: availability can depend on the target, linked backend, and OS version.

`VisionPipeline` attaches an analyzer to a `CameraSession` frame stream and keeps only the newest pending frame. Live OCR and pose detection therefore can't build an unbounded queue when inference is slower than the camera. While an analysis is busy, the pipeline reuses the pending buffer when newer same-sized frames replace it. Results and errors are delivered on the Codename One EDT.

`CameraFrame.getRawBytes()` can be `null` even after requesting NV21 or RGBA8888 when a port can't expose that format. JPEG bytes remain available; `VisionImage.fromCameraFrame(…)` automatically selects that fallback.

`DocumentScanner` currently performs still-image rectangle correction on Apple platforms. Google's Android document scanner is an interactive camera flow rather than an analyzer for an existing `VisionImage`, so the Android backend reports it as unsupported rather than launching a UI flow.

70.4.1. Vision feature and dependency selection

The Java package is always part of core. The builders scan the concrete analyzer classes and retain only the native adapter and dependency for each class the application actually uses:

API	Android	Apple default / optional backend
`TextRecognizer`	ML Kit text recognition	Vision / `GoogleMLKit/TextRecognition`
`BarcodeScanner`	ML Kit barcode scanning	Vision / `GoogleMLKit/BarcodeScanning`
`FaceDetector`	ML Kit face detection	Vision / `GoogleMLKit/FaceDetection`
`ImageLabeler`	ML Kit image labeling	Vision / `GoogleMLKit/ImageLabeling`
`PoseDetector`	ML Kit pose detection	Vision / `GoogleMLKit/PoseDetection`
`SelfieSegmenter`	ML Kit selfie segmentation	Vision / `GoogleMLKit/SegmentationSelfie`
`DocumentScanner`	Unsupported for still images	Core Image perspective correction

The optional iOS ML Kit pod is added only when both the analyzer and its feature-specific selector are referenced. For example, `VisionBackends.mlKitBarcodeScanning()` adds only the barcode pod. Current Google ML Kit iOS pods require iOS 15.5. Selecting one of these optional backends automatically raises the effective application and CocoaPods deployment target to 15.5 when the configured target is lower. Referencing one Android analyzer doesn't compile or package the other five adapters.

70.5. Portable LiteRT inference

`InferenceSession` loads `.tflite` data from bytes, a bundled resource, or a private file. Android uses LiteRT and can request NNAPI. iOS uses TensorFlow Lite Objective-C and may request the Core ML delegate. Both mobile implementations fall back to CPU unless `allowFallback(false)` is set. Android and iOS reject strict `Accelerator.NPU` sessions: NNAPI can mix delegated and CPU operations, and the Core ML delegate can schedule work across the Neural Engine, GPU, and CPU. Neither reports that the entire graph ran on the requested NPU. With fallback enabled, NPU selection is a best-effort accelerator preference. On iOS, strict `Accelerator.CORE_ML` sessions are also rejected because the delegate doesn't report whether unsupported model operations remained on LiteRT's CPU path. The JavaSE simulator and targets without a native LiteRT runtime, including Mac native, return `false` from `InferenceSession.isSupported()`.

```
InferenceSession.open(
        ModelSource.resource("/models/classifier.tflite"),
```

```
        new InferenceOptions().accelerator(
                InferenceOptions.Accelerator.AUTO))
    .ready(session -> {
        Tensor input = Tensor.floats("serving_default_input",
                new int[] {1, 224, 224, 3}, pixels);
        session.run(new Tensor[] {input})
                .ready(outputs -> {
                    try {
                        consume(outputs[0]);
                    } finally {
                        session.close();
                    }
                })
                .except(error -> {
                    try {
                        Log.e(error);
                    } finally {
                        session.close();
                    }
                });
    })
    .except(error -> Log.e(error));
```

Open is asynchronous because model initialization can allocate native resources. A session is reusable and supports multiple named inputs, multiple outputs, and `resizeInput(…)`. A session serializes its mutable native interpreter: while `run(…)` is pending, another run, input resize, or input/output metadata query throws `IllegalStateException`. Tensor data is copied across the port boundary. Always close the session.

Bundle small models as resources. For larger models use `ModelCache.fetch(url, cacheKey, sha256)`, which downloads over HTTPS, verifies the supplied SHA-256 digest, and reuses the app-private cached file. The initial URL must use HTTPS. Ports that expose redirect responses reject a redirect to HTTP and discard the temporary download. iOS follows redirects below the portable network layer, so iOS requires the SHA-256 argument and rejects the unpinned two-argument overload. Supply the digest on every platform for every third-party or remotely mutable model. Interrupted downloads restart through a temporary `.download` file and are never promoted to the cache until verification succeeds. Concurrent calls for the same cache key and content identity share one download but receive independent asynchronous resources. Canceling one caller suppresses only that caller's notification; it doesn't cancel the shared download or the other callers. When a cache key already has an active download, a request with a conflicting URL or digest fails instead of sharing the temporary file.

70.6. On-device language services

`LanguageIdentifier`, `Translator`, and `SmartReply` share `LanguageOptions`. Its minimum-confidence threshold clamps to 0 through 1 and rejects NaN. Translation models are downloaded and cached by ML Kit when a language pair is first used. Android and iOS builders select `language-id`, `translate`, and `smart-reply` independently, so using language identification alone doesn't package the other

two SDKs.

```
LanguageOptions options = new LanguageOptions()
        .backend(LanguageBackends.mlKitTranslation());
if (Translator.isSupported(options)) {
    Translator.translate("Where is the station?", "en", "fr",
            options)
        .ready(value -> Log.p(value))
        .except(error -> Log.e(error));
}
```

`LanguageBackends.auto()` selects ML Kit on Android and Apple's Natural Language framework for language identification on iOS. The Apple default keeps arm64 simulator builds native and adds no third-party dependency. `LanguageBackends.mlKitLanguageIdentification()` opts identification into the ML Kit iOS pod. iOS translation and Smart Reply are explicit opt-ins: use `LanguageBackends.mlKitTranslation()` or `LanguageBackends.mlKitSmartReply()` respectively. Merely referencing `Translator` or `SmartReply` doesn't add a pod or exclude the arm64 simulator. Selecting one of those ML Kit backends requires the x86_64 simulator slice and raises an iOS deployment target below 15.5.

The static language methods are convenient for one operation and close their temporary backend after either success or failure. For repeated work, call `LanguageIdentifier.open(…)`, `Translator.open(…)`, or `SmartReply.open(…)`, reuse the returned feature session, and close it when the last pending operation completes. Reusable sessions retain backend state and, on Android, cache ML Kit clients and translation clients per language pair. Mac native and the other unsupported targets return `false` from the language service `isSupported()` checks. Completion and failure callbacks arrive on the EDT.

70.7. Platform availability

The public API and the unsupported fallback are present on every Codename One target. Vision has native implementations on Android, iOS, and Mac native (Mac Catalyst); Apple Vision is the default on Apple targets and Android uses ML Kit. Language services and LiteRT inference have native implementations on Android and iOS. JavaSE, JavaScript, native Windows, native Linux, watchOS, and tvOS report unsupported for these new APIs; Mac native reports unsupported for language and LiteRT. No target sends data to a cloud inference service. Large portable runtimes or model payloads for the remaining targets remain appropriate opt-in libraries.

No build hints or cn1lib coordinates are required for the built-in mobile implementations. The local Maven plugin and BuildDaemon use the same `PlatformFeatureCatalog`, so local and cloud builds make the same dependency decisions.

70.8. Migrating from the AI cn1libs

The old `com.codename1.ai.mlkit.*` and `com.codename1.ai.tflite` packages aren't compatibility aliases. Remove those dependencies and move directly to `com.codename1.ai.vision`, `com.codename1.ai.language`, or `com.codename1.ai.inference`. Result coordinates are normalized floats

rather than packed platform-specific integer arrays. `cn1-ai-whisper` and `cn1-ai-stablediffusion` remain cn1libs because their native payloads and models are intentionally large.

Remove the retired cn1lib dependency before adding the built-in class. During migration, the builders deduplicate identical CocoaPods names and preserve an explicitly declared user version constraint. Removing the old library is still required because compatibility aliases would keep its old native-interface and packaging contracts alive.

70.8.1. Putting it all together

The pieces compose well. The following loop captures a photo, extracts text with the multi-modal `gpt-4o` model, speaks the result, and streams the same text into a `ChatView`:

The same code runs unchanged in the simulator when Ollama is detected, on Android with the multi-modal native client, and on iOS through the cloud build pipeline.

Chapter 71. Near-Field Communication (NFC)

Codename One ships an NFC API under `com.codename1.nfc` that covers the three things most apps need: read and write NDEF tags, exchange APDUs with ISO 7816 / MIFARE / FeliCa smart cards, and act as a host-emulated card for a nearby reader/terminal.

A single entry point exposes the platform NFC controller, a typed enum reports the technologies the discovered tag supports, and a typed error enum lets callers react to failures without string matching.

71.1. Quick start: Read an NDEF URI

```
Nfc nfc = Nfc.getInstance();
if (!nfc.canRead()) {
    // no NFC hardware, or it is disabled in system settings
    return;
}
nfc.readNdef(new NfcReadOptions()
        .setNdefOnly(true)
        .setAlertMessage("Hold near the poster"))
   .onResult((NdefMessage msg, Throwable err) -> {
        if (err != null) {
            return;
        }
        String url = msg.getFirstRecord().getUriPayload();
        Display.getInstance().execute(url);
   });
```

`setNdefOnly(true)` picks the fastest iOS Core NFC session (`NFCNDEFReaderSession`) and matches the most common tag layout. For mixed payloads use `readTag(…)` and inspect `tag.getTypes()`.

71.2. Quick start: Write an NDEF URI

NDEF records carry one of the well-known types (`T`, `U`, `Sp`), a MIME media type, an absolute URI, or an external `domain:type`. Use the factory methods on `NdefRecord` rather than constructing raw byte arrays:

Factory	Use for
`createUri(String)`	URI records — launches the associated app on tap
`createText(String lang, String text)`	Human-readable text
`createMime(String mimeType, byte[] payload)`	Binary payloads (vCard, JSON, …)

Factory	Use for
`createExternal(String domain, String type, byte[] payload)`	Custom vendor types
`createApplicationRecord(String pkg)`	Android Application Record (AAR)

71.3. Tag-technology APIs

Beyond NDEF, the discovered `Tag` exposes accessors for the underlying technologies:

Each accessor returns `null` when the tag doesn't advertise the corresponding technology, so always null-check before calling.

Accessor	Android	iOS	Notes
`getIsoDep()`	yes	yes	ISO 7816 — EMV, ePassport, smart cards
`getMifareClassic()`	yes	—	Apple doesn't expose MIFARE Classic on iOS
`getMifareUltralight()`	yes	yes	NTAG21x, Ultralight C
`getNfcA()`	yes	yes (limited)	Raw ISO 14443-3A
`getNfcB()`	yes	—	Raw ISO 14443-3B (Android-only)
`getNfcF()`	yes	yes	FeliCa (Suica / PASMO / ICOCA)
`getNfcV()`	yes	—	ISO 15693 (Android-only)

For FeliCa on iOS, set the system codes in `NfcReadOptions`:

The Codename One Maven plugin and build daemon automatically register `com.apple.developer.nfc.readersession.felica.systemcodes` from these values.

71.4. Quick start: Host card emulation

Host Card Emulation (HCE) lets the device pretend to be a contactless smart card. A nearby reader (Android phone, payment terminal, access control gate) sends ISO 7816 APDUs and your app responds. Subclass `HostCardEmulationService` and register the instance:

Set the AIDs as a build hint so they appear in the platform routing tables:

```
android.hceAids=F0010203040506
android.hceCategory=other
```

The Maven plugin and BuildDaemon generate `apduservice.xml` on Android, register the `CodenameOneHostApduService` in the manifest, and inject the iOS HCE entitlement so the same code

path works on both platforms.

iOS HCE (`CardSession`) requires iOS 17.4+ and, as of 2026, is EU-only. The Codename One plugin still injects the entitlement when you reference the class, but `canHostEmulate()` will report `false` on non-EU devices.

71.5. Permissions and build hints

The Codename One build pipeline automatically detects NFC usage. Apps that never touch `com.codename1.nfc` see no manifest / plist change.

Reference	Android injected	iOS injected
`com.codename1.nfc.*`	`android.permission.NFC` + `<uses-feature android:name="android.hardware.nfc">`	`NFCReaderUsageDescription`, `com.apple.developer.nfc.readersession.formats=TAG\nNDEF`, CoreNFC.framework
`com.codename1.nfc.HostCardEmulationService`	`BIND_NFC_SERVICE` permission, `android.hardware.nfc.hce` feature, `<service>` entry, generated `res/xml/apduservice.xml`	`com.apple.developer.nfc.hce` and `…hce.iso7816.select-identifiers` entitlements

Override any of the defaults with the matching build hint:

Build hint	Default	Notes
`ios.NFCReaderUsageDescription`	"Hold near an NFC tag to continue"	Localise this — Apple rejects builds that ship the default copy
`ios.entitlements.com.apple.developer.nfc.readersession.formats`	`TAG\nNDEF`	Multi-line entitlement value
`android.hceAids`	(none)	Comma-separated; required for HCE on Android
`android.hceCategory`	`other`	`payment` for EMV-conformant apps
`android.hceRequireUnlock`	`false`	Force lock screen unlock before APDU routing
`ios.hceAids`	falls back to `android.hceAids`	iOS HCE AID list

71.6. Simulator support

The JavaSE simulator implements `Nfc` via a virtual tag you can edit and tap from the `Simulate → NFC` submenu:

Menu item	Effect
Hardware Available / NFC Enabled / HCE Available	Toggle what `isSupported()` / `canRead()` / `canHostEmulate()` return
Next read outcome	Pick `DISCOVER_TAG`, `USER_CANCELED`, `TAG_LOST`, `TIMEOUT`, or `READ_ONLY` for the next `readTag()` call
Tap virtual tag	Fire the configured outcome on any pending read or registered listener
Set virtual tag URI…	Replace the virtual tag's NDEF payload with a URI record
Set virtual tag text…	Replace the virtual tag's NDEF payload with a text record
Tag is read-only	Lock the virtual tag so the next write fails with `READ_ONLY`
Send APDU to HCE service…	Hex-entry dialog that dispatches the bytes to your registered `HostCardEmulationService` and shows the response
Deactivate HCE field	Fires `onDeactivated(DEACTIVATION_LINK_LOSS)` on the registered service

All toggles persist between simulator runs (under `NfcSim.*` user preferences) so you can drive tests deterministically.

71.7. Platform behaviour

Platform	NDEF read	NDEF write	Tag tech	HCE	Notes
Android	yes	yes	full set	yes	`NfcAdapter.enableReaderMode` (API 19+); `HostApduService` (API 19+)
iOS	yes (iOS 11+)	yes (iOS 13+)	ISO 7816, FeliCa, MIFARE Ultralight	iOS 17.4+ EU-only	MIFARE Classic and NFC-B/V not exposed by Apple
JavaSE simulator	yes (virtual tag)	yes (virtual tag)	configurable	yes (synthetic APDUs)	Drive everything from the Simulate → NFC menu

Platform	NDEF read	NDEF write	Tag tech	HCE	Notes
Desktop deploy / JavaScript	no	no	no	no	Fallback base class — every method completes with `NfcError.NOT_AVAILABLE`

Always gate user-facing affordances on `canRead()` (for tag operations) or `canHostEmulate()` (for HCE). `isSupported()` only reports whether the hardware is present.

Chapter 72. Bluetooth

Codename One ships a first-class Bluetooth API under `com.codename1.bluetooth` that covers both Bluetooth Low Energy roles and classic Bluetooth: scan for and connect to BLE peripherals (the GATT client), act as a BLE peripheral (a local GATT server plus advertising), stream bytes over L2CAP connection-oriented channels, and exchange RFCOMM streams with classic serial devices.

`Bluetooth.getInstance()` is the single entry point and never returns `null` — ports without Bluetooth return a fallback whose operations fail fast with `BluetoothError.NOT_SUPPORTED`, so calling code needs no platform-specific `if`. Every callback of the API — `AsyncResource` results, scan sightings, connection events, notifications, adapter state changes — is delivered on the EDT; the only exceptions are the blocking RFCOMM/L2CAP streams, which must be consumed off the EDT.

Capability	Android	iOS / Mac Catalyst	Simulator (mock)	Simulator (native backend)	JavaScript
BLE central (scan, GATT client)	yes	yes	yes	yes	browser chooser only
BLE peripheral (GATT server + advertising)	yes	yes (not tvOS/watchOS)	yes	—	—
L2CAP channels	yes (Android 10+)	yes (iOS 11+)	yes	—	—
Classic RFCOMM	yes	—	yes	—	—
Bonding	yes	OS-managed	yes	—	—

On the JavaScript port, Web Bluetooth limits the central role to the browser's device chooser (Chrome-family browsers over HTTPS): `startScan(…)` surfaces the chooser built from the scan filters and delivers the single user-picked device as one `ScanResult`. No further sightings ever arrive, the RSSI is unknown, the MTU is fixed at 512 and peripheral mode, classic Bluetooth and L2CAP are unavailable.

Always branch through the capability queries — `isLeSupported()`, `isPeripheralModeSupported()`, `isL2capSupported()`, `isClassicSupported()` — rather than through platform detection.

72.1. Quick start: Scan and connect

```
Bluetooth bt = Bluetooth.getInstance();
if (!bt.isLeSupported()) {
    // hide the feature on ports without BLE
    return;
}
bt.requestPermissions(BluetoothPermission.SCAN, BluetoothPermission.CONNECT)
```

```
        .onResult((granted, permissionErr) -> {
    if (permissionErr != null || !granted) {
        return;
    }
    ScanSettings settings = new ScanSettings()
            .addFilter(new ScanFilter()
                    .setServiceUuid(BluetoothUuid.fromShort(0x180D))
                    .setNamePrefix("Polar"));
    BleScan[] scan = new BleScan[1];
    scan[0] = bt.getLE().startScan(settings, sighting -> {
        scan[0].stop();
        sighting.getPeripheral().connect()
                .onResult((peripheral, connectErr) -> {
            if (connectErr != null) {
                return;
            }
            peripheral.discoverServices()
                    .onResult((services, discoverErr) -> {
                // the GATT database is ready -- see the
                // following sections
            });
        });
    });
});
```

`ScanFilter` criteria on one filter are AND-combined; multiple filters added to the same `ScanSettings` are OR-combined, and a scan without filters reports every advertising device. Any number of scans may run concurrently — each `BleScan` handle only sees advertisements matching its own filters, and the platform scan stops when the last handle stops. On iOS a service-UUID filter is also what keeps scans alive in the background, so prefer filtered scans.

Peripheral addresses returned by `BluetoothDevice.getAddress()` are stable per app install — the MAC address on Android and desktop, the per-app CoreBluetooth identifier on iOS — and safe to persist for later reconnection through `BluetoothLE.getPeripheral(String)`. They're not portable across devices or platforms, so never treat them as the peripheral's real hardware address.

`getAdapterState()` reports where the local radio stands (`POWERED_ON`, `POWERED_OFF`, `UNAUTHORIZED`, …) and `addAdapterStateListener(…)` fires on the EDT whenever it changes — the natural place to pause scans and gray out UI. `requestEnable()` shows the Android system dialog asking the user to turn the adapter on; on iOS there is no programmatic enable flow, so it resolves `false` and the app should direct the user to the Settings app instead.

Connection lifecycle is equally explicit. `connect(ConnectionOptions)` accepts a timeout (iOS connect requests never time out on their own, so set one) and an `autoConnect` flag that asks the platform to re-establish the link whenever the peripheral comes back into range. `addConnectionListener(…)` reports every transition including link loss, and every failure across the API surfaces as a `BluetoothException` whose `getError()` returns a typed `BluetoothError` constant — no string matching required.

72.2. Reading and writing characteristics

`discoverServices()` caches the peripheral's GATT database; from there `getService(…)`, `getCharacteristic(…)` and the convenience accessors on `GattCharacteristic` cover the day-to-day operations:

```
GattCharacteristic measurement = peripheral.getCharacteristic(
        BluetoothUuid.fromShort(0x180D),
        BluetoothUuid.fromShort(0x2A37));
if (measurement != null && measurement.canRead()) {
    measurement.read().onResult((data, err) -> {
        if (err == null) {
            // raw bytes, delivered on the EDT
            updateHeartRateLabel(data[1] & 0xFF);
        }
    });
}

GattCharacteristic controlPoint = peripheral.getCharacteristic(
        BluetoothUuid.fromShort(0x180D),
        BluetoothUuid.fromShort(0x2A39));
if (controlPoint != null && controlPoint.canWrite()) {
    controlPoint.write(new byte[] {1})
            .onResult((acknowledged, err) -> {
        // resolves once the peripheral acknowledged the write
    });
}
```

All GATT operations return independent `AsyncResource` handles and may be issued concurrently: an internal per-peripheral queue serializes them toward the platform stack (which allows only one in-flight request per connection) and applies a safety timeout, so a lost platform callback can never wedge the queue. Operations on different peripherals run fully concurrently. `writeWithoutResponse(…)` resolves once the value is queued to the controller — there is no remote acknowledgement.

A default BLE connection carries 23 bytes per attribute operation; `requestMtu(…)` negotiates a larger value and resolves with what was granted (on iOS the OS negotiates by itself, so the request resolves immediately with the current value — check `getMtu()` before chunking data). `requestConnectionPriority(…)` trades latency against battery on Android and is a successful no-op on iOS. `createBond()` initiates pairing where the peripheral requires an encrypted link; on iOS bonding is OS-managed and triggered automatically by encrypted characteristics, so the call resolves `true` without user interaction.

72.3. Notifications

```
GattCharacteristic measurement = peripheral.getCharacteristic(
        BluetoothUuid.fromShort(0x180D),
```

```
        BluetoothUuid.fromShort(0x2A37));
GattNotificationListener listener = (characteristic, data) -> {
    // fires on the EDT for every notification while subscribed
    updateHeartRateLabel(data[1] & 0xFF);
};
measurement.subscribe(listener).onResult((armed, err) -> {
    if (err != null) {
        // the CCCD write was rejected or the link dropped
    }
});

// later, e.g. when leaving the form:
measurement.unsubscribe(listener);
```

The Client Characteristic Configuration descriptor (CCCD) is handled automatically: the first listener triggers the descriptor write that arms notifications (indications are used when the characteristic only supports those) and removing the last listener disarms it again. Values stream to the listeners on the EDT until unsubscribed or disconnected.

72.4. Peripheral mode

When `isPeripheralModeSupported()` returns `true` the device can serve data itself — open a local GATT server, add service definitions and advertise them:

```
Bluetooth bt = Bluetooth.getInstance();
if (!bt.isPeripheralModeSupported()) {
    return;
}
GattLocalCharacteristic status = new GattLocalCharacteristic(
        BluetoothUuid.fromShort(0x2A37),
        GattCharacteristic.PROPERTY_READ
                | GattCharacteristic.PROPERTY_NOTIFY,
        GattLocalCharacteristic.PERMISSION_READ)
        .setValue(new byte[] {0, 72});
GattLocalService service = new GattLocalService(
        BluetoothUuid.fromShort(0x180D))
        .addCharacteristic(status);
bt.getLE().openGattServer(new GattServerAdapter() {
    @Override
    public void centralConnected(BleCentral central) {
        // a central connected to this device
    }

    @Override
    public void characteristicWriteRequest(GattWriteRequest request) {
        // handle the written bytes, then acknowledge
        if (request.isResponseRequired()) {
            request.respond();
        }
```

```
    }
}).onResult((server, err) -> {
    if (err != null) {
        return;
    }
    server.addService(service).onResult((added, addErr) -> {
        if (addErr != null) {
            return;
        }
        bt.getLE().startAdvertising(
                new AdvertiseSettings(),
                new AdvertiseData()
                        .addServiceUuid(BluetoothUuid.fromShort(0x180D))
                        .setIncludeDeviceName(true),
                null).onResult((advertisement, advErr) -> {
            // broadcasting; keep the handle and stop() it when done
        });
        // push a new value to every subscribed central
        server.notifyValue(status, new byte[] {0, 80});
    });
});
```

Characteristics with a static value (`setValue(…)`) are served without involving the listener; characteristics without one route every read to `characteristicReadRequest(…)`, which must answer via `respond(…)` or `reject(…)` — centrals time out unanswered requests.

Two iOS platform notes: CoreBluetooth only broadcasts the local name and service UUIDs, dropping manufacturer data, service data and TX power from advertisements (Android broadcasts everything). And iOS serves descriptors from their static values only — descriptor read/write requests never reach the listener there, so always give local descriptors a static value for cross-platform behavior.

72.5. L2CAP channels

L2CAP connection-oriented channels provide a raw bidirectional byte stream — much higher throughput than chunking data through GATT characteristics. The peripheral side opens a listener and publishes its Protocol/Service Multiplexer (PSM), typically through a GATT characteristic; the central side connects to that PSM:

```
if (!Bluetooth.getInstance().isL2capSupported()) {
    return;
}
peripheral.openL2capChannel(psm, true).onResult((channel, err) -> {
    if (err != null) {
        return;
    }
    // the streams block -- always consume them off the EDT
    CN.startThread(() -> {
        try {
```

```
            OutputStream out = channel.getOutputStream();
            out.write(new byte[] {1, 2, 3});
            out.flush();
            InputStream in = channel.getInputStream();
            byte[] buffer = new byte[4096];
            int size = in.read(buffer);
            // process the response...
        } catch (IOException ioErr) {
            // transport failure
        } finally {
            try {
                channel.close();
            } catch (IOException ignored) {
            }
        }
    }, "L2CAP IO").start();
});

// the peripheral side listens and publishes its PSM to centrals
Bluetooth.getInstance().getLE().openL2capServer(true)
        .onResult((server, serverErr) -> {
    if (serverErr == null) {
        int assignedPsm = server.getPsm();
        // advertise assignedPsm through a GATT characteristic,
        // then server.accept() incoming channels
    }
});
```

On Android an L2CAP channel establishes its own link and doesn't require `connect()` first; on iOS the peripheral must be connected. The streams throw plain `java.io.IOException` on transport failure and, like all Bluetooth streams, block — never touch them on the EDT.

72.6. Classic Bluetooth

Classic (BR/EDR) Bluetooth covers inquiry discovery, bonding and RFCOMM stream connections — the Serial Port Profile (SPP) world of printers, scanners and industrial equipment. iOS doesn't expose classic Bluetooth to applications, so this role is Android, desktop and simulator only; `isClassicSupported()` is the gate:

```
Bluetooth bt = Bluetooth.getInstance();
if (!bt.isClassicSupported()) {
    // classic Bluetooth is unavailable -- iOS among others
    return;
}
BluetoothClassic classic = bt.getClassic();
classic.startDiscovery(sighting -> {
    if ("SerialPrinter".equals(sighting.getDevice().getName())) {
        classic.connect(sighting.getDevice(), BluetoothUuid.SPP, true)
                .onResult((connection, err) -> {
```

```
            if (err != null) {
                return;
            }
            // consume the blocking streams off the EDT, exactly
            // like the L2CAP example
        });
    }
});

// accepting incoming SPP clients
classic.listen("Codename One SPP", BluetoothUuid.SPP, true)
        .onResult((server, err) -> {
    if (err == null) {
        server.accept().onResult((connection, acceptErr) -> {
            // one client accepted; call accept() again for the next
        });
    }
});
```

`startDiscovery(…)` runs an inquiry scan (about 12 seconds) and returns a live `ClassicDiscovery` handle that resolves when the inquiry ends. `getBondedDevices()` lists already-paired devices without a discovery, and `connect(String, BluetoothUuid, boolean)` reconnects to a persisted address directly.

72.7. Permissions and build hints

The build pipeline scans the application's bytecode and injects only what the referenced packages need — apps that never touch `com.codename1.bluetooth` see no manifest or plist change, and a central-only app never carries advertise permissions.

Detected usage	Android injected (target SDK 31+)	iOS injected
Any `com.codename1.bluetooth` reference	`BLUETOOTH` (capped at `maxSdkVersion="30"`), `<uses-feature android:name="android.hardware.bluetooth_le">`	`NSBluetoothAlwaysUsageDescription` + `NSBluetoothPeripheralUsageDescription` defaults (only if unset), CoreBluetooth.framework, the CN1 Bluetooth natives
Scanning (`ScanSettings` / `BleScan` / classic discovery)	`BLUETOOTH_ADMIN` (capped at 30), `BLUETOOTH_SCAN` with `usesPermissionFlags="neverForLocation"`, `ACCESS_FINE_LOCATION` (capped at 30)	—
Connections (GATT, `BlePeripheral`, RFCOMM, L2CAP)	`BLUETOOTH_CONNECT`	—

Detected usage	Android injected (target SDK 31+)	iOS injected
Peripheral mode (`com.codename1.bluetooth.le.server`)	`BLUETOOTH_ADMIN` (capped at 30), `BLUETOOTH_ADVERTISE`	—
Classic (`com.codename1.bluetooth.classic`)	`<uses-feature android:name="android.hardware.bluetooth">`	not supported

Builds targeting an SDK below 31 receive the legacy permissions uncapped and none of the Android 12 ones. Permissions you declare yourself (via `android.xpermissions`) are never duplicated.

Override the defaults with the matching build hints:

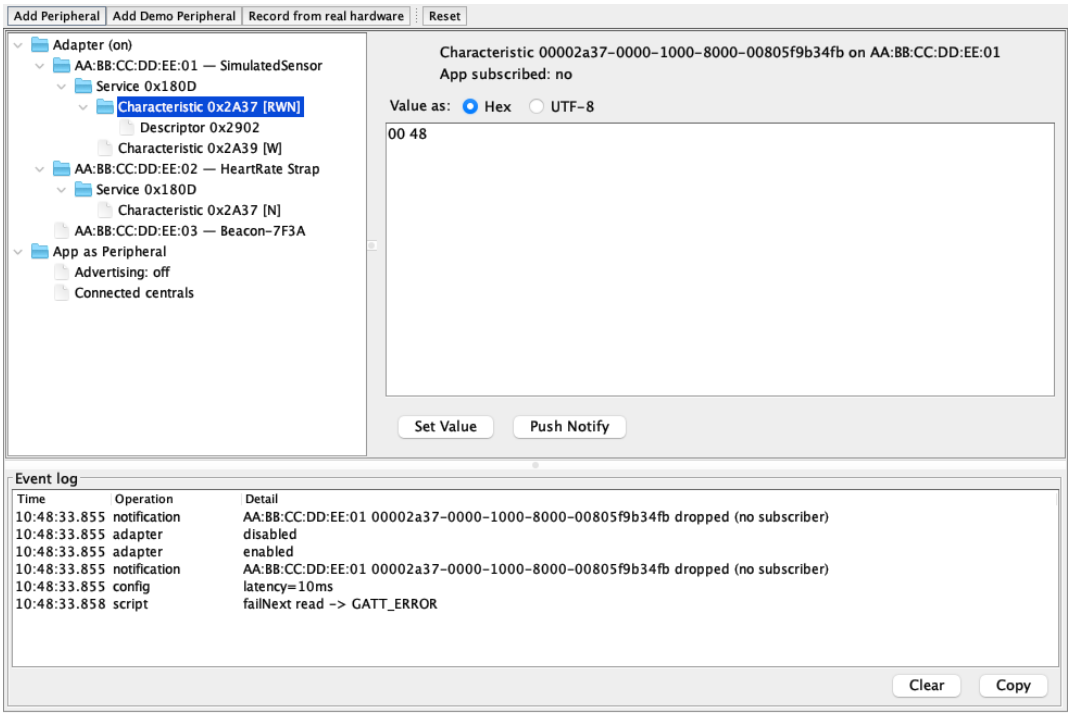
Build hint	Default	Notes
`android.bluetooth.neverForLocation`	`true`	Declares that scan results never derive the user's location, which caps `ACCESS_FINE_LOCATION` at API 30 and flags `BLUETOOTH_SCAN` accordingly
`android.bluetooth.required`	`false`	Marks the BLE hardware feature required so stores hide the app on devices without it
`ios.bluetooth.background`	(none)	`central`, `peripheral` or `central,peripheral` — merged into `UIBackgroundModes` as `bluetooth-central` / `bluetooth-peripheral`
`ios.NSBluetoothAlwaysUsageDescription`	"Communicates with nearby Bluetooth accessories."	Localise this — Apple rejects builds that ship generic privacy copy

Beacon and indoor-positioning apps are the one case where the `neverForLocation` default is wrong: an app that derives location from BLE sightings must set `android.bluetooth.neverForLocation=false`, otherwise Android 12+ filters beacon advertisements out of its scan results. With the hint set to `false` the location permission stays uncapped and the app keeps prompting for it on Android 12+.

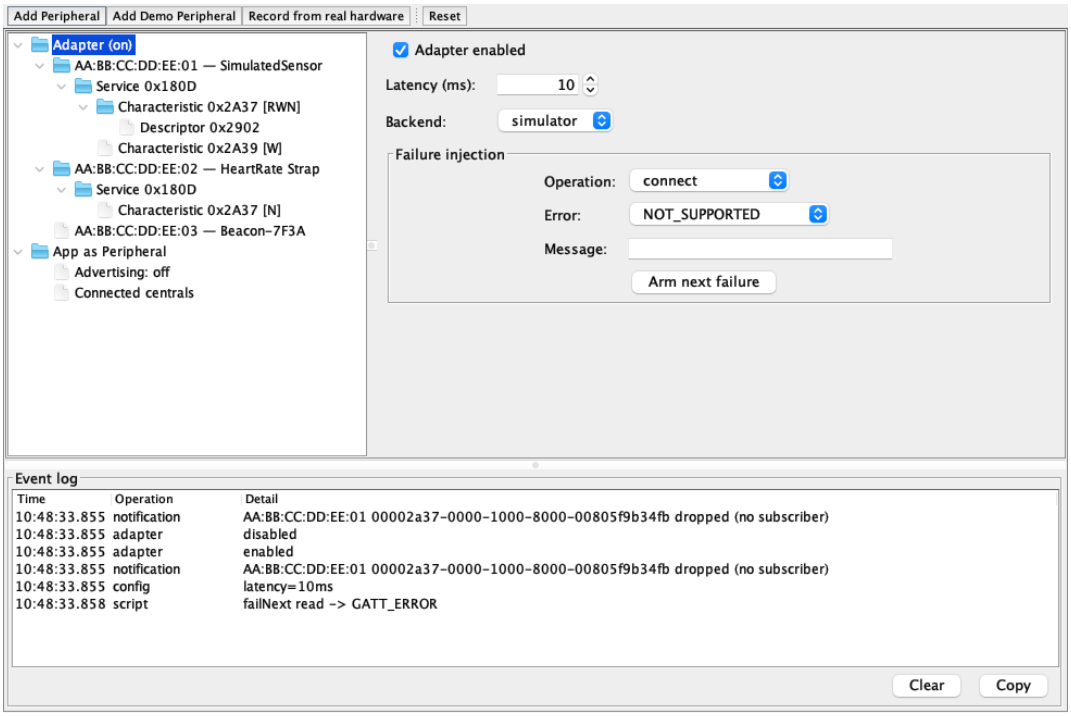
72.8. Simulator support

The JavaSE simulator implements the whole API against a scriptable virtual Bluetooth stack. `Simulate → Bluetooth Simulation` opens a window with a tree of the simulated adapter, the virtual peripherals and the app's own peripheral role; selecting a node opens its detail editor (hex value editors on characteristics), a toolbar stages demo devices, failure injection arms the next operation of a chosen kind (`connect`, `read`, `write`, `discover`, `subscribe`, `scan` and the rest) to fail with a chosen

`BluetoothError`, a latency spinner delays every asynchronous completion and a live event log shows what the stack does. All settings persist between simulator runs.



The window's backend selector switches the simulator between the virtual stack and a native backend that drives the host machine's real Bluetooth radio through a bundled btleplug library (`libcn1ble`) loaded in-process (CoreBluetooth on macOS, BlueZ on Linux, WinRT on Windows). The native backend is central-only — real scanning, connections and GATT operations against physical devices, without leaving the simulator. The initial backend comes from the `cn1.bluetooth.backend` system property (default `simulator`); point `cn1.bluetooth.libraryPath` at a locally built `libcn1ble` to override the bundled one.



The `Simulate → Bluetooth` menu exposes the same operations as one-click hooks; every item is also callable from cross-platform code via `CN.execute("bluetooth:itemN")`:

Hook	Menu label	Effect
`bluetooth:item1`	Toggle Adapter	Flips the simulated adapter between `POWERED_ON` and `POWERED_OFF`
`bluetooth:item2`	Add Demo Peripheral	Registers the canonical demo device: `AA:BB:CC:DD:EE:01` "SimulatedSensor" with service `0x180D`, a read/write/notify characteristic `0x2A37` (carrying a CCCD) and a writable `0x2A39`
`bluetooth:item3`	Push Demo Notification	Pushes a rolling one-byte notification from the demo peripheral's `0x2A37` characteristic
`bluetooth:item4`	Disconnect All	Drops every live connection from the remote side
`bluetooth:item5`	Clear Peripherals	Removes every registered virtual peripheral
`bluetooth:item6`	Use Simulator Backend	Switches back to the virtual stack
`bluetooth:item7`	Use Native Backend	Switches to the host machine's real radio
`bluetooth:item8`	(API only)	Arms the next GATT read to fail with `GATT_ERROR`

Everything the window does goes through the scriptable `BluetoothSimulator` API in `com.codename1.impl.javase.bluetooth`, so tests and sample apps can stage the same devices programmatically. This code runs against the JavaSE port only — it isn't part of the cross-platform API and won't compile in a portable code base, so keep it in your test or debug scaffolding:

```
BluetoothSimulator.addPeripheral(
        new VirtualPeripheral("AA:BB:CC:DD:EE:10")
                .setName("Thermometer")
                .addAdvertisedServiceUuid(BluetoothUuid.fromShort(0x1809))
                .withService(new VirtualService(BluetoothUuid.fromShort(0x1809))
                        .withCharacteristic(new VirtualCharacteristic(
                                BluetoothUuid.fromShort(0x2A1C),
                                GattCharacteristic.PROPERTY_READ
                                        | GattCharacteristic.PROPERTY_INDICATE,
                                new byte[] {0, 0, 0, 98})
                                .withDescriptor(new VirtualDescriptor(
                                        BluetoothUuid.CCCD, new byte[] {0, 0})))));
BluetoothSimulator.setLatencyMillis(50);
BluetoothSimulator.failNext("connect",
```

```
        BluetoothError.CONNECTION_FAILED, "Injected for testing");
```

72.8.1. Record and replay

Fixtures bridge the two backends: the window's "Record from real hardware" button (or `scripts/bluetooth/capture-fixture.sh`) scans the host machine's real radio through the native backend for a chosen duration, captures device identities, RSSI timelines, advertisement payloads and GATT databases, scrambles the trace deterministically so committed fixtures never carry real device identities, and writes it as JSON. `BluetoothSimulator.loadFixture(InputStream)` replays the trace into the virtual stack — devices appear at their recorded first-sighting times and their RSSI timelines replay on the stack's scheduler, so a scan against messy real-world radio traffic becomes a reproducible test input.

72.9. Testing

Cross-platform test code drives the simulation through the hooks: `CN.execute("bluetooth:item2")` stages the demo peripheral, the test scans for `SimulatedSensor`, connects and subscribes to `0x2A37`, and `CN.execute("bluetooth:item3")` pushes a notification to assert on — no simulator-only imports needed, so the same test class compiles for device targets and runs unchanged in the simulator's test runner. Failure paths test the same way: `CN.execute("bluetooth:item8")` arms the next GATT read to fail with `GATT_ERROR`, letting a portable test verify the app's error handling.

For fully deterministic unit tests the JavaSE port also allows constructing a private `SimulatedBluetoothStack` on a `ManualScheduler`, where time only advances when the test pumps it — no sleeps, no flakiness from real timers; see the Bluetooth tests under the `javase` Maven module for the pattern.

72.10. Platform behaviour

Platform	Address semantics	MTU	Bonding	Notes
Android	Real MAC address	`requestMtu(…)` negotiates, default 23	`createBond()` prompts the user	Background scanning is throttled by the OS; use filtered scans and expect slower sightings
iOS / Mac Catalyst	Per-app CoreBluetooth identifier UUID	Negotiated by the OS; `requestMtu(…)` resolves with the current value	OS-managed, triggered by encrypted characteristics; `getBondedPeripherals()` is empty	Background operation needs `ios.bluetooth.background` plus service-UUID scan filters; advertisements carry only name and service UUIDs

Platform	Address semantics	MTU	Bonding	Notes
Simulator (mock)	Synthetic (`AA:BB:CC:DD:EE:01`)	Simulated negotiation	Simulated	Timestamps come from a monotonic counter — treat them as ordered, not as wall-clock times
Simulator (native backend)	Host-OS device identifier	Real negotiation	Not supported	Central role only; requires the bundled `libcn1ble` library and OS Bluetooth permission for the terminal/IDE
JavaScript	Per-origin Web Bluetooth identifier	Fixed at 512	Browser-managed	One user-picked device per chooser; `startScan` requires a user gesture (call it from a button handler); RSSI unavailable

Persist addresses only for same-install reconnection, branch on the capability queries rather than on `Display.getPlatformName()`, and keep every byte-stream consumer off the EDT — those three rules keep one code base working across every column of the capability matrix.

Chapter 73. Health

Codename One ships a cross-platform health API under `com.codename1.health` that reads and writes the platform health store — HealthKit on iOS and watchOS, Health Connect on Android — summarizes data over time, watches it for changes, records workouts, and streams live measurements from standard Bluetooth health sensors.

`Health.getInstance()` is the single entry point and never returns `null`: ports without health support return a fallback whose operations fail fast with `HealthError.NOT_SUPPORTED`, so calling code needs no platform-specific `if`.

Every callback — `AsyncResource` results, sensor samples, workout events, change batches — is delivered on the EDT, on every platform. That includes the local-backed ports: the simulator, desktop and JavaScript stores hand results back on the EDT rather than on whichever thread called, so a read started from a worker thread can update the UI from its callback directly, with no `callSerially` of your own. A background change delivery may arrive with no visible UI, after the OS relaunched your app, but it's still on the EDT.

Capability	**iOS 26+**	**iOS < 26**	**watchOS**	**Android**	**Simulator**	**Desktop / JavaScript**
Read samples and aggregates	yes	yes	yes	yes	yes (scripted)	local only
Write samples	yes	yes	yes	yes	yes	local only
Read-authorization introspection	no	no	no	yes	yes	n/a
Background push delivery	**no**	**no**	**no**	**no**	**no**	no
Change delivery while running	yes	yes	yes	yes	**no**	**no**
Live workout session	**recorded**	recorded	**recorded**	recorded	recorded	recorded
Automatic sensor collection in a workout	**no**	no	**no**	no	no	no

Capability	iOS 26+	iOS < 26	watchOS	Android	Simulator	Desktop / JavaScript
Bluetooth health sensors	yes	yes	limited	yes	yes	yes

Subscriptions are the one place the local-backed ports aren't a substitute for a device. `subscribe()` registers, persists its cursor and restores across launches everywhere, but nothing on the local store records a mutation as a change, so `drainChanges()` there resolves with zero and no listener ever fires. Subscription code can be written and its registration exercised on the desktop; whether it actually delivers has to be tested on a phone.

Branch through the capability queries — `Health.getAvailability()`, `HealthStore.isTypeSupported(…)`, `WorkoutManager.isLiveSessionSupported()`, `HealthSubscription.isPushDelivery()` — rather than through platform detection.

73.1. Quick start

```
Health health = Health.getInstance();
HealthAvailability availability = health.getAvailability();
if (availability == HealthAvailability.PROVIDER_NOT_INSTALLED
        || availability == HealthAvailability.PROVIDER_UPDATE_REQUIRED) {
    health.openProviderSetup();
    return;
}
if (availability == HealthAvailability.NOT_SUPPORTED) {
    return;
}
HealthStore store = health.getStore();
store.requestAuthorization(
        HealthAccess.read(HealthDataType.STEPS),
        HealthAccess.read(HealthDataType.HEART_RATE))
     .onResult((asked, err) -> {
         if (err != null) {
             Log.e(err);
             return;
         }
         // asked == true means the user has now been ASKED, not that
         // anything was granted. On iOS the sheet completes identically
         // whether they enabled every switch or none.
     });
```

Check `getAvailability()` before anything else. On Android it distinguishes a working provider from one that's missing or out of date, and both of those are recoverable by the user through `openProviderSetup()`. On the desktop and JavaScript ports it returns `LOCAL_ONLY`, meaning reads and writes work and are durable but the data is only ever your own app's.

73.2. Permissions, and the one thing that will surprise you

You can't ask whether you are allowed to read.

HealthKit doesn't disclose read authorization. A read the user denied returns an empty result, indistinguishable from having no data at all, because an app that could tell the two apart could infer that a user is hiding a pregnancy or a prescription. That's a privacy guarantee, not a gap in this API.

Three consequences follow, and all three are load-bearing:

1. `requestAuthorization(…)` resolving `true` means the user **has been asked**, not that anything was granted. On iOS the sheet completes identically whether they enabled every switch or none of them.
2. `getReadAuthorizationStatus(…)` returns `UNKNOWN` on iOS in every case. Android answers, because its read permissions are ordinary runtime grants; the Android port doesn't pretend otherwise, and neither should your UI.
3. There is no `hasReadPermission()` anywhere in this API, because on iOS it would be a lie.

```
HealthStore store = Health.getInstance().getStore();

// Write authorization is answerable on both platforms.
HealthAuthorizationStatus write =
        store.getWriteAuthorizationStatus(HealthDataType.BODY_MASS);

// Read authorization is NOT. On iOS this is always UNKNOWN, by design.
HealthAuthorizationStatus read =
        store.getReadAuthorizationStatus(HealthDataType.STEPS);
if (read == HealthAuthorizationStatus.UNKNOWN) {
    // The only honest probe: ask whether any data actually came back.
    store.hasAnyData(HealthDataType.STEPS, HealthTimeRange.lastDays(7))
         .onResult((hasData, err) -> {
             if (err == null && !hasData.booleanValue()) {
                 // Denied OR genuinely empty -- indistinguishable.
                 // Say "no data available", never "you denied access".
                 Health.getInstance().openHealthSettings();
             }
         });
}
```

`hasAnyData(…)` is the only honest probe, and its name says exactly what it measures. Note also that on iOS your own writes stay readable to you even when read access was denied, so a `true` here doesn't prove you can see other apps' data.

Never tell a user "you denied access" on the strength of an empty result. Say "no data available," and offer `openHealthSettings()`. Getting this wrong produces an app that accuses users of

something they didn't do.

Request the narrowest set of types you actually need, at the moment you need them. Both stores present every requested type on one sheet, and a long list at first launch is the most common reason people decline.

73.3. Reading samples

```
SampleQuery query = new SampleQuery()
        .addType(HealthDataType.HEART_RATE)
        .setTimeRange(HealthTimeRange.lastHours(24))
        .setSortDescending(true)
        .setLimit(500);

Health.getInstance().getStore().readSamples(query)
        .onResult((samples, err) -> {
            if (err != null) {
                Log.e(err);
                return;
            }
            for (HealthSample s : samples) {
                QuantitySample hr = (QuantitySample) s;
                // Reading a value forces you to name the unit, which is
                // what removes the whole wrong-unit class of bug.
                double bpm = hr.getValue(HealthUnit.COUNT_PER_MINUTE);
                Log.p(hr.getStartMillis() + ": " + bpm + " bpm");
            }
        });
```

Values always come back in the type's canonical unit unless the query asks for another, so the two ports return identical objects. Reading a number out of a `HealthQuantity` requires naming the unit at the call site — there is no zero-argument `getValue()` — which removes the entire class of bug where pounds are read as kilograms and every downstream chart is wrong by a factor of 2.2.

On Android a limit applies per data type when a query names several: Health Connect pages per record type, so a limit of ten over two types can return twenty. Query one type per call when the cap has to be exact, which is what iOS does anyway — `HKSampleQuery` reads one type per query.

Always set a limit for high-frequency types. A year of continuous heart rate is on the order of half a million samples; the default cap of 10,000 exists so a naive query can't exhaust the heap, and `readSamplePage(…)` is there for walking more than that.

Health Connect stores a heart-rate series as one record containing many samples, while HealthKit returns many independent samples. `SampleQuery.setFlattenSeries(boolean)` defaults to `true` so both platforms hand back plain `QuantitySample` objects. Turn it off to keep a Health Connect record whole, which is what you want when you intend to delete it: you get one `SeriesSample` carrying the record identifier and every measurement in it. HealthKit has no series records at all, so iOS answers with plain samples whichever way the flag is set. Every flattened measurement still carries

its record identifier, so record-level deletion works either way — and `isDeletable(…)` is a separate question from `isWritable(…)`, so the series-shaped types Health Connect won't let you write can still be deleted.

Writing a `SeriesSample` to a phone stores its measurements individually — neither platform accepts a series through this API — so the write result carries one identifier per measurement rather than one for the record. A series is unit-checked and converted exactly like a scalar write, and one longer than the platform's batch limit is split across several calls. Sleep and workout samples can't be written to HealthKit or Health Connect at all; writing one is refused with `TYPE_NOT_SUPPORTED` rather than reported as a success that stored nothing.

Sleep isn't readable on either phone in this release. The iOS type map carries quantity types only, so `HealthDataType.SLEEP` is refused before a query runs, and `SampleQuery.setSleepSessionGapMillis(long)` — which configures the session reassembly that port will eventually need — changes nothing today. Android refuses it too: sleep is absent from the readable type set and the bridge has no record class for it, so a query fails with `TYPE_NOT_SUPPORTED` rather than returning nothing. Sleep works fully against the local and simulator stores, which is where to develop against it for now.

73.4. Aggregates and time buckets

```
AggregateQuery query = new AggregateQuery()
        .addType(HealthDataType.STEPS)
        .addMetric(AggregateMetric.TOTAL)
        .setTimeRange(HealthTimeRange.calendarDays(7, ZoneId.systemDefault()))
        // Calendar days, not a fixed 24 hours: a day is 23 or 25 hours
        // across a daylight-saving transition.
        .setBucket(HealthInterval.calendarDays(1, ZoneId.systemDefault()));

Health.getInstance().getStore().aggregate(query)
        .onResult((buckets, err) -> {
            if (err != null) {
                Log.e(err);
                return;
            }
            for (AggregateResult bucket : buckets) {
                HealthQuantity total = bucket.get(HealthDataType.STEPS,
                        AggregateMetric.TOTAL);
                if (total == null) {
                    // No data for that day. NOT the same as zero steps --
                    // render a gap rather than a bar at zero.
                    continue;
                }
                Log.p(bucket.getBucketStartMillis() + ": "
                        + (long) total.getValue(HealthUnit.COUNT));
            }
        });
```

Two rules here are easy to get wrong and hard to notice.

A bucket with no data returns `null`, never zero. A day on which no data was recorded and a day on which the user genuinely took no steps are different facts. Substituting zero turns an absence of data into a claim that nothing happened and draws a flat line through every day the phone stayed in a drawer.

Use calendar buckets when your UI labels them with dates. `HealthInterval.calendarDays(1, tz)` follows the calendar; a fixed `86_400_000` doesn't, and drifts against the dates the user sees twice a year at each daylight-saving transition. Calendar intervals require an explicit `TimeZone` — nothing in this API reads the JVM default, because a server-side default of UTC would file a user's evening walk under the wrong day.



Overlapping sources are counted twice, on every platform including iOS. When a phone and a watch both record steps for one walk, a total over them counts the walk twice. HealthKit's statistics engine does de-duplicate overlapping sources, but no port uses it in this release — every metric is computed by shared code from raw samples, so the bucket arithmetic has one implementation rather than one per platform that can drift, and iOS double-counts exactly as Android does. This API leaves that visible rather than papering over it with a heuristic, because guessing which of two overlapping sources is authoritative is exactly the kind of quiet wrongness health data can't afford. Use `addSource(…)` to pin a query to the source you trust, and tell the user which device a figure came from.

73.5. Writing samples

```
QuantitySample weight = QuantitySample.create(
        HealthDataType.BODY_MASS,
        new HealthQuantity(178.4, HealthUnit.POUND),
        System.currentTimeMillis());
weight.setRecordingMethod(RecordingMethod.MANUAL_ENTRY);

Health.getInstance().getStore().write(weight)
        .onResult((result, err) -> {
            if (err instanceof HealthException
                    && ((HealthException) err).getError()
                            == HealthError.UNAUTHORIZED) {
                Health.getInstance().openHealthSettings();
                return;
            }
            if (result != null && result.hasRejections()) {
                // A batch can partially succeed; surface what was dropped
                // rather than discarding it silently.
                for (String reason : result.getRejections()) {
                    Log.p("rejected: " + reason);
                }
            }
        });
```

Writes are validated before the platform is touched: a type this app can't write, an instantaneous sample of a cumulative type such as `STEPS`, or a quantity whose unit measures the wrong dimension all fail immediately with a message naming the offending sample, rather than as an opaque platform error later. Large batches are chunked automatically to the platform's limit.

Set `RecordingMethod` on what you write. Other apps use it to decide how much to trust a value, and a manually typed weight is a different kind of evidence from one a scale reported.

`HealthSample.getId()` is assigned by the platform, is scoped to this install, and won't survive a reinstall, so don't use it as a primary key on your server. Keep your own identifier in your own storage and correlate on that. `getMetadata()` looks like the natural home for it and isn't: metadata round-trips through the local and simulator stores but isn't written to HealthKit or Health Connect in this release, so on a phone the identifier is gone the moment you read the sample back.

73.6. Background delivery

```
SubscriptionRequest request = new SubscriptionRequest("steps-v1")
        .addType(HealthDataType.STEPS)
        .setIncludeDeletions(true);

HealthSubscription sub = Health.getInstance().getStore()
        .subscribe(request, StepWatcher.class);

if (!sub.isPushDelivery()) {
    // Android: Health Connect never wakes the app. Drain from your
    // background-fetch handler instead of assuming push.
    Display.getInstance().setPreferredBackgroundFetchInterval(900);
}
```

The subscription id keys the persisted cursor that records how far you have read. Reuse it across launches and app updates and you resume exactly where you left off; change it and the framework treats it as new and resynchronizes from scratch. Hard-code it, and version it explicitly (`"steps-v1"`) if you ever want a clean start.

No platform wakes your app for health data today. `isPushDelivery()` answers `false` everywhere, and that's the honest answer rather than a placeholder: nothing arrives while your app is closed, so your app has to ask.

On Android this is permanent. Health Connect has no push mechanism at all and Google's own guidance is to poll. On iOS, HealthKit does offer `HKObserverQuery` with background relaunch, but this release doesn't register one, so iOS behaves the same way as Android for now. `isPushDelivery()` is the query to branch on; it will start answering `true` on iOS when observers land, and code that already polls will simply poll less often.

Draining is what delivers changes, and your app has to ask for it. Nothing here hooks the application lifecycle: call `drainChanges()` when you come to the foreground, and from your background-fetch handler. Omit it and a subscription delivers nothing at all, however much data arrives. This is the one place where forgetting a call looks exactly like having no new data.

The two backends drain differently, and the difference is visible to you. Android uses Health Connect change tokens, so a drain reports deletions as well as additions and never re-reports a sample you have already seen. iOS re-reads each subscribed type over the window since the last drain: additions arrive, deletions don't, and a sample edited in place looks like a new one. Both advance their cursor only after your listener returns, so a crash re-reads rather than skips.

On iOS a subscription can miss backdated data, and there is no way around it in this release. The cursor is a timestamp, so it finds samples by when they were **measured**, not by when they arrived in the store. A watch that syncs an hour late, or a reading the user types in for yesterday, lands entirely behind the cursor and no later drain will ever see it. HealthKit's own `HKAnchoredObjectQuery` is what solves this — it's ordered by insertion and would fix deletions and edit-detection at the same time — and this release doesn't use it. If your app has to be complete rather than merely current on iOS, do a full `readSamples` over the range you care about on a schedule of your own, and treat the subscription as a prompt to refresh rather than as the source of truth. Android is unaffected: change tokens are insertion-ordered already.

A busy window on iOS can hold more samples than one HealthKit query returns, and HealthKit offers no continuation token. When that happens the cursor advances only as far as the samples actually delivered and the batch reports `hasMore()`, so the next drain picks up the remainder rather than skipping it. Draining more often keeps each window small; a subscription left undrained for weeks will take several drains to catch up.

Handle `isResyncRequired()`. A Health Connect change token expires after 30 days and an iOS anchor can be rejected after a restore from backup; when that happens the batch carries no data and you must do a full time-range read to catch up.

A `HealthBackgroundListener` must be a public top-level class with a public no-argument constructor. Unlike some older callback APIs in this framework there is **no** need to do anything to keep it from being stripped: the build server scans for implementations and generates a factory that constructs each one with a direct `new`, so shrinking and obfuscation both follow the reference correctly.

73.7. Workouts

```
WorkoutManager workouts = Health.getInstance().getWorkouts();
workouts.startSession(new WorkoutConfiguration()
        .setActivityType(WorkoutActivityType.RUNNING)
        .setLocationType(WorkoutLocationType.OUTDOOR))
    .onResult((session, err) -> {
        if (err != null) {
            Log.e(err);
            return;
        }
        if (!session.isLive()) {
            // Android phones and iOS before 26: the clock and the saved
            // record are real, but nothing is collected unless you feed
            // it -- from a sensor, from location, or by hand.
            Log.p("Recording; connect a strap for heart rate");
        }
        session.start();
```

```
    });
```

Two capabilities that are easy to conflate and that this API keeps separate:

- `isLiveSessionSupported()` — the OS runs a real session and keeps the app alive. **False everywhere in this release.** `HKWorkoutSession` exists on watchOS and on iOS 26 and later, and `androidx.health.services` exists on Wear OS, but neither is wired up yet, so every platform currently uses the recorded session described below. Branch on this query rather than on the OS version and your code will pick up native sessions when they land.
- `isSensorCollectionSupported()` — the OS also gathers heart rate and energy into that session by itself. **False everywhere in this release**, for the same reason: the live session is what collects, and there isn't one yet.

The second being false everywhere today, **a workout records only what you feed it**. On an Android phone that isn't a degraded fallback: Health Connect has no live-session concept, and inserting a session, batching your own data and updating it at the end is precisely the flow Google documents. Feed it with `addSamples(⋯)`, or by attaching a Bluetooth sensor.

`getStatistic(⋯)` returns `null` rather than a fabricated zero when nothing has been collected. Sessions aren't restored after the process dies — a killed workout is over, `end()` was never called, and nothing is written.

Ending a workout tells you what the platform would not keep. `WorkoutSample.WORKOUT_NOT_PERSISTED` is set when the session record itself has no write form — neither mobile platform accepts one through this API — and `WorkoutSample.SAMPLES_NOT_PERSISTED` names the data types whose samples were refused, comma separated. Power, speed and both cadences are the ones that bite: they're exactly what a bike or foot pod feeds in, and Health Connect has no single-value write form for any of them. Check both and upload what matters to you; the workout object itself is complete either way.

73.8. Bluetooth health sensors

```
HealthSensors sensors = Health.getInstance().getSensors();
if (!sensors.isSupported()) {
    return;
}
SensorScanSettings settings = new SensorScanSettings()
        .addProfile(HealthSensorProfile.HEART_RATE)
        .setTimeoutMillis(15000);

sensors.startScan(settings, new SensorDiscoveryListener() {
    public void sensorDiscovered(HealthSensor sensor) {
        sensors.connect(sensor, HealthSensorProfile.HEART_RATE,
                new SensorSessionOptions().setAutoReconnect(true))
            .onResult((session, err) -> {
                if (err != null) {
                    return;
                }
```

```
                session.addListener(new SensorSampleListener() {
                    public void sensorSample(SensorSession s,
                            HealthSample sample) {
                        Log.p("bpm: " + ((QuantitySample) sample)
                                .getValue(HealthUnit.COUNT_PER_MINUTE));
                    }
                    public void sensorStateChanged(SensorSession s,
                            SensorSessionState state) { }
                    public void sensorError(SensorSession s,
                            HealthException e) { }
                });
            });
    }
    public void scanFailed(HealthException e) {
        Log.e(e);
    }
});
```

`com.codename1.health.sensors` covers the adopted Bluetooth SIG profiles — Heart Rate, Cycling Power, Cycling and Running Speed & Cadence, Health Thermometer, Weight Scale, Blood Pressure and Glucose — so any conforming device works without per-vendor code.

This layer is built entirely on `com.codename1.bluetooth.le` and needs no platform health store, so it works identically everywhere Bluetooth LE does, including the desktop and JavaScript ports where no health store exists at all. It needs **Bluetooth** permissions, not health ones, and the build server makes the same distinction: an app that uses only this package isn't treated as a health-data app and gets neither a HealthKit entitlement nor a Google Play health-permissions review.

Speed and cadence sensors transmit cumulative counters rather than rates, with counters and event timers that wrap every 32 or 64 seconds. The framework differences them for you; hand-rolling that arithmetic is the most commonly botched part of these profiles.

`SensorSessionOptions.setWriteToStore(…)` defaults to `false` on purpose. When the OS is recording heart rate into HealthKit during a workout, a strap that also writes its own samples double-counts in every downstream average. Attach the session to the workout with `setWorkoutSession(…)` instead; turn write-through on for standalone measurements the OS knows nothing about, such as a weight from a scale.

A heart-rate session reports the rate. RR intervals are decoded by `HeartRateMeasurement` but aren't delivered as samples — they're the input to an HRV calculation, not a measurement of it, and no platform has a data type that would store them. If you need HRV, subscribe to characteristic `0x2A37` through `com.codename1.bluetooth.le` and call `HeartRateMeasurement.parse(…)` on the raw value yourself; both are public API.

Blood pressure is the exception: it's two values in one reading, which HealthKit models as a correlation and Health Connect as its own record type, and neither is implemented in this release. A cuff reading routed to the store fails with `TYPE_NOT_SUPPORTED` on both phones rather than being dropped without a word; the local and simulator stores keep it fine. Read it off the session and persist it yourself for now.

73.9. Nutrition

`com.codename1.health.nutrition` logs food and drink as a sparse set of nutrient amounts. A logged apple sets four fields and a scanned packaged food might set thirty; both platforms model this as a record with several dozen optional fields, and `NutritionSample` keeps that sparseness explicit rather than exposing forty nullable getters. A nutrient that was never measured reads back as `null`, not zero — the same distinction aggregate buckets make, and one that matters to someone managing their sodium.

`HealthDataType.NUTRITION` is local and simulator only in this release. Neither the wire format nor either port's type map knows the multi-nutrient record shape, so a read or write of it on a phone is refused with `TYPE_NOT_SUPPORTED` before it reaches the platform rather than being dropped without a word. Individual dietary quantities do reach the phones as ordinary quantity samples — `HYDRATION` on both, `DIETARY_ENERGY` on iOS — so use those when the number has to land in the user's real health store.

73.10. Build configuration

Health data is the most regulated data these APIs touch, and the build is strict about it.

Table 15. iOS build hints

Hint	Effect
`ios.NSHealthShareUsageDescription`	Why your app reads health data. **Required** if you touch the store; the build fails without it.
`ios.NSHealthUpdateUsageDescription`	Why your app writes health data. Required if you write.
`ios.health.backgroundDelivery`	Adds the background-delivery entitlement. Inferred from observer usage.
`ios.health.recalibrateEstimates`	Adds the estimate-recalibration entitlement.
`ios.health.required`	Adds `healthkit` to `UIRequiredDeviceCapabilities`.

Table 16. Android build hints

Hint	Effect
`android.health.read`	Comma-separated data types to read. **Required.**
`android.health.write`	Comma-separated data types to write.
`android.health.background`	Adds `READ_HEALTH_DATA_IN_BACKGROUND`.
`android.health.history`	Adds `READ_HEALTH_DATA_HISTORY`.
`android.health.privacyPolicyUrl`	Your privacy policy. **Required.**
`android.health.connectVersion`	Overrides the Health Connect client version.

Codename One **doesn't** inject a placeholder privacy string for health, unlike camera or Bluetooth. Apple reviews health purpose strings against what the app actually does, so a generic placeholder is precisely what gets an app rejected — it would not even achieve the "keeps the build working" goal.

The build fails with a message naming the hint instead.

The Android data types can't be inferred. A data type is referenced as a constant, which compiles to a field read, and the build server's class scanner records only type and method references — so the permission set genuinely can't be derived from your bytecode. Declaring it explicitly also matches Google Play policy, which requires you to request exactly what you use. Health Connect additionally raises `minSdkVersion` to 26 and requires `targetSdkVersion` 30 or higher.



HealthKit is entitlement-gated, unlike CoreBluetooth. The `com.apple.developer.healthkit` capability must be enabled for your App ID and present in the provisioning profile you build with, or the build fails at code-signing.

An app that only uses `Health.getSensors()` needs none of this. The sensor layer is built on `com.codename1.bluetooth.le` and touches no HealthKit, so the build injects neither the framework nor the entitlement nor the usage strings, and none of the hints above apply — it needs `ios.NSBluetoothAlwaysUsageDescription` instead. Reaching for the store is what pulls HealthKit in, and it's `getStore()` rather than `Health.getInstance()` that reports a missing usage string, so the sensor-only path stays clear of the whole apparatus.

73.11. Privacy and store policy

Read this section before shipping. Both stores enforce policies that go well beyond the technical permissions.

Apple. HealthKit data may not be used for advertising or use-based data mining beyond health, fitness and medical research, and may not be shared with third parties without explicit consent for that specific sharing. A privacy policy URL in App Store Connect is mandatory — HealthKit apps are rejected without one. HealthKit data may not be stored in iCloud. Purpose strings must be specific: "Reads your step count to show weekly activity trends," not "Needs health access."

Google. Health Connect access is gated by the Play Console health apps declaration form, and each permission is approved individually. Your app must implement a privacy-policy activity that's shown **before** the permission dialog — Codename One declares it for you, and this isn't optional: Health Connect declines to show its consent dialog to an app that lacks one. Play's Health Apps policy prohibits using the data for ads, for credit, insurance or employment decisions, or transferring it to a data broker.

What Codename One itself does. The framework never uploads health data anywhere. The simulator's event log records data types and counts but never sample values. And the build fails rather than inventing a purpose string on your behalf.

Practical rules. Request the narrowest type set, at the moment of use rather than at launch. Never persist samples to unencrypted `Storage`. Treat the on-device store as the source of truth and avoid server round-trips. Delete health data on account deletion.

Finally, the read-authorization asymmetry is a privacy **feature**, not an obstacle. iOS hides read denial so that an app can't infer what a user is concealing. Design your UI to respect that.

73.12. Simulating health

The simulator ships a scriptable health store under Simulate → Health, and every action is also callable from a test with `CN.execute("health:itemN")`.

Action	Effect
Grant / Deny All Permissions	The permissive and fully denied cases
Grant Write, Deny Read Without Error	Reproduces the iOS trap exactly
Load Demo Dataset (7 days)	Synthetic steps, heart rate, sleep and weight
Emulate HealthKit / Health Connect Permissions	Switches read-denial behaviour
Make Health Unavailable	Emulates a missing provider
`health:item10` / `health:item11`	Arm a one-shot query or save failure

Click "Grant Write, Deny Read Without Error" before you ship. It's the default-adjacent case real users will produce and the one a permissive store never shows you: authorization appears to succeed, the status reads `UNKNOWN`, and every query comes back empty with no error. If your UI shows an error or accuses the user of denying access, it's wrong.

The simulator's data is synthetic and seeded, not recorded from anyone. That's intentional: unlike Bluetooth traces, where the sensitive parts are identifiers that can be scrubbed, the sensitive part of a health sample **is** the value being asserted on, and a resting heart rate together with sleep timing and step cadence is quasi-identifying.

73.13. Troubleshooting

A query returns empty even though permission was granted. On iOS you can't distinguish a denied read from an empty store — this is by design. Use `hasAnyData(⋯)`, show "no data available," rather than an error, and offer `openHealthSettings()`. Reproduce it in the simulator with "Grant Write, Deny Read Without Error."

The chart shows zero for days the user was active. You are substituting zero for a `null` aggregate. An empty bucket means no data, not no steps.

Totals are double. A phone and a watch are both writing the same type, on iOS as much as on Android — aggregation runs on raw samples in shared code, so HealthKit's de-duplicating statistics engine never sees the query. Pin it with `addSource(⋯)`.

Background delivery never fires. No platform pushes health data to a closed app in this release — see the subscriptions section. Call `drainChanges()` from your background-fetch handler and branch on `isPushDelivery()` rather than assuming.

Replaying stored glucose records isn't implemented. `SensorSession.requestStoredRecords(⋯)` always fails with `NOT_SUPPORTED` in this release. Replay runs over the Record Access Control Point — a stateful protocol on a second characteristic — and shipping it untested against real meters would be worse than saying so. Live glucose notifications work normally.

Sample metadata doesn't reach the phone's health store. `HealthSample.getMetadata()` round-trips locally — the simulator, desktop and JavaScript store — but isn't persisted to HealthKit or Health Connect in this release, so a sample written and read back on a device returns without it. Health Connect has no arbitrary metadata at all, only a single client record id. Keep your correlation identifier in your own storage.

Recorded workouts aren't kept alive on Android. The manifest declares no foreground service, so a recorded workout lives exactly as long as your process does. If you need one to survive backgrounding, keep the app alive yourself.

`BODY_MASS_INDEX` is rejected on Android. Health Connect has no BMI permission or record — BMI is derived, not stored — so the build fails rather than requesting an unrelated body-composition permission on your behalf. Read `BODY_MASS` and `HEIGHT` and compute it. HealthKit does store BMI, so iOS is unaffected.

Deleting a sample by id does nothing. Health Connect deletes by record class plus id, so `HealthDeleteRequest.byId(…)` takes the data type alongside the identifier. Naming the wrong type deletes nothing and still reports success.

The weekly chart is off by a day twice a year. You are bucketing by a fixed 24 hours across a daylight-saving transition. Use `HealthInterval.calendarDays(1, tz)`.

The iOS build fails complaining about a usage description. That's intentional — add `ios.NSHealthShareUsageDescription` with text describing what your app actually does with the data.

The Health Connect permission dialog never appears. Health Connect requires the permissions-rationale activity, which Codename One declares only when it detects health usage. Check that the build log mentions the health manifest fragments.

Chapter 74. Printing

Codename One can hand a document to the platform printing system through the `com.codename1.printing` package. The API shows the native print dialog where the user picks a printer, adjusts options and confirms the job. It works on iOS (AirPrint), Android, desktop builds, the simulator, Windows and the JavaScript port.

The document is a file in `FileSystemStorage` identified by its path and mime type. Every printing platform accepts PDF (`application/pdf`) and common image types (`image/png`, `image/jpeg`).

74.1. Quick start: Print a PDF

`Printer.printPDF(path, listener)` is shorthand for `Printer.print(path, "application/pdf", listener)`. Use the general `print` variant for other document types, for example `Printer.print(photoPath, "image/jpeg", listener)`.

Gate the call on `isPrintingSupported()`. When it returns `false` the listener receives a `FAILED` result without any UI appearing, so an unguarded call does no harm, but a well-behaved app hides or disables its print button instead.

74.2. Printing images

`Printer.printImage(image, listener)` prints a `com.codename1.ui.Image` directly. The image is encoded to a temporary PNG file behind the scenes and the file is deleted once the print flow finishes:

This pairs well with drawing on a mutable image, so you can compose a printable page (a receipt, a ticket, a chart) with the standard `Graphics` API and send it to paper without generating a PDF.

74.3. Handling the result

The listener receives a `PrintResult` on the EDT, exactly once per request. The listener may be `null` for fire-and-forget printing. There are three statuses:

- `COMPLETED` – the job was handed to the platform printing system. Most platforms can't observe the physical printer, so this means "queued/sent," not "paper came out."
- `CANCELLED` – the user dismissed the print dialog without printing, on platforms that can observe the dialog outcome.
- `FAILED` – the job couldn't start or the printing system reported an error. `getError()` may carry a short platform message.

Cancel detection is best-effort. iOS, Windows, PDF printing on Android and image printing in the simulator report `CANCELLED` reliably. Image printing on Android, PDF hand-off on the desktop and the browsers report `COMPLETED` once the job is posed, since their printing pipelines don't expose the user's choice.

74.4. Permissions and the user gate

Printing requires no permission, entitlement, manifest entry or build hint on any platform. This is safe because printing is always user-confirmed: the API can only pose the native print dialog, and nothing reaches a printer until the user approves the job there. In that sense it follows the same trust model as the share API.

Silent printing - sending a job to a preselected printer without showing a dialog - isn't supported by design.

Device policy can disable printing entirely, for example a managed Android work profile with the printing restriction or an iOS MDM profile that blocks AirPrint. Those cases surface as `isPrintingSupported()` returning `false` or as a `FAILED` result, so the result-handling code above covers them.

74.5. Platform notes

- iOS – `UIPrintInteractionController` presents the AirPrint sheet; on the iPad it appears as a popover. The completion handler reports completed, cancelled or failed.
- Android – the `android.print` framework. PDFs stream through a `PrintDocumentAdapter`; images go through the support library `PrintHelper`, which reports completion but can't observe cancellation. Printing requires a foreground activity, so calls from background mode fail.
- Desktop / simulator – images print through `java.awt.print.PrinterJob` with the native dialog; PDFs are handed to the operating system's default PDF print flow.
- Windows (native port) – the Win32 print dialog with GDI rendering; PDF pages render through the `Windows.Data.Pdf` API at the printer's resolution.
- JavaScript – the document loads into a hidden frame and prints through the browser's print dialog. Browsers don't report whether the user printed or cancelled, and PDF behavior varies with the browser's built-in viewer.

74.6. Trying it in the simulator

The `PrinterSample` in the samples project demonstrates both flows: it prints a generated image and downloads a small PDF to print. On the desktop the native print dialog of your operating system appears, which makes it a convenient way to test result handling - print to a file or preview instead of paper.

Chapter 75. Deeper Network Connectivity

Codename One ships a set of APIs under `com.codename1.io.wifi`, `com.codename1.io.bonjour`, `com.codename1.io.usb` and an extension of `com.codename1.io.NetworkManager` that lets apps inspect and manage the device's local network beyond plain HTTP.

A quick connectivity check is `NetworkManager.getInstance().isConnected()` — it reads the cached platform network state and returns immediately without an HTTP probe, so it's safe to call from the EDT before deciding whether to fire a `ConnectionRequest`.

These APIs share three guarantees:

- Every callback runs on the EDT so you can update UI directly from the result.
- The build pipeline injects the matching Android permissions, iOS entitlements, and `NSXxxUsageDescription` / `NSBonjourServices` Info.plist entries automatically by scanning the classpath. Apps that never reference these classes see no manifest or entitlement changes.
- The simulator implements best-effort versions of each API so you can wire and test your UI without a device. The simulator also prints the list of permissions/entitlements each API will need in a real build the first time you call it.

The simulator's "API usage report" prints to stdout on JVM shutdown so you can spot anything you pulled in via a transitive cn1lib.

75.1. WiFi information

The fastest way to know whether the user is on WiFi, what SSID they joined, and what their local IP is:

```
if (WiFi.isInfoSupported()) {
    String ssid = WiFi.getCurrentSSID();   // e.g. "Codename One"
    String bssid = WiFi.getBSSID();        // "aa:bb:cc:11:22:33"
    String gw = WiFi.getGateway();         // "192.168.1.1"
    String ip = WiFi.getIp();              // "192.168.1.42"
}
```

Any of these calls may return `null` if the device isn't on WiFi, if the user denied a runtime permission, or if the platform never exposed the value to apps. The framework normalizes the wrapping that Android adds to the SSID and treats `<unknown ssid>` and the obfuscated BSSID `02:00:00:00:00:00` as `null`.

75.1.1. Required permissions and entitlements

The build pipeline injects the following automatically the first time it sees `com.codename1.io.wifi.WiFi` on the classpath:

Platform	Injected	Notes
Android	`ACCESS_WIFI_STATE`, `ACCESS_NETWORK_STATE`	"normal" permissions; no runtime prompt
Android API 26+	`ACCESS_FINE_LOCATION` (`maxSdkVersion=32`)	Required by the OS to return a real SSID
Android API 33+	`NEARBY_WIFI_DEVICES` with `neverForLocation`	Replaces fine-location for scan flows
iOS	`com.apple.developer.networking.wifi-info` entitlement	Required since iOS 13
iOS	`NSLocationWhenInUseUsageDescription`	iOS still checks CoreLocation behind the scenes

You can override any injected automatically value via the standard build hints:

```
ios.locationUsageDescription=Allow access to read your Wi-Fi network name to discover printers on your network.
android.xpermissions=<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

75.2. WiFi scan

Scanning returns an array of `WiFiNetwork`s sorted by signal strength. The callback fires once.



iOS doesn't expose a public WiFi scan API. On iOS `WiFi.scan(…)` always reports an `UnsupportedOperationException`. Plan for a graceful fallback if your UI exposes a scan button.

Android throttles scans to 4 per 2 minutes per foreground app on API 28+; throttled scans return cached results, not an error.

75.3. WiFi connect

To associate the device with a specific SSID:

- On Android 10+ the OS shows a system confirmation dialog with the SSID before joining; this dialog can't be bypassed.
- On Android 9 and below the framework falls back to the legacy `WifiConfiguration` API and joins without an in-app prompt.
- On iOS 11+ this uses `NEHotspotConfiguration`. The user is shown a system prompt the first time the app tries to join the SSID; subsequent attempts reuse the cached configuration.
- `WiFiSecurity` must match the AP's actual security mode (`OPEN`, `WEP`, `WPA_PSK`, `WPA3_SAE`). Passing the wrong mode causes the join to fail.

`WiFi.disconnect(ssid)` releases the request on Android 10+ and removes the hotspot configuration on iOS.



On iOS the build pipeline also enables the `NetworkExtension.framework` and the `HotspotConfiguration` entitlement only when it sees a call to `WiFi.connect(…)` or `WiFi.disconnect(…)`. Apps that only **read** SSID/BSSID won't pull in the framework, which keeps App Store API-usage scans clean.

75.4. Bonjour / mDNS discovery

`BonjourBrowser` watches the local network for services of a given mDNS type. `BonjourPublisher` advertises a service.

```
BonjourBrowser browser = BonjourBrowser.browse("_http._tcp.",
        new BonjourServiceListener() {
    @Override
    public void onServiceResolved(BonjourService s) {
        Log.p("Found " + s.getName() + " at " + s.getHost() + ":" + s.getPort());
    }
    @Override
    public void onServiceLost(BonjourService s) { /* null */ }
    @Override
    public void onBrowseError(Throwable t) { Log.e(t); }
});

// when done:
browser.stop();
```

To advertise an HTTP server you wrote on port 8080:

75.4.1. iOS requirements

Bonjour on iOS 14+ returns no results unless:

- The Info.plist `NSLocalNetworkUsageDescription` key explains why the app needs local-network access.
- The Info.plist `NSBonjourServices` array lists each service type the app expects to find.

The build pipeline injects both keys automatically when `com.codename1.io.bonjour` is on the classpath, with `_http._tcp.` as the seed type. To publish or browse a custom type, set the build hint:

```
ios.NSBonjourServices=_myapp._tcp.,_http._tcp.
ios.NSLocalNetworkUsageDescription=Discover other instances of MyApp on this Wi-Fi network.
```

The comma-separated values are expanded into the Info.plist array at build time.

75.4.2. Android requirements

The pipeline adds `CHANGE_WIFI_MULTICAST_STATE` automatically. `NsdManager` does the rest; there is no

runtime permission prompt.

75.4.3. Simulator behaviour

If `javax.jmdns.JmDNS` is on the simulator classpath, browse calls dispatch through it. Otherwise the listener receives a single `UnsupportedOperationException` and the simulator prints a hint. Add JmDNS to your **application's** `executable-jar` / simulator profile (typically the `<profile id="executable-jar">` block in your cn1app's `common/pom.xml` or your `javase/pom.xml`) to exercise real discovery in the simulator:

```
<!-- in YOUR app's pom, NOT in the framework / common pom that gets shipped
     to the device. The cn1app archetype's executable-jar profile is the
     right place because it only applies when launching the simulator. -->
<dependency>
    <groupId>org.jmdns</groupId>
    <artifactId>jmdns</artifactId>
    <version>3.5.9</version>
    <scope>runtime</scope>
</dependency>
```



Don't add JmDNS to the cn1app `common/pom.xml` outside a simulator-scoped profile — it would then ship with your Android / iOS binaries even though both platforms supply native mDNS implementations.

75.5. Network type change events

Subscribe to network type transitions (`WiFi <-> Cellular <-> Ethernet <-> None`):

```
NetworkManager.getInstance().addNetworkTypeListener(new NetworkTypeListener() {
    @Override
    public void onNetworkTypeChanged(int oldType, int newType, boolean vpnActive) {
        if (newType == NetworkManager.NETWORK_TYPE_NONE) {
            // offline -- queue uploads for later
        }
        if (vpnActive) {
            // refuse to roam to corporate-only resources
        }
    }
});
```

The current snapshot is available without subscribing:

```
int type = NetworkManager.getInstance().getCurrentNetworkType();
boolean vpn = NetworkManager.getInstance().isVPNActive();
```

The platform implementations are:

- **Android**: `ConnectivityManager.registerNetworkCallback` on API 21+, falling back to the legacy `CONNECTIVITY_ACTION` broadcast on older devices.
- **iOS**: `SCNetworkReachability` with a kernel callback scheduled on the main run loop.
- **Simulator**: derives the type from `NetworkInterface.getDisplayName`; transitions aren't synthesized.

VPN detection is best-effort and intentionally heuristic. On Android it uses `NetworkCapabilities.TRANSPORT_VPN` plus an interface-name probe; on iOS it inspects `utun*`, `tun*`, `ppp*` and `ipsec*` interfaces. Use it as a hint, not as a security boundary.

75.6. WiFi Direct (Wi-Fi P2P)

WiFi Direct lets two devices form a peer-to-peer link without an access point. The API is Android-only — iOS uses MultipeerConnectivity for similar scenarios and is intentionally out of scope.

The build pipeline injects `CHANGE_WIFI_STATE`, `ACCESS_WIFI_STATE`, `ACCESS_NETWORK_STATE`, `ACCESS_FINE_LOCATION` and `NEARBY_WIFI_DEVICES` automatically when `WiFiDirect` is on the classpath, plus a `<uses-feature android:name="android.hardware.wifi.direct"/>` declaration so devices without the radio can still install the app via Play Store filtering.

75.7. USB host

The USB API in `com.codename1.io.usb` lets the device act as a USB host and talk to attached peripherals — a barcode scanner, a serial converter, a microcontroller. It's **Android-only**; iOS has no third-party USB host access and the simulator stubs everything out.

```
if (!Usb.isSupported()) { return; }

Usb.addDeviceListener(new UsbDeviceListener() {
    @Override
    public void onDeviceAttached(UsbDevice d) {
        if (d.getVendorId() == 0x0403 && d.getProductId() == 0x6001) {
            Usb.requestPermission(d);
        }
    }
    @Override
    public void onDeviceDetached(UsbDevice d) { }
    @Override
    public void onPermissionResult(UsbDevice d, boolean granted) {
        if (granted) {
            try (InputStream in = Usb.openInputStream(d, 0x81);
                 OutputStream out = Usb.openOutputStream(d, 0x02)) {
                out.write("AT\r\n".getBytes());
                byte[] buf = new byte[64];
                int n = in.read(buf);
                // null
            } catch (IOException e) { Log.e(e); }
        }
```

```
    }
});
```

The build pipeline injects:

- `<uses-feature android:name="android.hardware.usb.host" android:required="false"/>`.

If you want the OS to launch your app when a matching device is plugged in, ship a `device_filter.xml` resource and declare an `<intent-filter>` via the `android.xintent_filter` build hint:

```
<!-- native/android/res/xml/device_filter.xml -->
<resources>
    <usb-device vendor-id="1027" product-id="24577" />
</resources>
```

```
android.xintent_filter=<intent-filter><action android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED" /></intent-filter><meta-data android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED" android:resource="@xml/device_filter" />
```

75.8. Releasing platform resources

Most of these APIs hold background resources (broadcast receivers, network callbacks, NSNetService delegates) that you must release explicitly:

- `BonjourBrowser` — call `.stop()`.
- `BonjourPublisher` — call `.unpublish()`.
- `WiFiDirect` — call `.stopDiscovery()` and `.disconnect()`.
- `WiFi.connect(…)` — call `WiFi.disconnect(ssid)` when the user navigates away.
- `Usb.addDeviceListener(…)` — call `removeDeviceListener(…)`.
- `NetworkManager.addNetworkTypeListener(…)` — call `removeNetworkTypeListener(…)`.

Each of these keeps a radio or system service awake. Bonjour browsers and WiFi scans in particular drain battery noticeably — they're fine during a few user-facing screens but shouldn't stay armed for the lifetime of the app. Release them as soon as the screen that needs them is no longer visible.

Chapter 76. Signing, certificates & provisioning

In this section you try to explain how to acquire certificates for the various platforms and how to set them up.

The good news is that this is a "one time issue" and once it’s done the work becomes easier (except for the case of iOS where a provisioning profile should be maintained).

76.1. Common terms in signing & distribution

This section uses some terms that you might not be familiar with if you haven’t used cryptography or built mobile apps before. Here is a quick "FAQ" covering some common terms/concepts in this section.

76.1.1. What’s a certificate

Certificates use cryptographic principals to "sign" data (for example, an application). Think of them as you would think of a company stamp, you use them to sign an app so users know who it’s from.

76.1.2. What’s provisioning

Provisioning provides the hints/guidelines for the application install. For example, if an application needs some service from the OS such as push it can request that with provisioning.

In iOS provisioning is separate from the app and you need to also define the devices supported by the app during development.

76.1.3. What’s a signing authority

Certificates are issued by a signing authority which is a body that certifies that you’re who you say you are. Apple issues certificates for iOS and is in effect a signing authority.

Android uses self-signed certificates which don’t use a signing authority so anyone can ship an Android app.

The logic with the Android approach is that a signature indicates that you’re the same person who earlier shipped the app. Hence an app will be updated with the exact same certificate.

76.1.4. What’s UDID

The UDID is the Universal Device Identifier. It identifies mobile devices uniquely, notice that some operating systems for example, iOS block access to this value due to privacy concerns.

You need the iOS device UDID value during development to add the device into the list of allowed devices.



Don’t use an app to get the UDID!
Most return the wrong value. Use the Finder device summary (macOS), Apple Configurator, or a trusted service such as http://get.udid.io/ which seems to work rather well

76.1.5. Should you reuse the same certificate for all apps

For iOS yes. It’s designed to work in that way.

You would recommend it for all platforms for simplicity but some developers prefer creating per-application certificates for Android. The advantage here is that you can transfer ownership of the application later on without giving away what’s effectively "your house keys."

76.2. Certificate Wizard

Codename One features a standalone Certificate Wizard to manage signing assets for Apple, Android and desktop targets. It replaces the old certificate wizard inside `CodenameOneSettings` and is launched as a Maven tool bound to the current project.

Run the wizard from a Codename One Maven project:

```
mvn cn1:certificatewizard
```

The Maven goal resolves the wizard artifact and its runtime dependencies from Maven, then launches it with the current project’s settings path. The wizard uses the Codename One cloud signing API for Apple assets and local `keytool` for Android keystores. On a first run the overview shows that the App Store Connect API key isn’t configured and links to the API key page. This is an Apple credential that allows the cloud service to create and manage certificates, bundle IDs, devices, provisioning profiles, APNs keys and Mac signing assets for your Apple Developer Program team.

76.2.1. Creating the App Store Connect API key

Open App Store Connect > Users and Access > Integrations > App Store Connect API [https://appstoreconnect.apple.com/access/integrations/api]. If this is the first time your team uses the API, App Store Connect asks you to request API access first. Click **Request Access**, accept the terms, and submit the request.

Once access is enabled, go to **Team Keys** and click **Generate API Key**. Give the key a recognizable name, such as `Codename One Signing`, and choose **Admin** access.



Use **Admin** access for the Certificate Wizard. The wizard needs to create and list signing certificates, bundle IDs, devices, and provisioning profiles. App Manager or Developer access may not be sufficient for these operations.

After creating the key, App Store Connect shows the **Issuer ID** above the table and the **Key ID** in the key row. Copy both values into the wizard. Download the API key file. Apple only allows the `.p8` file

to be downloaded once, so store a backup in a secure location.

In the Codename One Certificate Wizard, fill these fields:

- **Key ID**: The value shown in the generated key row in App Store Connect. This isn't your Apple ID.
- **Issuer ID**: The UUID shown above the Team Keys table on the App Store Connect API page.
- **Private key .p8**: Click **Import .p8 file** to select the downloaded file with the native file chooser, or paste the full file contents, including the `BEGIN PRIVATE KEY` and `END PRIVATE KEY` lines.

The `.p8` private key is sent to the Codename One cloud signing service and isn't returned by the API later. If you delete the key from App Store Connect or lose the local `.p8` backup, generate and store a new API key.

When you download a generated `.p12` certificate from the wizard, the **P12 password** field is a new password that you choose for that downloaded file. It's not the App Store Connect Issuer ID, and it doesn't come from Apple.

76.2.2. Recommended fast path: Auto Setup

After the API key is stored, the top toolbar shows **Auto Setup** next to the API key status. This is the recommended path for a normal Codename One project. It creates the common Apple signing assets for the current project and installs the downloaded files into `codenameone_settings.properties`.

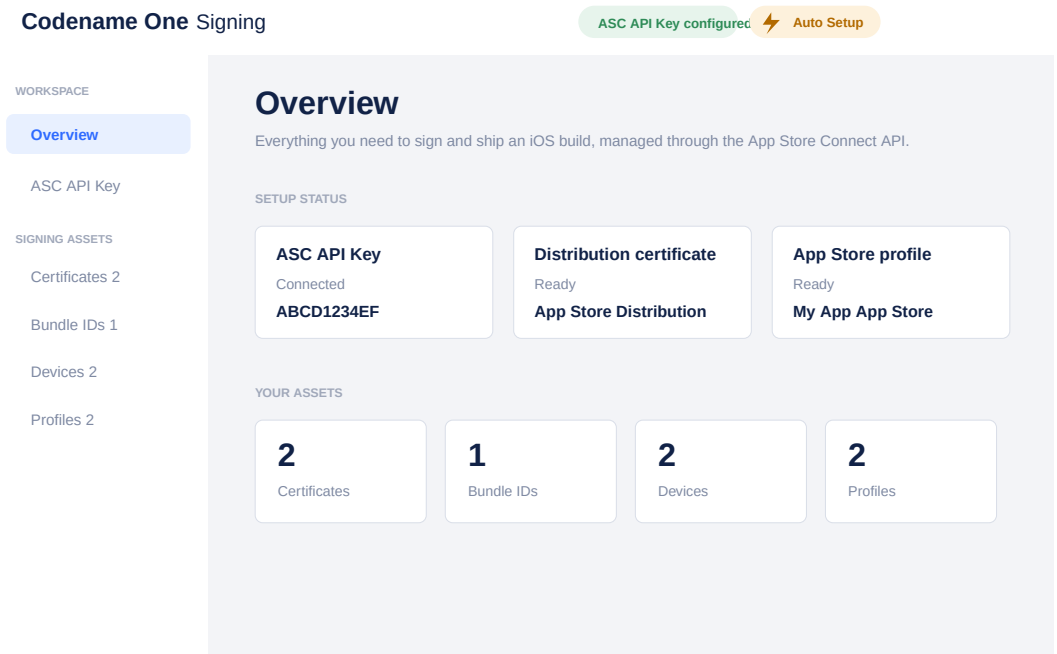


Figure 345. Certificate wizard overview with mocked signing assets

Click **Auto Setup**. The wizard reads the current project's `codenameone_settings.properties`, uses `codename1.packageName` as the bundle ID, uses `codename1.displayName` or `codename1.mainName` as the app name, enables push notifications for the bundle ID, and then creates or reuses the signing assets it needs.

The fast path creates the common Apple signing setup:

- A Bundle ID for the current project, if it doesn't already exist.
- An iOS Development certificate, if no compatible active certificate exists.
- An iOS Development provisioning profile, if at least one development device is registered.
- An iOS Distribution certificate, if no compatible active certificate exists.
- An App Store provisioning profile.
- Debug signing settings for the development certificate/profile pair.
- Release signing settings for the App Store certificate/profile pair.
- Mac App Store or Developer ID assets when you use the Mac Signing page.

Development provisioning requires at least one registered device. If no devices are registered, the wizard defers development profile creation and continues with the App Store signing assets. Register the device in the **Devices** section and run **Auto Setup** again to fill in the debug signing settings.

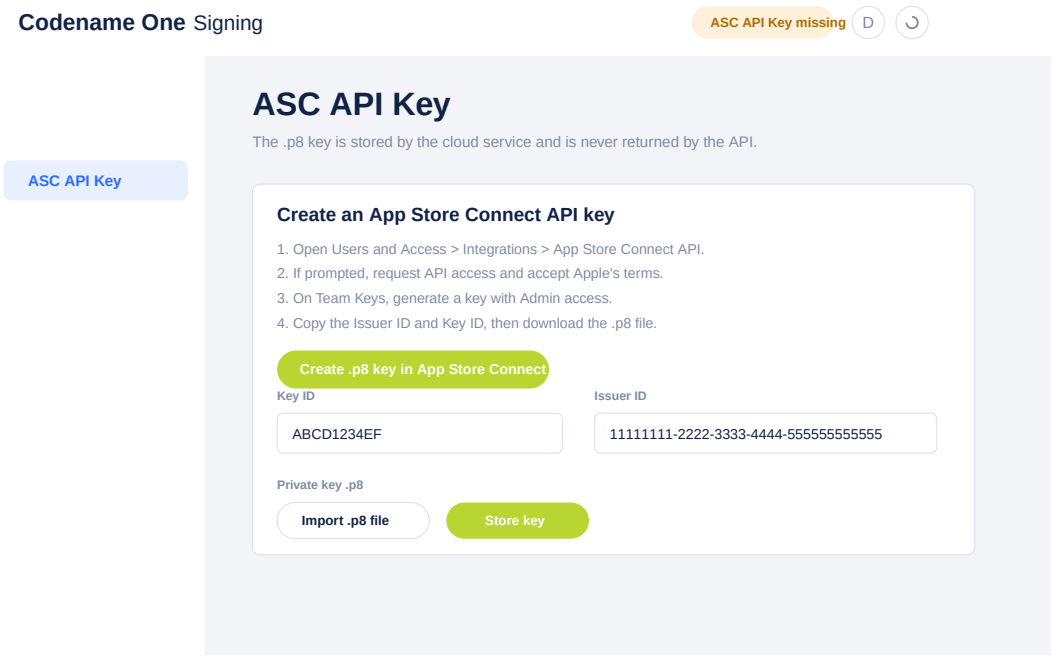


Figure 346. Certificate wizard API key screen with mocked values

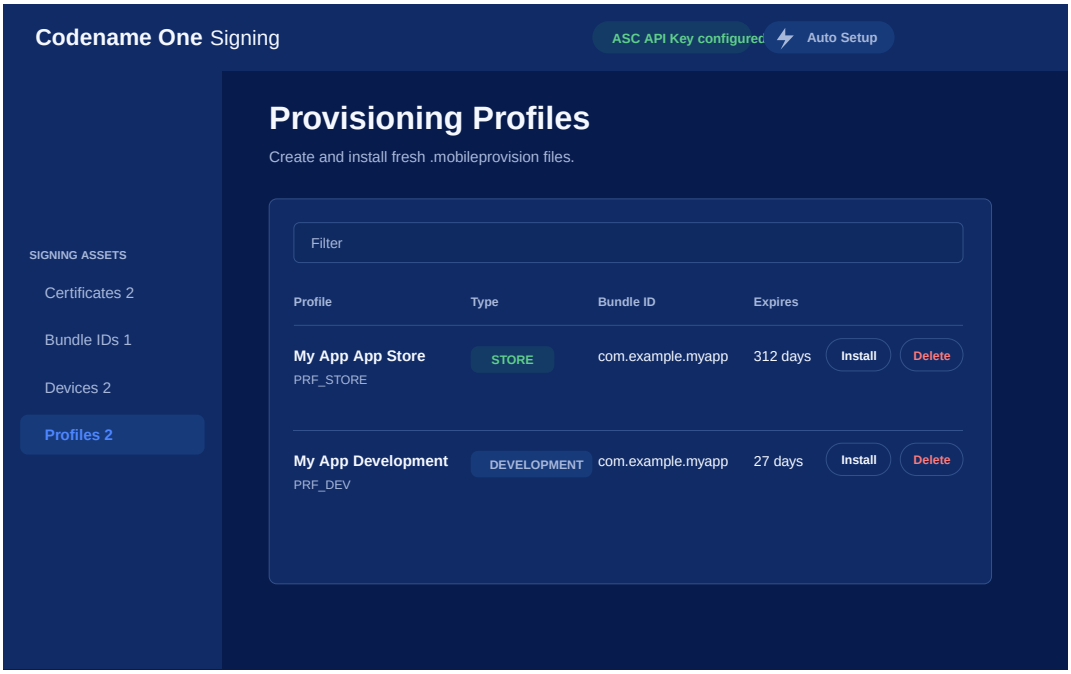


Figure 347. Certificate wizard profiles screen with mocked values

The wizard writes downloaded Apple `.p12`, `.mobileprovision` and Mac `.provisionprofile` files into the current project signing configuration. For a development iOS profile it updates the iOS debug settings. For an App Store iOS profile it updates the iOS release settings. For Mac profiles it updates `codename1.mac.certificate`, `codename1.mac.certificatePassword`, `codename1.mac.provision` and the relevant `macNative.*` hints.

76.2.3. Mac signing and notarization

The **Mac** section uses the same App Store Connect API key and the unified Apple signing API. It supports Mac App Store certificates/profiles and Developer ID certificates for direct distribution. Developer ID builds can use notarization credentials derived from the stored App Store Connect key when the build server performs notarization.

Mac native signing uses the project settings documented in the Mac native chapter:

- `codename1.mac.certificate`: The generated or imported Mac `.p12`.
- `codename1.mac.certificatePassword`: The password for that `.p12`.
- `codename1.mac.provision`: The Mac `.provisionprofile`.
- `codename1.arg.macNative.enabled=true`: Enables the Mac native signing path.
- `codename1.arg.macNative.distribution=appStore` or `developerID`: Chooses the signing channel.

76.2.4. Android signing

The **Android** section generates a local self-signed Android keystore using the JDK `keytool` executable. It stores the file under `androidCerts/KeyChain.ks` in the project by default and writes:

- `codename1.android.keystore`
- `codename1.android.keystoreAlias`

- `codename1.android.keystorePassword`



Back up the generated keystore and password. Android apps already published with one certificate can't be updated with a different certificate.

76.2.5. Windows signing

The **Windows** section doesn't issue certificates. Windows Authenticode certificates must come from a code-signing certificate authority or a managed signing service such as Azure Trusted Signing, DigiCert KeyLocker, GlobalSign GCC, Sectigo, or another OV/EV provider. Once you have a `.pfx` or `.p12`, configure:

- `codename1.windows.signing.certificate`
- `codename1.windows.signing.password`
- `codename1.windows.signing.timestamp`

76.2.6. Clearing cached signing data

The **Clear Signing Data** section is a privacy tool, not a recommended part of normal signing setup or troubleshooting. Use it only when you need to delete cached signing data owned by the current Codename One account from the cloud signing service.

It clears stored Apple API credentials, generated certificate private keys, provisioning profile metadata, APNs keys and notarization data. It doesn't delete assets from Apple, but future cloud builds will need signing data to be regenerated or re-imported.

76.2.7. Manual asset management

Use the individual wizard sections when you need to inspect assets, register a device, sync certificates with Apple, create an ad-hoc or Mac profile, install a specific certificate/profile pair, generate an Android keystore, or delete/revoke an obsolete asset.

The asset tables are live-filterable and sortable. Click a row to show the values returned by the signing API. Existing Apple assets can't currently be edited through this API; for changes, create a new asset or delete/revoke the old one where Apple permits it. Destructive actions such as deleting a profile, deleting an APNs key, deleting the stored API key, or revoking a certificate prompt for confirmation.

76.2.8. Selecting devices

Development and ad-hoc provisioning profiles can only include devices that are registered on your Apple Developer account. In the standalone wizard, open the **Devices** section to register a device by name and UDID before creating a development or ad-hoc provisioning profile.



Figure 348. Devices form

Select the ones that you want to include in your provisioning profile and click next.



Apple allows up to 100 devices per device type (iPhone, iPad, Apple Watch, Apple TV, and Apple Vision) for testing purposes in each membership year. Make sure every device you intend to install builds on is registered here before you generate a new provisioning profile.

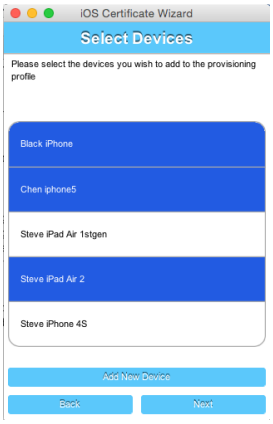


Figure 349. After selecting devices

If you don't have any devices registered yet, you can click the "Add New Device" button, which will prompt you to enter the UDID for your device.

76.2.9. Decisions & edge cases



If you already have iOS P12 development/distribution certificates you should reuse them for all your apps from that account and you shouldn't regenerate them

After you click `Next` on the device form, the wizard checks to see if you already have a valid certificate. If your project already has a valid certificate and it matches the one that's active in your Apple developer account, then it will use the same certificate. If the certificate doesn't match the currently active one, or you haven't provided a certificate, you will be prompted to overwrite the old certificate with a new one.

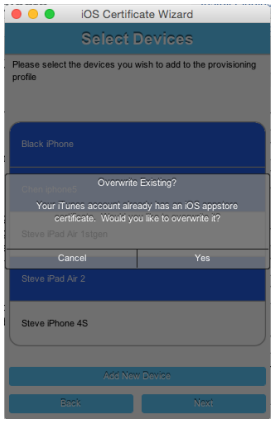


Figure 350. Prompt to overwrite existing certificate

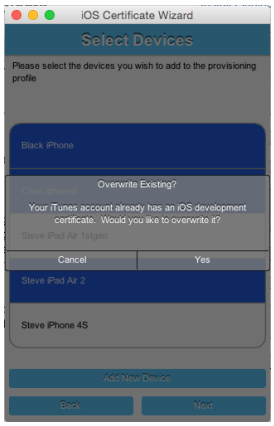


Figure 351. Prompt to overwrite other certificate

The same decision need to be made twice: Once for the development certificate, and once for the App Store distribution certificate.



Each Apple Developer account can have three active iOS Development certificates and three active iOS Distribution certificates at a time. Reuse existing certificates whenever possible to avoid hitting this limit.



You can revoke a certificate when you have an application in the store shipping with said certificate!
This won't affect the shipping app see this [http://stackoverflow.com/questions/6320255/if-i-revoke-an-existing-distribution-certificate-will-it-mess-up-anything-with].

Why Two Certificates?

A typical iOS app has at least two certificates:

- Development - this is used during development and can't be shipped to 3rd parties. An application signed with this certificate can only be installed on one of the up to 100 devices listed above.
- Distribution - this is used when you're ready to upload your app to App Store Connect whether for final shipping or beta testing. Notice that you can't install a distribution build on your own device. You need to upload it to App Store Connect.
- Two push certificates exist, separate from the signing certificates. Don't confuse them!

They're used to authenticate with Apple when sending push messages.

76.2.10. App IDs and provisioning profiles

The next form in the wizard asks for your app's bundle ID. This should have been pre-filled, but you can change the app ID to a wildcard ID if you prefer.

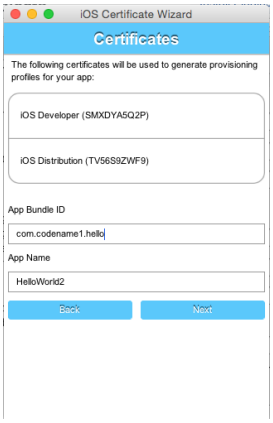


Figure 352. Enter the app bundle ID

Wildcard Card Provisioning

Wildcard IDs such as `com.mycompany.` or `even` allow one generic development profile to cover multiple applications. The standalone Certificate Wizard can reuse an existing wildcard profile, but wildcard apps can't use capabilities such as push notifications and aren't suitable for App Store distribution.

76.2.11. Installing files locally

Once the wizard is finished generating your provisioning profiles, you should click `Install Locally`, which will open a file dialog for you to navigate to a folder in which to store the generated files.

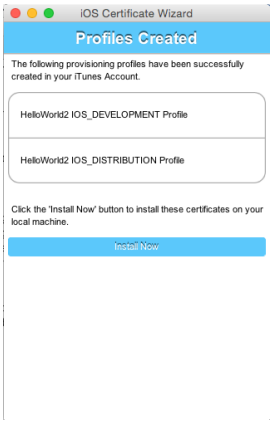


Figure 353. Install files locally

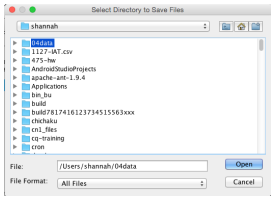


Figure 354. Select directory to save files in

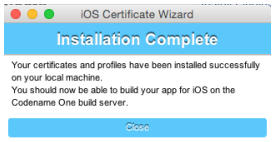


Figure 355. Final Done Message



You can see the password for the P12 files in the `codenameone_settings.properties` file. You can use the values defined there when creating a new application

76.2.12. Building your app

After selecting your local install location, and closing the wizard, you should see the fields of the "iOS Signing" properties panel filled in. You should now be able to send iOS debug or App Store builds without the usual hassles.

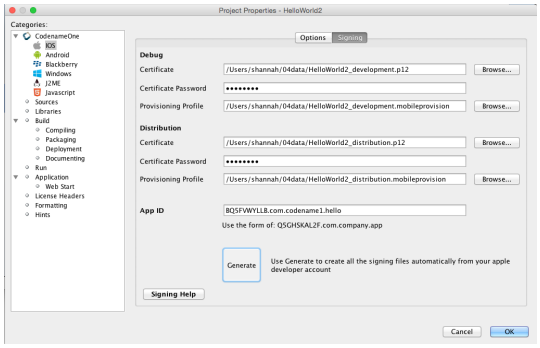


Figure 356. Filled in signing panel after wizard complete

76.3. Advanced iOS signing



You should use the Certificate Wizard, if you don't have a Mac. This section is here for reference and edge cases that don't work with the Certificate Wizard

iOS signing has two distinct modes: App Store signing which is valid for distribution via App Store Connect (you won't be able to run the resulting application without submitting it to Apple) and development mode signing.

You've two major files to keep track of:

1. **Certificate** - your signature
2. **Provisioning Profile** - details about the application and who is allowed to execute it

You need two versions of each file (4 total files) one pair is for development and the other pair is for uploading to App Store Connect.



You need to use a Mac to create a certificate file for iOS

The first step you need to do is signing up as a developer to Apple's iOS development program [http://developer.apple.com/], even for testing on a device this is required!
This step requires that you pay Apple on an annual basis.

The Apple website will guide you through the process of applying for a certificate at the end of this process you should have a distribution and development certificate pair. After that point you can log in to the iOS provisioning portal [https://developer.apple.com/ios/manage/overview/index.action] where there are plenty of videos and tutorials to guide you through the process. Within the iOS provisioning portal you need to create an application ID and register your development devices.

You can then create a provisioning profile which comes in two flavors:

- Distribution - for building the release version of your application
- Development - the development provisioning profile needs to contain the devices on which you want to test.

You can then configure the 4 files in the IDE and start sending builds to the Codename One cloud.

76.4. Provisioning profile & certificates visual guide

One of the hardest parts in developing for iOS is the certificate & provisioning process. In this step-by-step guide you walk over the manual certificate generation process. Notice that the UI for the Apple website changes occasionally but the basic process remains the same...

Start by logging in to the iOS-provisioning portal



Figure 357. Login for the iOS provisioning portal

In the certificates section you can download your development and distribution certificates.

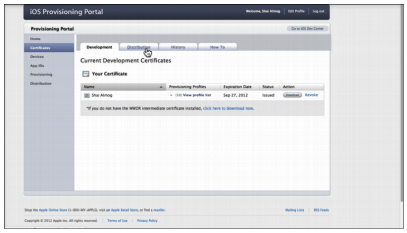


Figure 358. Download development provisioning profile

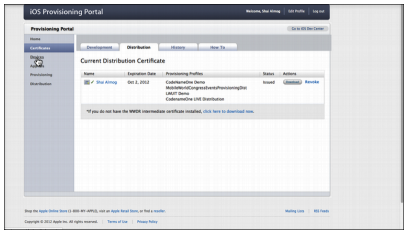


Figure 359. Download distribution provisioning profile

In the devices section add device ids for the development devices you want to support. Notice no more than 100 devices are supported!

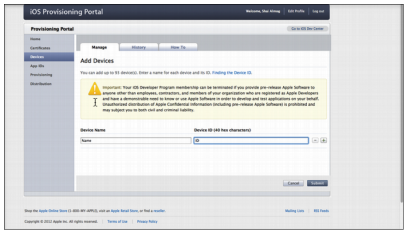


Figure 360. Add devices

Create an application ID; it must match the package identifier of your application exactly!

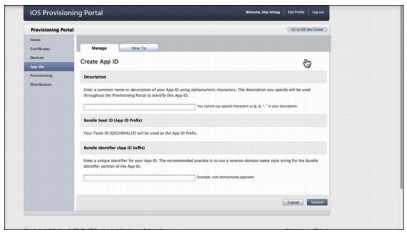


Figure 361. Create application ID

Create a provisioning profile for development, make sure to select the right app and make sure to add the devices you want to use during debug.

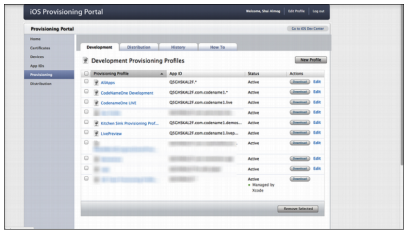


Figure 362. Create provisioning profile step 1

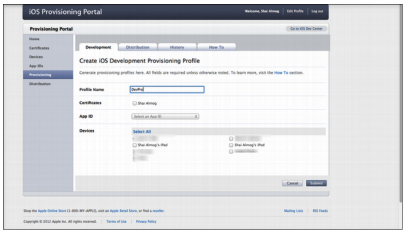


Figure 363. Create provisioning profile step 2

Refresh the screen to see the profile you created and press the download button to download your development provisioning profile.

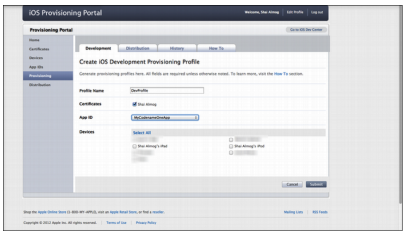


Figure 364. Create provisioning profile step 3

Create a distribution provisioning profile; it will be used when uploading to the App Store. No need to specify devices here.

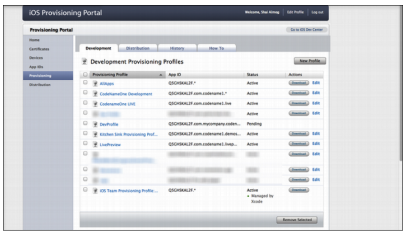


Figure 365. Create distribution provisioning profile

Download the distribution provisioning profile.

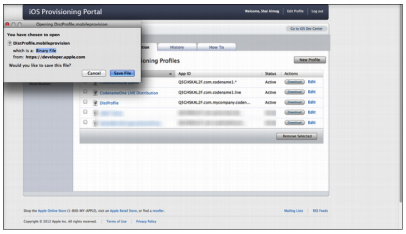


Figure 366. Download distribution provisioning profile

You can now import the cer files into the Keychain tool on a Mac by double clicking the file, on Windows the process is slightly more elaborate

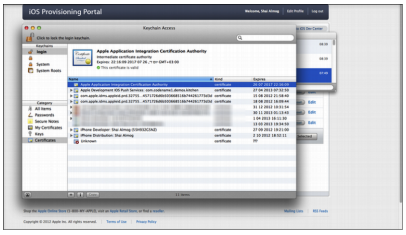


Figure 367. Import cer files

You can export the p12 files for the distribution and development profiles through the keychain

tool

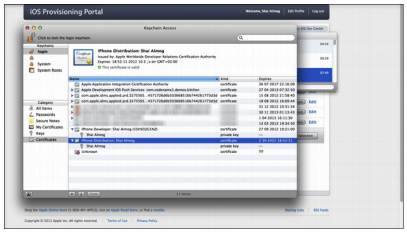


Figure 368. Export p12 files

In the IDE you enter the project settings, configure your provisioning profile, the password you typed when exporting and the p12 certificates. It's now possible to send the build to the server.

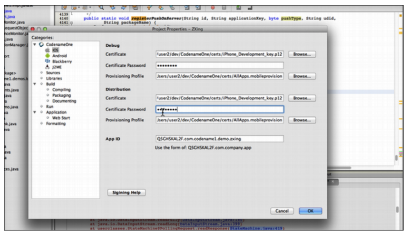


Figure 369. IOS Project Settings

76.4.1. Configuring `codenameone_settings.properties` for manual signing

The Codename One build servers read signing assets from `codenameone_settings.properties`. When you bypass the Certificate Wizard, populate the following keys manually so the packaging process can locate your certificates, provisioning profiles, and keystores:

iOS

- `codename1.ios.debug.certificate` / `codename1.ios.debug.certificatePassword` / `codename1.ios.debug.provision`: Development P12 and provisioning profile for device/debug builds.
- `codename1.ios.release.certificate` / `codename1.ios.release.certificatePassword` / `codename1.ios.release.provision`: Distribution P12 and provisioning profile for App Store/TestFlight builds.
- `codename1.ios.certificate`, `codename1.ios.certificatePassword`, and `codename1.ios.provision`: Legacy/global defaults. The IDE uses these as fallbacks when the debug/release-specific properties are blank, so keep them aligned with your preferred certificates.
- `codename1.ios.appext.<Name>.provision`: Provisioning profile for an embedded app extension (for example the generated `CN1Widgets` widget extension), used for both build types. Add `codename1.ios.debug.appext.<Name>.provision` / `codename1.ios.release.appext.<Name>.provision` to use the development profile for device/debug builds and the distribution profile for release builds; the variant matching the build target wins and the unqualified key is the fallback.

Paths may be absolute or relative to the project directory. The passwords must match the values you used when exporting the P12 files from Keychain Access.

Android

- `codename1.android.keystore`: Path to the Android keystore (`.ks` or `.keystore`).
- `codename1.android.keystorePassword`: Password protecting the keystore and key entry.
- `codename1.android.keystoreAlias`: Alias of the keypair used to sign the APK/AAB.

If the keystore fields are empty the Maven and Ant builders will create a default keystore, but production apps should commit the real paths and credentials so reproducible builds use your long-lived signing key.

76.4.2. iOS code signing failure checklist

Below is a list of common issues when singing and a set of suggestions for things to check. Notice that some of these signing failures will sometimes manifest themselves during build and sometimes will manifest during the installation of the application.



Most of these issues aren't applicable when using the wizard for example, a Mac isn't required for the Certificate Wizard as it uses the Codename One cloud

- **You must use a Mac to generate P12 certificates manually**. The workaround you found is the Certificate Wizard!

  Notice that this is something you need to do once a year (generate P12), you will also need a Mac to upload your final app to the store now.

- When exporting the P12 certificate **make sure that you selected BOTH the public and the private keys** as illustrated here. If you see one entry (no private key) then you created the CSR (singing request) on a different machine than the one where you imported the resulting CER file.

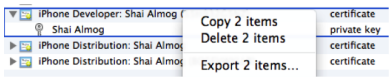


Figure 370. p12 export

- Launch Codename One Settings with `mvn cn1:settings` and make sure the **Package Name** in **Basic** matches the App ID used by the provisioning profile.
- Make sure the App ID prefix in the provisioning profile matches the prefix assigned by Apple. The standalone Certificate Wizard handles this automatically for profiles it creates.
- Make sure your provisioning profile's app ID matches your package name or is a * provisioning profile. Both are sampled in the pictures below, notice that you would need an actual package name for push/in-app purchase support as well as for app store distribution.

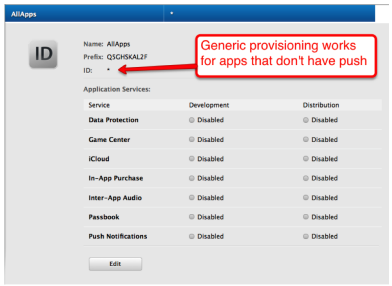


Figure 371. The star (*) Provisioning Profile

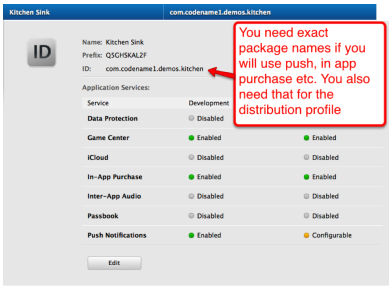


Figure 372. Provisioning Profile with app ID

- Make sure the certificate and provisioning profile are from the same source (if you work with multiple accounts), notice that provisioning profiles and certificates expire so you will need to regenerate provisioning when your certificate expires or is revoked.
- If you declare push in the provisioning profile then `ios.includePush` (in the build arguments) MUST be set to true, otherwise it **MUST** be set to false (see pictures below). Notice that this should be configured via the UI in the iOS section. The build server automatically enables the notification entitlement when your project uses `LocalNotification`, even if `ios.includePush` is false, so don't try to disable the entitlement manually to "fix" warning messages from App Store Connect.

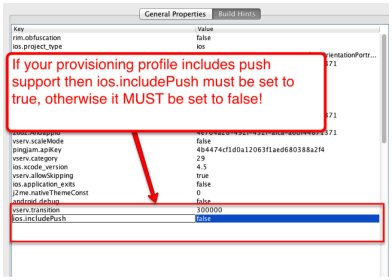


Figure 373. Include push build hint

76.5. Android

Signing Android applications is trivial when compared to the pain of iOS signing.

The NetBeans and Eclipse plugins have a simple wizard to generate the certificate that you can launch by pressing this button:

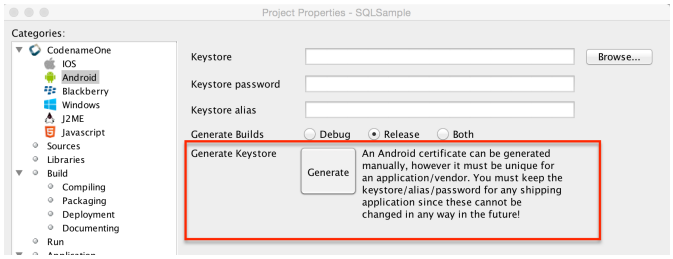


Figure 374. Android keystore generation wizard

Then fill out your details and password in the form:

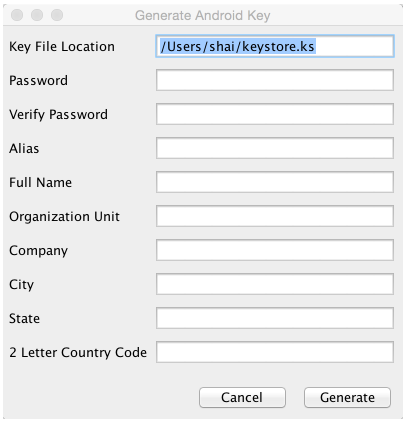


Figure 375. UI for Android certificate details

This will seamlessly generate a certificate for your project, you can reuse it for other projects as well.

76.5.1. Generating an Android certificate manually

You need the JDK's keytool executable (it should be under the JDK's bin directory) and execute the following command:

```
keytool -genkey -keystore Keystore.ks -alias [alias_name] -keyalg RSA -keysize 2048 -validity 15000 -dname "CN=[full
name], OU=[ou], O=[comp], L=[City], S=[State], C=[Country Code]" -storepass [password] -keypass [password]
```

The elements in the brackets should be filled up based on this:

```
Alias: [alias_name] (just use your name/company name without spaces)
Full name: [full name]
Organizational Unit: [ou]
Company: [comp]
City: [City]
State: [State]
CountryCode: [Country Code]
Password: [password] (we expect both passwords to be identical)
```

Executing the command produces a `Keystore.ks` file in that directory. Keep it safe: if you lose it, you can no longer publish upgrades signed with the same identity. Enter its path, alias, and passwords in `common/codenameone_settings.properties`, or use the **Android** page in the standalone Certificate Wizard (`mvn cn1:certificatewizard`) to generate and install a keystore.

For more details see http://developer.android.com/guide/publishing/app-signing.html

Chapter 77. Working with iOS

77.1. Troubleshooting iOS debug build installs

If you've access to a Mac, connect the device, open Xcode, and use the device explorer console to inspect messages that may explain what went wrong. If not, check the following:

- Make sure the build is the debug version and not the App Store version. The App Store version won't install on the device and can be distributed through Apple's store or TestFlight.
- Check that the UDID is correct. If you got the UDID from an app, it's probably wrong because apps no longer have access to the device UDID. Get it from the iOS Settings app or iTunes.
- Make sure the device isn't locked for installing third-party apps. This can happen on devices configured with parental controls.
- Check that you own the package name. For example, if you earlier installed an app with the same package name but a different certificate, a new install will fail. This is true for Android too. If you installed the kitchen sink from the store and then built your own app with the same package name, the installation will collide.
- This can be a problem if you use a generic package name that someone else already claimed, so use your own domain.
- Make sure the device has a recent enough version of iOS for the dependencies. As of 2024, Codename One builds target iOS 12.0 or newer by default. You can raise the need with the `ios.deployment_target` build hint if a library needs a newer version.
- Verify that you're using Safari when installing on the device. If you used a cable, that isn't a problem. Some developers had issues with Firefox not launching the install process.
- Check that the `ios.includePush` build hint matches your iOS provisioning. It must be false if your provisioning profile doesn't include push.

77.2. Launch screen storyboard best practices

Launch screen storyboards are the default approach for Codename One iOS builds. Apple requires a storyboard-based launch experience for modern devices, so the legacy screenshot generator was removed in favor of a single adaptive layout. You can still opt back into the old behavior by setting the `ios.generateSplashScreens=true` build hint, but it's best to use a storyboard unless your use case can't be expressed with Auto Layout.

77.2.1. Key files

The build server provides a minimal launch storyboard automatically. Customize it by adding any of the following files under your project's `native/ios` directory:

1. `Launch.Foreground.png` - Shown in the center of the screen instead of your app icon.
2. `Launch.Background.png` - Drawn behind the content to provide a color or illustration.
3. `LaunchScreen.storyboard` - A custom storyboard created in Xcode that replaces the default layout

entirely.



Make sure to add the `ios.multitasking=true` build hint or your launch storyboard won't be used.

77.2.2. Designing a flexible Layout

Keep the launch storyboard simple and static. The layout is rendered before your app code runs, so avoid views that depend on live data or animation. Follow these guidelines when editing `LaunchScreen.storyboard` in Xcode:

- Use Auto Layout constraints and safe-area guides so the design scales to every device, including split view on iPad.
- Prefer system colors or vector/PDF assets for logos so the result stays crisp on high-density screens and supports Dark Mode.
- Reserve text for short taglines or status messages that don't need localization at launch. Dynamic localization isn't available.
- Avoid referencing application delegate outlets or custom classes. Only design-time UIKit elements are supported.

77.2.3. Asset reference

The default storyboard expects PNG assets with the following characteristics. All sizes are specified in points (pt). Supply @2x and @3x variants for Retina displays when possible.

Table 17. Launch Screen Asset Reference

Asset	Purpose	Suggested 1x dimensions	Notes
`Launch.Foreground.png`	Brand mark centered on screen	152×152	Provide optional `Launch.Foreground@2x.png` (304×304) and `Launch.Foreground@3x.png` (456×456) for sharper output. Use transparency to let the background show through.
`Launch.Background.png`	Full-screen backdrop	1024×1024	Supply complementary `Launch.Background@2x.png` (2048×2048) and `Launch.Background@3x.png` (3072×3072) if you rely on artwork instead of a flat color. Keep file sizes small (<2 MB) to avoid slowing startup.

Asset	Purpose	Suggested 1x dimensions	Notes
`LaunchScreen.storyboard`	Complete custom layout	N/A	Target iOS 12.0 and later, enable Auto Layout, and include constraints for every view. Avoid timers or code connections.

77.2.4. Testing changes



Changes to the launch screen don't take effect until the device has been restarted. If you install your app on a device, then change the launch screen and update the app, the launch screen won't change until the device is restarted.

When iterating locally with a Mac, open the generated Xcode project and run it on a device or simulator to verify that the layout adapts. On Windows or Linux, submit a TestFlight or Ad-Hoc build and check on hardware before shipping.

77.3. Local notifications on iOS and Android

Local notifications are like push notifications, except that they're initiated locally by the app, rather than remotely. They're useful for communicating information to the user while the app is running in the background, since they manifest themselves as pop-up notifications on supported devices.



To set the notification icon on Android place a 24×24 icon named `ic_stat_notify.png` under the `native/android` folder of the app. The icon can be white with transparency areas

77.3.1. Sending notifications

The process for sending a notification is:

1. Create a LocalNotification [https://www.codenameone.com/javadoc/com/codename1/notifications/LocalNotification.html] object with the information you want to send in the notification.
2. Pass the object to `Display.scheduleLocalNotification()`.

Notifications can either be set up as one-time or as repeating.

Example sending notification



On iOS, scheduling a local notification with an `id` that already exists now replaces the earlier scheduled notification with that same `id` instead of keeping duplicates.

The resulting notification will look like

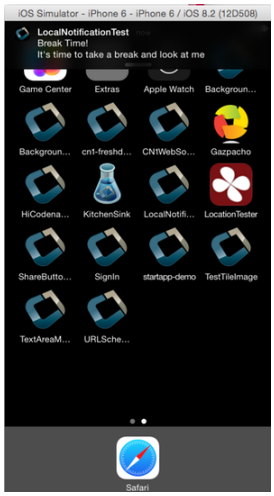


Figure 376. Resulting notification in iOS

77.3.2. Receiving notifications

The API for receiving/handling local notifications is also like push. Your application's main lifecycle class needs to implement the `com.codename1.notifications.LocalNotificationCallback` interface which includes a single method:

```
public void localNotificationReceived(String notificationId) {
}
```

The `notificationId` parameter will match the `id` value of the notification as set using `LocalNotification.setId()`.

Example receiving notification



`localNotificationReceived()` is called when the user responds to the notification by tapping on the alert. If the user doesn't click on the notification, then this event handler will never be fired.

77.3.3. Canceling notifications

Repeating notifications will continue until they're canceled by the app. You can cancel a single notification by calling:

Where `notificationId` is the string id that was set for the notification using `LocalNotification.setId()`.

77.4. iOS beta testing (testflight)

Apple provides the ability to distribute beta versions of your application to beta testers using testflight. This allows you to recruit up to 1000 beta testers without the typical UDID limits a typical Apple account has.



This is supported for pro users as part of the crash protection feature.

To take advantage of that capability use the build hint `ios.testFlight=true` and then submit the app to the store for beta testing. Make sure to use a release build target.

77.5. Metal renderer

iOS builds use the Metal rendering backend by default. The Metal pipeline drives a `CAMetalLayer`, replacing the legacy OpenGL ES 2 path on `CAEAGLLayer`. OpenGL ES is deprecated on iOS and Metal is the supported graphics API going forward, so new projects pick it up automatically.

To opt out and fall back to the OpenGL ES 2 backend, set the build hint:

```
codename1.arg.ios.metal=false
```

If you hit a rendering regression after the default flip, opting out with `ios.metal=false` is a safe escape hatch, but please file an issue with a reproducer so the Metal path can be fixed.

77.6. Choosing a color space for the Metal renderer

With the Metal renderer (the default), the `CAMetalLayer` is configured with the sRGB color space. This matches the behavior of the legacy OpenGL ES 2 backend on `CAEAGLLayer`: CG-rasterized images and gradients (which are tagged `DeviceRGB` in their `CGBitmapContext`) end up displayed with the same brightness on both backends.

For most apps the default is the right choice. Apps that need a different color profile — for example, wide-gamut artwork that should be displayed in Display P3, or a strictly device-RGB pipeline that matches a custom rendering toolchain — can override the choice with the `ios.metal.colorSpace` build hint:

Value	Effect
`sRGB` (default)	Sets `metalLayer.colorspace` to `kCGColorSpaceSRGB`. Use this unless you have a specific reason not to.
`displayP3`	Sets `metalLayer.colorspace` to `kCGColorSpaceDisplayP3`. Use for wide-gamut content on devices with a P3 display.
`deviceRGB`	Calls `CGColorSpaceCreateDeviceRGB()`. Skips sRGB gamma encoding — raw bytes are written to the layer.
`linearSRGB`	Sets `metalLayer.colorspace` to `kCGColorSpaceLinearSRGB` for linear (non-gamma-encoded) sRGB output.
`extendedSRGB`	Sets `metalLayer.colorspace` to `kCGColorSpaceExtendedSRGB` (extended-range sRGB, allows values outside [0,1]).
`extendedLinearSRGB`	Sets `metalLayer.colorspace` to `kCGColorSpaceExtendedLinearSRGB`.
`none`	Leaves `metalLayer.colorspace` unset, so Metal uses the system default for the device.



The hint only takes effect on the Metal backend (the default). If you have opted out

> with `ios.metal=false`, the layer is `CAEAGLLayer` and the color space is fixed by the system.

77.7. Accessing insecure URLs

Due to security exploits Apple blocked some access to insecure URLs which means that http code that worked before could stop working for you on iOS 9+. This is a good move, you should use https and avoid http as much as possible but that's sometimes impractical when working with an internal or debug environment.

You can disable the strict URL checks from Apple by using the venerable `ios.plistInject` build hint and setting it to:

```
<key>NSAppTransportSecurity</key><dict><key>NSAllowsArbitraryLoads</key><true/></dict>
```

For example, it seems that Apple will reject your app if you include that and don't have a good reason.

77.8. Using Cocoapods



> CocoaPods remains fully supported, but it's no longer the only supported iOS dependency path. Swift Package Manager (SPM) is also supported. The current guidance is documented in this section.

CocoaPods [https://cocoapods.org/] is a dependency manager for Swift and Objective-C Cocoa projects. It has over eighteen thousand libraries and can help you scale your projects. Cocoapods can be used in your Codename One project to include native iOS libraries without having to go through the hassle of bundling the actual library into your project. Rather than bundling `.h` and `.a` files in your ios/native directory, you can specify which "pods" your app uses through the `ios.pods` build hint. (Other build hints also exist if you need more advanced features.)

Examples

Include the AFNetworking [https://github.com/AFNetworking/AFNetworking] library in your app:

```
ios.pods=AFNetworking
```

Include the AFNetworking [https://github.com/AFNetworking/AFNetworking] version 3.0.x library in your app:

```
ios.pods=AFNetworking ~> 3.0
```

For full versioning syntax specifying pods see the Podfile spec for the "pod" directive [https://guides.cocoapods.org/syntax/podfile.html#pod].

77.8.1. Including multiple pods

Multiple pods can be separated by either commas or semi-colons in the value of the `ios.pods` build hint. For example: To include GoogleMaps and AFNetworking, you could:

```
ios.pods=GoogleMaps,AFNetworking
```

Or specifying versions:

```
ios.pods=AFNetworking ~> 3.0,GoogleMaps
```

77.8.2. Other pod related build hints

`ios.pods.platform` : The least platform to target. Sometimes, Cocoapods require functionality that isn't in older version of iOS. For example, the GoogleMaps pod requires iOS 7.0 or higher, so you would need to add the `ios.pods.platform=7.0` build hint.

`ios.pods.sources` : Some pods require that you specify a URL of the source of the pod spec. This may be optional if the spec is hosted in the central CocoaPods source (`https://github.com/CocoaPods/Specs.git`).

77.8.3. Converting PodFile to build hints

Most documentation for Cocoapods "pods" provide instructions on what you need to add to your Xcode project's PodFile. Here is an example from the GoogleMaps cocoapod to show you how a PodFile can be converted into equivalent build hints in a Codename One project.

The GoogleMaps cocoapod directs you to add the following to your PodFile:

```
source 'https://github.com/CocoaPods/Specs.git'
platform :ios, '7.0'
pod 'GoogleMaps'
```

This would translate to the following build hints in your Codename One project:

```
ios.pods.sources=https://github.com/CocoaPods/Specs.git
ios.pods.platform=7.0
ios.pods=GoogleMaps
```

(Note that the `ios.pods.sources` directive is optional).

77.9. Using Swift package manager

Swift Package Manager can be used as an alternative to CocoaPods for remote Swift package dependencies on iOS. Select the dependency path with `ios.dependencyManager`. Supported values are

`auto`, `CocoaPods`, `spm`, and `both`.

For an SPM-only configuration, declare the packages in `ios.spm.packages` using the format `<identity>|<url>|<requirement>` and then declare the products to link using `ios.spm.products.<identity>`.

```
ios.dependencyManager=spm
ios.spm.packages=swift-collections|https://github.com/apple/swift-collections.git|from:1.1.0
ios.spm.products.swift-collections=Collections
```

Supported need formats are `from:`, `exact:`, `branch:`, `revision:`, and `range:`.

`ios.dependencyManager=auto` preserves backward compatibility. Existing projects with `ios.pods` continue to use CocoaPods. Projects with `ios.spm.*` use SPM. If both hint families are present, both are applied.

Explicit dependency-manager modes are validated rather than dropping dependencies. `ios.dependencyManager=spm` fails when the project or a built-in API requires a CocoaPod; use `auto` or `both`. Likewise, `ios.dependencyManager=none` fails while pod or SPM hints are present; remove the dependency hints only if suppression is intentional, or switch to `auto`. This is a migration-visible change for older projects that used `spm` or `none` to ignore declared native dependencies.

77.10. Including dynamic frameworks

If you need to use a dynamic framework (for example: SomeThirdPartySDK.framework), and it isn't available through CocoaPods, then you can add it to your project by zipping up the framework and copying it to your native/ios directory.

For example: native/ios/SomeThirdPartySDK.framework.zip

No build hints necessary for this approach. The build server will automatically detect the framework and link it into your app.

Chapter 78. On-Device Debugging (iOS)

The Codename One simulator runs your app on the JVM, so the IDE's normal Java debugger works against it directly. Some bugs only appear on a real device — ParparVM's threading model, iOS-specific native behaviour, performance characteristics on real hardware, memory pressure under iOS background limits, and timing around UIKit interactions are common examples. On-device debugging lets you keep using a standard Java debugger (IntelliJ IDEA, jdb, VS Code, anything that speaks JDWP) against the running iOS app on either the native iOS simulator or a physical device.

It works by adding a small listener thread to the ParparVM-generated iOS binary. The app dials out to a desktop proxy over Wi-Fi or loopback; the proxy speaks JDWP on the other side, so to the IDE everything looks like attaching to a normal remote JVM.

78.1. When to use it

- You can reproduce a bug on a built iOS app but not in the Codename One simulator — for instance a native-call issue, a layout glitch that only shows up on iOS modern theme, or an AOT-vs.-JIT numerical discrepancy.
- You want to single-step through real ParparVM-translated code and read the values its variables actually hold at runtime.
- You want to inspect object state on a tethered device while reproducing a problem a customer reported, without having to add log lines and rebuild.

If your bug is reproducible in the simulator, stay in the simulator — its debugger is faster and has fewer limitations.

78.2. Quick start (IntelliJ IDEA)

Codename One projects generated from the `cn1app-archetype` ship with two ready-made IntelliJ run configurations under the **On-Device Debug** folder in the run-config dropdown: **CN1 Debug Proxy** and **CN1 Attach iOS**. The rest of this section assumes you're using them. Skip to Using jdb instead if you prefer the command-line debugger.

78.2.1. 1. Enable the build hints

In `common/codenameone_settings.properties`, uncomment the four lines the archetype generated:

```
codename1.arg.ios.onDeviceDebug=true
codename1.arg.ios.onDeviceDebug.proxyHost=127.0.0.1
codename1.arg.ios.onDeviceDebug.proxyPort=55333
# Optional: block the app at startup until the IDE attaches.
codename1.arg.ios.onDeviceDebug.waitForAttach=true
```

The build hint names start at `ios.onDeviceDebug`. Entries in `codenameone_settings.properties` use the `codename1.arg.` prefix, as shown above.

For a physical device, set `proxyHost` to the laptop's LAN IP (run `ipconfig` on Windows or `ifconfig` on macOS and Linux) instead of `127.0.0.1`. The native iOS simulator can always use `127.0.0.1`.

78.2.2. 2. Build the app as you normally do

Either the local Xcode path (`cn1:buildIosXcodeProject`) or a cloud build (`cn1:buildIosOnDeviceDebug`) will produce an app instrumented for on-device debugging once the build hints are set.

78.2.3. 3. Run the proxy from IntelliJ

Select **CN1 Debug Proxy** from the Run-config dropdown and click **Run** (the green play icon — not the bug icon; **Debug** on this config would attach IntelliJ's own debugger to the proxy process, which is not what you want). The Run tool window opens and shows:

```
On-device-debug proxy starting:
  device  : listening on tcp://0.0.0.0:55333
  jdwp    : listening on tcp://0.0.0.0:8000
  symbols : streamed from the device on connect (no local file)
[device] listening on port 55333 for ParparVM app to dial in
[jdwp] listening on port 8000 for debugger (jdb) to attach
```

When both listener lines appear, the proxy is ready for the app.

78.2.4. 4. Launch the app and wait for the device handshake

Launch the iOS app in the native iOS simulator under Xcode or on a physical device. The app contains its compressed debug symbol table and streams it to the proxy when it connects. Wait for these lines before attaching the IDE:

```
[device] connected from ...
[device] loaded symbols: ...
[jdwp] device handshake ...
```

With `waitForAttach=true`, the app remains on the "Waiting for debugger" overlay while you complete the next step.

Attaching the IDE before launching the app is also supported. The proxy keeps the JDWP session open and waits for the device to stream its symbols. If the device still hasn't connected after ten seconds, the proxy prints a checklist for the build hint, `proxyHost`, and the device port. You can correct the problem and launch the app without restarting the IDE session; rebuild and reinstall first if you change a build hint.

78.2.5. 5. Attach the debugger from IntelliJ

Switch the Run-config dropdown to **CN1 Attach iOS** and click **Debug**. IntelliJ connects to `localhost:8000`, opens a Debug tool window, and releases the app after the debugger connection completes.

Set breakpoints, step, inspect, evaluate expressions, just like a normal remote attach. Device output from `System.out.println` / `Log.p` / `printf` / `NSLog` lands in the **CN1 Debug Proxy** Run window prefixed with `[device]`.

78.3. Attach from NetBeans

NetBeans uses the same proxy and connection order. Start the proxy, launch the app, and wait for the device handshake and symbol-loading lines shown above. Then choose **Debug** > **Attach Debugger** in NetBeans. Select **Java Debugger (JPDA)** and the **SocketAttach** connector, set the host to `localhost` and the port to `8000`, then attach. Keep the Codename One project open so NetBeans can resolve your source files and breakpoints.

Don't point NetBeans at port `55333`. The iPhone uses `55333` to reach the proxy. The IDE uses the proxy's JDWP port, `8000`.

78.4. Quick start (Maven from the command line)

If you don't want to use the IntelliJ run configs, the same flow is two terminals:

```
# Terminal 1 — start the proxy
mvn cn1:ios-on-device-debugging

# Launch the app and wait for the proxy to report that it loaded symbols.

# Terminal 2 — attach jdb after the device handshake
jdb -attach localhost:8000
```

For VS Code (Debugger for Java extension), add a launch configuration of type `java` with `"request": "attach"`, `"hostName": "localhost"`, `"port": 8000`.

78.4.1. Using jdb instead

`jdb` is a minimal Java debugger that ships with every JDK and works fine over the same JDWP socket. Useful when you don't have an IDE available, or for scripted regression checks. Source path resolution is manual:

```
jdb -attach localhost:8000 \
    -sourcepath src/main/java:$HOME/.m2/repository/com/codenameone/codenameone-core/8.0-SNAPSHOT/codenameone-core-8.0-SNAPSHOT-sources.jar
```

The `-sourcepath` is required for `list` / `where` to render framework source — jdb won't unzip the `-sources.jar` automatically; if you prefer plain files, extract it once and point at the directory.

78.5. What works today

- Class loading: the IDE sees every class in your build.

- Line breakpoints in user code **and** in the Codename One framework classes (`com.codename1.ui.`, `com.codename1.io.`, etc.).
- Step into / over / out.
- Stack walking — full Java stack including the Codename One framework frames above your code.
- Inspecting primitive locals (int, long, float, double, boolean, byte, char, short).
- Inspecting `java.lang.String` values directly.
- Inspecting object references — class name and identity, plus field-by-field drilldown via the IDE's variables view. The proxy walks the ParparVM struct layout to read instance fields directly at known offsets, so `dump this` / "Variables" shows the same fields you'd see in a simulator debug session.
- Array inspection — `Object[]`, `int[]`, `byte[]`, etc. report their length and per-index values to the IDE so expanding e.g. `ArrayList.elementData` shows the actual element references rather than an opaque reference. Primitive element types round-trip with their JVM type tag.
- Method invocation — the IDE's "Evaluate Expression" / `print` command can call instance and static methods on objects in scope. The classic chain `Display.getInstance().getCurrent().getTitle()` evaluates to the current form's title string, so `myForm.getComponentCount()` and similar accessor calls work too. The call runs on the suspended Java thread inside a catch-all try block, so uncaught throws round-trip back as an exception reference rather than tearing down the debugger.
- Native device output — `System.out.println`, `Log.p`, and `printf` / `NSLog` from native code are surfaced in the proxy's console (the IntelliJ **CN1 Debug Proxy** Run window when launched via the run config) prefixed with `[device]`.
- Pause and resume from the IDE.

78.6. Pointing the IDE at the Codename One sources

The proxy answers `Method.LineTable` and `ReferenceType.SourceFile` for framework classes, but the IDE still needs to find the actual `.java` files to render the source pane while stepping. Three options in order of convenience:

- **Maven sources jar (recommended).** The `codenameone-core-<version>-sources.jar` is published alongside the regular `codenameone-core` artifact. IntelliJ resolves it automatically once you run **Maven → Reimport** with "Sources" enabled in the Maven settings. Other IDEs work the same way as long as their Maven integration is set to download sources.
- **Local clone of the Codename One repository.** If you've already checked out the CodenameOne GitHub repository [https://github.com/codenameone/CodenameOne] (for instance to follow the contributor workflow), point the IDE at its `CodenameOne/src` directory. In IntelliJ: **Run → Edit Configurations… → CN1 Attach iOS → Configuration → Source roots → +**.
- **jdb command line.** Pass the framework source directory on the `-sourcepath`, as shown in Using jdb instead.

Without a source path, breakpoints in framework classes still trigger and locals / fields still read — you'll just see "Sources not found" in the IDE editor when stepping into framework code.

78.7. Known limitations

- **Method invocation is partial.** The debugger can call instance and static methods on Codename One framework classes and on user code: jdb's `print myForm.getTitle()` and IntelliJ's "Evaluate Expression" pop-up work for the chains you actually want (`Display.getInstance().getCurrent().getTitle()` is the canonical test). Methods on `java.io.`, `java.net.`, `java.nio.`, and `com.codename1.impl.` are unsupported — those classes are linked against hand-written native code that has fallen out of sync with the generic invoke-thunk calling convention, so the translator skips them. The same limit applies to watch expressions.
- **Constructor invocation** isn't supported. The debugger can call existing instance methods but not `new Foo(…)`.
- **Hot-swap** isn't supported. Code changes require a rebuild and reinstall of the app.
- **Hot threads** — if your app is doing heavy work on a non-EDT thread when a breakpoint hits, that other thread continues running. Only the thread that hit the breakpoint suspends by default. Use the IDE's "suspend VM" if you need a fully frozen picture.
- **Local variable names** are present only when the user's Java sources were compiled with `-g`. Codename One's archetypes do compile with debug info by default; if a variable shows up as `v1`, `v2`, … that's because that particular class was compiled without debug info.
- **Static field reads** are unsupported. Instance field reads are fully wired; static-field support requires an additional per-class static table that hasn't been emitted yet.
- **NSLog on a real device** (as opposed to the native iOS simulator) may not reach the `[device]` console. NSLog on iOS routes to `os_log` which doesn't always mirror to `stderr`. `printf` / `fprintf` / `Log.p` are unaffected and show up correctly.

78.8. Performance

The on-device-debug instrumentation is gated by a compile-time flag that's **only set in debug builds**. Release builds are completely unaffected — no listener thread, no per-line callback, no extra metadata in the binary.

In debug builds, the per-line callback adds a predictable load+branch around the macro that ParparVM already emits to record source line numbers for stack traces. When no debugger is attached, the runtime cost is close to zero. When a debugger **is** attached, expect numerical inner loops to run on the order of two to three times slower than a release build — this is normal and matches the overhead of `-g`-style debugging on other native VMs.

78.9. Debugging the generated Xcode project natively

The JDWP flow above keeps you at the Java level. For native-level work — stepping through Objective-C sources, debugging a native interface, chasing a crash inside the VM, or profiling with Instruments — debug the generated Xcode project directly:

- **Local build**: `mvn cn1:buildIosXcodeProject` writes the complete Xcode project under `target/…-ios-source/`.
- **Cloud build**: run `mvn cn1:settings`, enable **Include Source** on the **Basic** page, save, then send

the build and download the sources result from the build server.

Open the `.xcworkspace` file if one was generated, otherwise the `.xcodeproj`, and run on a device or the native iOS simulator from Xcode as you would any iOS app.



The `.xcworkspace` is no longer exclusive to CocoaPods-based builds. Use it whenever it's generated, whether the project uses CocoaPods, Swift Package Manager, or both.

Inside Xcode you get LLDB and Instruments against the real binary. Your Java code appears as the C sources ParparVM generated from it — a Java method like `com.mycompany.MyClass.doStuff(int)` becomes a C function named along the lines of `com_mycompany_MyClass_doStuff___int`, so symbolic breakpoints on the translated code work too. The native-IDE path and the JDWP path are complementary: use Xcode for native frames and profiling, the JDWP proxy for source-level Java debugging.

78.10. Troubleshooting

78.10.1. The proxy reports "device disconnected" before the breakpoint fires

The app might be crashing before it reaches user code. Run the app from Xcode and watch the console — `installSignalHandlers` will intercept native crashes and rethrow them as Java exceptions you can see. If the connection drops mid-debug, check the proxy log for a device-side I/O error.

78.10.2. "Connection refused" when the device tries to reach the proxy

The device's `ios.onDeviceDebug.proxyHost` value is wrong, or a firewall is blocking the port. Verify:

- The native iOS simulator can always reach `127.0.0.1`.
- A physical device needs the laptop's **LAN IP** (not `localhost`), and the laptop's firewall must allow incoming connections on the configured port (default `55333`).
- macOS will prompt the first time the proxy listens — accept the **Allow incoming connections** dialog.

78.10.3. jdb reports "Internal exception: Unexpected JDWP Error: 100"

This means an unsupported JDWP command was invoked. The session itself survives; the most common cause is the IDE asking for something the proxy explicitly stubs (see "Known limitations" above). If you hit a less obvious case, the proxy logs the unknown command code — file an issue with that code attached.

78.10.4. The app launches but the breakpoint never fires

Confirm the build hint actually took effect. Inspect the generated Xcode project's `cn1_globals.h` and look for `#define CN1_ON_DEVICE_DEBUG` **without** a leading `//`. If the line is still commented out, the build was a release/non-debug build, or `ios.onDeviceDebug` wasn't set as a build hint.

78.10.5. Make the app wait while the proxy and IDE attach

Set `ios.onDeviceDebug.waitForAttach=true`. The app will block at startup until the proxy is connected and the IDE tells the VM to continue.

Chapter 79. On-Device Debugging (Android)

The companion to the iOS chapter, for the Android side. The motivation is the same — bugs that only reproduce on a real device or on the Android emulator — but the implementation is much simpler. The Dalvik/ART runtime already exposes a JDWP socket per debuggable process, so there is no desktop proxy and no custom protocol. The Codename One Maven plugin just orchestrates `adb`: install the APK, mark it as the debug-app, launch the activity, forward the JDWP socket onto `localhost`, and tail logcat.

Everything below assumes you have the Android platform-tools on your machine (Android Studio installs them, or the standalone package [https://developer.android.com/tools/releases/platform-tools]) and `adb` is reachable through `ANDROID_HOME`, `ANDROID_SDK_ROOT`, or `$PATH`.

79.1. When to use it

- A bug only reproduces on a physical Android device or in the Android emulator and not in the Codename One simulator.
- You want to single-step through your code while it runs on the device, with full access to locals, fields, and the `Display.getInstance()⋯` accessor chains.
- You want to attach without a USB cable — the wireless flow below covers both the legacy `adb tcpip` path and the Android 11+ `adb pair` path.

If the bug reproduces in the simulator, stay in the simulator — its debugger is faster and has zero device-side moving parts.

79.2. Quick start (IntelliJ IDEA)

Projects generated from the `cn1app-archetype` ship with two run configurations under the **On-Device Debug** folder: **CN1 Android On-Device Debug** and **CN1 Attach Android**. The flow is:

79.2.1. 1. Enable the build hint

In `common/codenameone_settings.properties`, uncomment the line the archetype generated:

```
codename1.arg.android.onDeviceDebug=true
```

The build hint name is `android.onDeviceDebug`. Entries in `codenameone_settings.properties` use the `codename1.arg.` prefix, as shown above.

This flips the generated `AndroidManifest.xml` to `debuggable="true"` and disables R8/proguard so symbols and locals survive the build. Release builds (anything without this hint) are unaffected.

79.2.2. 2. Build the APK

Either the cloud build (`cn1:buildAndroidOnDeviceDebug` — wraps `cn1:buildAndroid` and force-sets the hint above) or your usual `cn1:buildAndroid` once the hint is set. Both produce a signed, debuggable

APK in the project's `target/` directory.

For a fully local build, use `cn1:buildAndroidGradleProject` to generate the Gradle project under `target/…-android-source/` and run `./gradlew assembleDebug` from there; the Mojo will autodetect the resulting APK under `build/outputs/apk/`.

79.2.3. 3. Connect the device

Plug a device in over USB with USB debugging enabled, **or** connect wirelessly (see Wireless debugging below).

79.2.4. 4. Run the debug session from IntelliJ

Select **CN1 Android On-Device Debug** from the Run-config dropdown and click **Run** (the green play icon — not the bug icon). The Run tool window prints:

```
Using adb: /Users/you/Library/Android/sdk/platform-tools/adb
Target device: emulator-5554
Installing my-app-1.0.apk on emulator-5554
Marking com.example.myapp as the debug app (waits for debugger).
Launching com.example.myapp.MyAppStub
App PID on device: 12345

================================================================
 JDWP forwarded:  localhost:5005  ->  device pid 12345
 Attach IntelliJ: Run -> 'CN1 Attach Android' (Remote JVM Debug)
================================================================
```

After that banner, logcat output (filtered to the app's PID) streams through the Run window prefixed with `[device]`.

79.2.5. 5. Attach the debugger

Switch the Run-config dropdown to **CN1 Attach Android** and click **Debug**. IntelliJ connects to `localhost:5005` and opens a Debug tool window. The app is paused inside `Debug.waitForDebugger()` at this point — set your breakpoints, then resume from the IDE to let it proceed past boot.

Set breakpoints, step, inspect, evaluate expressions — all the normal remote-attach features work. Unlike the iOS path, there is no method-invocation or static-field limitation here; the JVM is the real Android runtime.

79.3. Wireless debugging

79.3.1. Android 11 and newer (recommended)

1. On the device: **Settings → Developer options → Wireless debugging → Pair device with pairing code**. Note the **IP & port** (for example `192.168.1.42:37051`) and the **six-digit pairing code**.

2. Pair from your laptop (this only has to be done once per network):

```
adb pair 192.168.1.42:37051
# When prompted, enter the 6-digit code shown on the device.
```

3. Connect:

```
adb connect 192.168.1.42:5555
```

   The connect port is **not** the pairing port — it's the one shown on the **Wireless debugging** screen above the **Pair device** button.

4. Run the debug session as normal. Pass the IP and port through the Mojo if you prefer to do the `adb connect` in one step:

```
mvn cn1:android-on-device-debugging \
    -Dcn1.android.onDeviceDebug.wireless=192.168.1.42:5555
```

79.3.2. Android 10 and older

1. Plug the device in over USB.
2. Switch adb to TCP/IP mode and grab the device's IP from **Settings** → **About phone** → **Status**:

```
adb tcpip 5555
adb connect 192.168.1.42:5555
```

3. Unplug the cable, then run the debug session as normal.

In either flow, the JDWP forward, app launch, and logcat stream all happen over the same Wi-Fi link.

79.4. Quick start (Maven from the command line)

Without the IntelliJ run configs, the same flow is two terminals:

```
# Terminal 1 — build the APK once
mvn cn1:buildAndroidOnDeviceDebug

# Terminal 2 — install, launch, forward JDWP, tail logcat
mvn cn1:android-on-device-debugging
```

Attach jdb (or another JDWP client) to the forwarded port:

```
jdb -attach localhost:5005 \
    -sourcepath src/main/java:$HOME/.m2/repository/com/codenameone/codenameone-core/8.0-SNAPSHOT/codenameone-core-8.0-
SNAPSHOT-sources.jar
```

For VS Code (Debugger for Java extension), add a launch configuration of type `java` with `"request": "attach"`, `"hostName": "localhost"`, `"port": 5005`.

79.5. Useful command-line flags

The Mojo's defaults match what the archetype's run config does. The flags below exist for unusual setups:

Property	Meaning
`-Dcn1.android.onDeviceDebug.adb=<path>`	Use a specific adb executable (default: search `ANDROID_HOME`, `ANDROID_SDK_ROOT`, the standard Android Studio SDK locations, and `$PATH`).
`-Dcn1.android.onDeviceDebug.deviceSerial=<serial>`	Force a target device when more than one is online (`adb devices`).
`-Dcn1.android.onDeviceDebug.wireless=<ip:port>`	Run `adb connect <ip:port>` before everything else (covers both the Android 11 wireless-debug path and the legacy `adb tcpip` path).
`-Dcn1.android.onDeviceDebug.jdwpPort=<port>`	Local TCP port for `adb forward`. Default `5005`. Match this with the **CN1 Attach Android** run config if you change it.
`-Dcn1.android.onDeviceDebug.apk=<path>`	Skip APK autodetection and install this APK instead.
`-Dcn1.android.onDeviceDebug.skipInstall=true`	Skip the install step — the app is already on the device.
`-Dcn1.android.onDeviceDebug.waitForAttach=false`	Don't run `am set-debug-app -w`. The app launches normally and you attach the debugger afterwards. Default is `true`.

79.6. What you can step through

Everything runs in one Dalvik/ART process, so the JDWP attach sees every class loaded by the app. In practical terms:

- **Your common-module Java** (`com.<yourcompany>.<app>.`, anything under the `-common` module's `src/main/java`). Breakpoints work out of the box — this is the module the *CN1 Attach Android* run config is scoped to, so IntelliJ resolves the source pane and the variables view from your common-module classpath.
- **Your android-module Java or Kotlin** (`src/main/java` of the `-android` module — native interface implementations, custom Activities, Android-only helpers). Open the file and set a breakpoint; the IDE's **Remote JVM Debug** config doesn't restrict which module breakpoints live in, the `<module>` setting only decides which module's classpath is used to **display** state. If the variables view ever looks empty when you stop inside `-android` code, switch the run config's **Use classpath of module** to the `-android` module and reattach.

- **Codename One framework code** in `codenameone-core` and the Android port (`com.codename1.impl.android.*` such as `AndroidImplementation` and `CodenameOneActivity`). Both need source resolution — see the next section.
- *Native C/C via the NDK* is *not* debuggable through this path. JDWP only speaks JVM. If you've added native sources through the Android NDK, attach Android Studio's LLDB to the same process for C/C debugging — the two attaches are independent and can run side-by-side against one device.

79.7. Pointing the IDE at the Codename One sources

The Android runtime serves real `.class` files, so the IDE only needs the matching `.java` files to render the source pane. The same two options cover both `codenameone-core` and the Android port (`codenameone-android`):

- **Maven sources jars (recommended).** IntelliJ resolves `codenameone-core-<version>-sources.jar` **and** `codenameone-android-<version>-sources.jar` automatically once you run **Maven → Reimport** with "Sources" enabled in the Maven settings.
- **Local clone of the Codename One GitHub repository [https://github.com/codenameone/CodenameOne].** In IntelliJ: **Run → Edit Configurations… → CN1 Attach Android → Configuration → Source roots → +**. Add two entries: the clone's `CodenameOne/src` directory for the framework core, and `Ports/Android/src` for `AndroidImplementation` and the rest of the Android port.

Without a source path, breakpoints in framework classes still trigger and locals / fields still read — you'll just see "Sources not found" in the editor when stepping into framework code.

79.8. Debugging the generated Gradle project from Android Studio

Everything above drives a Codename One-built APK through `adb`. The alternative is to open the generated Android project itself in Android Studio and use it like any native Android app — useful when you're debugging native interface code, working on the Android port, or want Android Studio's profiler and layout inspector:

- **Local build**: `mvn cn1:buildAndroidGradleProject` writes a complete Gradle project under `target/⋯-android-source/`. Open that directory in Android Studio (current versions open it directly — no Gradle version fiddling required), connect a device, and press **Debug**.
- **Cloud build**: check the **Include Source** flag in Codename One Settings before sending the build, then download the sources result from the build server and open it the same way.

When iterating, regenerate the project rather than hand-merging files — the generated project is a build artifact, not a source of truth.

For native C/C++ added through the NDK, attach Android Studio's LLDB to the process; as noted in Section 79.6, "What you can step through", the LLDB and JDWP attaches are independent and can run side by side against one device.

79.9. Troubleshooting

79.9.1. "No Android device is online"

`adb devices` returns nothing useful. Common causes:

- The device's **USB debugging** toggle is off, or the per-laptop RSA fingerprint prompt is still pending on-device.
- The cable is power-only (some short USB-C cables are charge-only).
- For wireless: pairing expired (Android 11+) or the laptop is on a different Wi-Fi network.

79.9.2. "Multiple devices online"

Pass `-Dcn1.android.onDeviceDebug.deviceSerial=<serial>` (run `adb devices` to see serials). The emulator's serial looks like `emulator-5554`; a USB-attached phone is its hardware ID; a wireless device is `<ip>:<port>`.

79.9.3. App installs but won't pause for the debugger

`waitForAttach` only takes effect when the APK is built with `android.onDeviceDebug=true` (or `android.xapplication_attr` containing `android:debuggable="true"`). Verify by running `adb shell dumpsys package <your.package> | grep flags` — `DEBUGGABLE` must be in the list. If it isn't, rebuild with `mvn cn1:buildAndroidOnDeviceDebug`.

79.9.4. Breakpoint never fires

Check the **CN1 Attach Android** run config:

- **Host** is `localhost`, **Port** matches the `jdwpPort` printed by the debug session.
- The **Use module classpath** dropdown is the `-common` module, so IntelliJ can resolve your `.java` files.

If a class loaded on the device doesn't match the class IntelliJ thinks is current, breakpoints stop firing with no error message. Rebuild and reinstall (`cn1:buildAndroidOnDeviceDebug` + re-run **CN1 Android On-Device Debug**).

79.9.5. logcat shows the app exiting with `Debug.waitForDebugger`-like noise

The system killed the process for taking too long to attach. Set `-Dcn1.android.onDeviceDebug.waitForAttach=false` and trigger the code path you want to debug manually after attach — Android's debug-app wait isn't bound to the breakpoint, only to process start.

79.9.6. Wireless connection drops mid-session

Wi-Fi connections to debug-mode devices are sensitive to power-saving. On the device, keep the screen on (or set **Settings → Developer options → Stay awake** while charging). For long sessions,

plug into USB and use `adb -s <wireless-serial>` to keep the wireless TCP socket alive without depending on the radio.

Chapter 80. Working with JavaScript

This section covers the Codename One JavaScript port, which allows you to compile your app as native JavaScript and run it inside a browser. This is different from the BrowserComponent [https://www.codenameone.com/javadoc/com/codename1/ui/BrowserComponent.html] and other methods of displaying HTML/JavaScript inside a Codename One app.

80.1. Choosing the JavaScript compiler

JavaScript builds use the ParparVM-based JavaScript port by default. The same builder is available to all Codename One accounts and is also used by the local `local-javascript` Maven target.

Cloud builds retain the original TeaVM port as a public fallback. Select the compiler with the `javascript.port` build hint:

Value	Behavior
`parparvm`	Use the current ParparVM/JavaScriptPort builder. This is the default.
`teavm`	Use the legacy TeaVM builder.

For example, add `javascript.port=teavm` to `codenameone_settings.properties` to diagnose a ParparVM-specific regression. An unknown value fails the build instead of selecting a different compiler.

80.2. Troubleshooting build errors

If your JavaScript build fails, download the error log and inspect the first compiler error. The following error applies specifically to the fallback TeaVM builder:

"Method XXX wasn't found"

TeaVM uses its own Java runtime library. It's complete, but you may occasionally run into methods that haven't been implemented. If you run into errors saying that certain classes or methods weren't found, please post them to the Codename One issue tracker [https://github.com/codenameone/CodenameOne/issues]. You can also work around these by changing your own code to not use such functions. If this missing method doesn't fall on a critical path on your app, you can also force the app to still build despite this error by adding the `javascript.stopOnErrors=false` build hint.

80.3. ZIP, WAR, or preview: What's the difference

The default ParparVM builder produces a static browser ZIP and, unless proxy packaging is disabled or an external proxy is configured, a deployable proxy bundle. The deployable bundle is a Jakarta Servlet WAR by default. Use `javascript.proxy.target` to select another deployment platform as described in Section 80.4, "Setting up a proxy for network requests".

The legacy TeaVM builder can result in up to three different bundles:

1. YourApp-1.0.war

2. YourApp-1.0.zip
3. YourApp-1.0-Preview.html

YourApp-1.0.war is the legacy TeaVM self-contained application bundle. It can be installed in a Java EE servlet container and includes the old `javax.servlet` proxy implementation. New ParparVM builds use the current proxy profiles below and default to Jakarta Servlet.

As an example, the `PropertyCross.war` file contains the following files:

```
$ jar tvf PropertyCross-1.0.war
     0 Thu Apr 30 15:57:38 PDT 2015 META-INF/
   132 Thu Apr 30 15:57:36 PDT 2015 META-INF/MANIFEST.MF
     0 Thu Apr 30 15:57:36 PDT 2015 assets/
     0 Thu Apr 30 15:57:36 PDT 2015 assets/META-INF/
     0 Thu Apr 30 15:57:36 PDT 2015 js/
     0 Thu Apr 30 15:57:36 PDT 2015 teavm/
     0 Thu Apr 30 15:57:36 PDT 2015 WEB-INF/
     0 Thu Apr 30 15:57:36 PDT 2015 WEB-INF/classes/
     0 Thu Apr 30 15:57:36 PDT 2015 WEB-INF/classes/com/
     0 Thu Apr 30 15:57:36 PDT 2015 WEB-INF/classes/com/codename1/
     0 Thu Apr 30 15:57:36 PDT 2015 WEB-INF/classes/com/codename1/corsproxy/
     0 Thu Apr 30 15:57:36 PDT 2015 WEB-INF/lib/
 27568 Thu Apr 30 15:57:12 PDT 2015 assets/CN1Resource.res
306312 Thu Apr 30 15:57:12 PDT 2015 assets/iOS7Theme.res
427737 Thu Apr 30 15:57:12 PDT 2015 assets/iPhoneTheme.res
   350 Thu Apr 30 15:57:12 PDT 2015 assets/META-INF/MANIFEST.MF
 92671 Thu Apr 30 15:57:12 PDT 2015 assets/theme.res
 23549 Thu Apr 30 15:57:14 PDT 2015 icon.png
  2976 Thu Apr 30 15:57:14 PDT 2015 index.html
 30695 Thu Apr 30 15:57:12 PDT 2015 js/fontmetrics.js
 84319 Thu Apr 30 15:57:12 PDT 2015 js/jquery.min.js
 13261 Thu Apr 30 15:57:12 PDT 2015 progress.gif
  2816 Thu Apr 30 15:57:12 PDT 2015 style.css
1886163 Thu Apr 30 15:57:36 PDT 2015 teavm/classes.js
359150 Thu Apr 30 15:57:36 PDT 2015 teavm/classes.js.map
1147502 Thu Apr 30 15:57:36 PDT 2015 teavm/classes.js.teavmdbg
 30325 Thu Apr 30 15:57:36 PDT 2015 teavm/runtime.js
  1011 Thu Apr 30 15:57:18 PDT 2015 WEB-INF/classes/com/codename1/corsproxy/CORSProxy.class
232771 Wed Nov 05 17:35:12 PST 2014 WEB-INF/lib/commons-codec-1.6.jar
 62050 Wed Apr 15 14:35:56 PDT 2015 WEB-INF/lib/commons-logging-1.1.3.jar
590004 Wed Apr 15 14:35:58 PDT 2015 WEB-INF/lib/httpclient-4.3.4.jar
282269 Wed Apr 15 14:35:56 PDT 2015 WEB-INF/lib/httpcore-4.3.2.jar
 14527 Wed Apr 15 14:35:56 PDT 2015 WEB-INF/lib/smiley-http-proxy-servlet-1.6.jar
   903 Thu Apr 30 15:57:12 PDT 2015 WEB-INF/web.xml
  9458 Thu Apr 30 15:57:14 PDT 2015 META-INF/maven/com.propertycross/PropertyCross/pom.xml
   113 Thu Apr 30 15:57:36 PDT 2015 META-INF/maven/com.propertycross/PropertyCross/pom.properties
```

Some things to note in this file listing:

1. The **index.html** file is the entry point to the application.
2. **CORSProxy.class** is the proxy servlet for making network requests to other domains.
3. The **assets** directory contains all your application's **jar** resources. All resource files in your app

will end up in this directory.

4. The `teavm` directory contains all the generated JavaScript for your application. Notice that there are some debugging files generated (**classes.js.map** and **classes.js.teavmdbg**). These aren't loaded by the browser when your app is run, but they can be used by Chrome when you're doing debugging.
5. The **jar** files in the **WEB-INF/lib** directory are dependencies of the proxy servlet. They aren't required for your app to run - unless you're using the proxy.

YourApp-1.0.zip is appropriate for deploying the application on any web server. It contains all the same files as the `.war` file, excluding the WEB-INF directory (that is, it doesn't include any servlets, class files, or Java libraries - it contains purely client-side JavaScript files and HTML).

As an example, this is a listing of the files in the zip distribution of the PropertyCross demo:

```
$ unzip -vl PropertyCross-1.0.zip
Archive:  /path/to/PropertyCross-1.0.zip
 Length   Method    Size  Ratio   Date   Time   CRC-32    Name
--------  ------  ------- -----   ----   ----   ------    ----
   27568  Defl:N    26583   4%  04-30-15 15:57  9dc91739  assets/CN1Resource.res
  306312  Defl:N   125797  59%  04-30-15 15:57  0b5c1c3a  assets/iOS7Theme.res
  427737  Defl:N   218975  49%  04-30-15 15:57  3de499c8  assets/iPhoneTheme.res
     350  Defl:N      241  31%  04-30-15 15:57  7e7e3714  assets/META-INF/MANIFEST.MF
   92671  Defl:N    91829   1%  04-30-15 15:57  004ad9d7  assets/theme.res
   23549  Defl:N    23452   0%  04-30-15 15:57  acd79066  icon.png
    2903  Defl:N     1149  60%  04-30-15 15:57  e5341de1  index.html
   30695  Defl:N     7937  74%  04-30-15 15:57  2e008f6c  js/fontmetrics.js
   84319  Defl:N    29541  65%  04-30-15 15:57  15b91689  js/jquery.min.js
   13261  Defl:N    11944  10%  04-30-15 15:57  51b895c7  progress.gif
    2816  Defl:N      653  77%  04-30-15 15:57  a12159c7  style.css
 1886163  Defl:N   315437  83%  04-30-15 15:57  2b34c50f  teavm/classes.js
  359150  Defl:N    92874  74%  04-30-15 15:57  30abdf13  teavm/classes.js.map
 1147502  Defl:N   470472  59%  04-30-15 15:57  e5c456f7  teavm/classes.js.teavmdbg
   30325  Defl:N     5859  81%  04-30-15 15:57  46651f06  teavm/runtime.js
--------          -------  ---                            -------
 4435321          1422743  68%                            15 files
```

You'll notice that it has many of the same files as the `.war` distribution. It's missing the proxy servlet and dependencies.

YourApp-1.0-Preview.html is a single-page HTML file with all the application's resources embedded into a single page. This is generated for convenience so that you can preview your application on the build server directly. While you could use this file in production, you're probably better to use the ZIP or WAR distribution instead as some mobile devices have file size limitations that may cause problems for the "one large single file" approach. If you do decide to use this file for your production app (that is, copy the file to your own web server), you will need to change the proxy settings, as it's configured to use the proxy on the Codename One build server - which won't be available when the app is hosted on a different server.

80.4. Setting up a proxy for network requests

The Codename One API includes a network layer (the NetworkManager [https://www.codenameone.com/javadoc/com/codename1/io/NetworkManager.html] and ConnectionRequest [https://www.codenameone.com/javadoc/com/codename1/io/ConnectionRequest.html] classes) that allows you to make HTTP requests to arbitrary destinations. When an application is running inside a browser as a JavaScript app, it's constrained by the same origin policy. You can make network requests to the same host that served the app originally.

For example, If your application is hosted at http://example.com/myapp/index.html, then your app will be able to perform network requests to retrieve other resources under the **example.com** domain, but it won't be able to retrieve resources from **example2.com**, **foo.net**, etc.



The HTTP standard does support cross-origin requests in the browser via the `Access-Control-Allow-Origin` HTTP header. Some web services supply this header when serving resources, but not all. The way to be make network requests to arbitrary resources is to do it through a proxy.

A ParparVM JavaScript build generates a same-origin proxy wrapper by default and configures the application to use `/cn1-cors-proxy?_target=`. Select its platform with the `javascript.proxy.target` build hint:

Value	Generated deployment bundle
`jakarta-servlet`	A Jakarta Servlet WAR for current containers such as Tomcat 10.1 or newer. This is the default.
`javax-servlet`	A Servlet 4 WAR for older containers such as Tomcat 9.
`node`	A Node.js 20 application ZIP that serves both the static app and the proxy.
`php`	A PHP 8 application ZIP containing the static app, proxy endpoint, and Apache rewrite configuration.
`aws-lambda`	An AWS SAM/Node.js 20 proxy ZIP. Static assets are included under `public/` for deployment to your static host.
`google-cloud-functions`	A Google Cloud Functions/Node.js 20 proxy ZIP. Static assets are included under `public/`.
`cloudflare-workers`	A Cloudflare Worker ZIP with a static-assets binding.
`none`	Don't generate a proxy deployment bundle.

Restrict a generated proxy before deploying it publicly. Set `javascript.proxy.allowedTargets` to a comma-separated list of allowed origins, host names, or wildcard subdomains. For example:

```
javascript.proxy.allowedTargets=https://api.example.com,*.services.example.org
```

If this hint is omitted, the generated proxy accepts any HTTP or HTTPS target and the build emits a warning.

Proxy hint precedence is explicit:

1. `javascript.inject_proxy=false` disables proxy generation and URL injection.
2. `javascript.proxy.url` configures an existing external proxy and suppresses generated proxy packaging unless `javascript.proxy.target` is also set.
3. `javascript.proxy.target=none` disables generated proxy packaging.
4. Otherwise, `javascript.proxy.target` selects the bundle, defaulting to `jakarta-servlet`.

80.4.1. Step 1: Setting up a proxy



You can skip this step when deploying one of the generated ParparVM wrappers. Use it when you host only the static ZIP or want to operate the proxy independently.

For a ParparVM build, select the matching generated wrapper with `javascript.proxy.target`. The old cors-proxy [https://github.com/shannah/cors-proxy] project remains relevant only to legacy TeaVM WARs and older Java EE deployments.

If you don't want to install the `.war` file, but would rather copy the proxy servlet into an existing web project, you can do that also. See the cors-proxy wiki for more information about this [https://github.com/shannah/cors-proxy/wiki/Embedding-Servlet-into-Existing-Project].

80.4.2. Step 2: Configuring your application to use the proxy

There are three ways to configure your application to use your proxy.

1. Using the **javascript.proxy.url** build hint.

   E.g.:

   ```
   javascript.proxy.url=http://example.com/myapp/cn1-cors-proxy?_target=
   ```

2. By modifying your app's **index.html** file after the build.

   E.g.:

   ```
   <script type="text/javascript">
       window.cn1CORSProxyURL='http://example.com/myapp/cn1-cors-proxy?_target=';
   </script>
   ```

3. By setting the `javascript.proxy.url` property in your Java source. You would do this inside your `init()` method, but it has to be executed before you make a network request that requires the proxy.

   ```
   Display.getInstance().setProperty(
           "javascript.proxy.url",
   ```

```
        "http://example.com/myapp/cn1-cors-proxy?_target="
);
```

The method you choose will depend on the workflow that you prefer. Options #1 and #3 will almost always result in fewer changes than #2 because you've to set them up once, and the builds will retain the settings each time you build your project.

80.5. Using the CORS proxy for same origin requests

By default, the CORS proxy is used for HTTP requests to URLS at a different domain than the one that the app is running in. Some circumstances where you may want to **even** use the proxy for same domain requests. You can do this by setting the `javascript.useProxyForSameDomain` display property to `true`. For example:

```
Display.getInstance().setProperty("javascript.useProxyForSameDomain", "true");
```

Why would you want to do this?

The browser shields some HTTP headers (for example, "Set-Cookie") from JavaScript so that your app can't access them. Going through the proxy works around this limitation by copying and encoding such headers in a format that the browser will allow, and then decoding them client-side to make them available to your app seamlessly.

80.5.1. Using apache as a proxy

If you're hosting your application on an Apache 2 web server with mod_proxy installed, and you need to make CORS requests to a single domain (or a limited set of domains), you can use Apache to serve as your proxy. One sample configuration (which you would place either in your `VirtualHost` definition or your `.htaccess` file is as follows:

```
SSLProxyEngine on
ProxyPass /app https://www.myexternaldomain.com
ProxyPassReverse /app https://www.myexternaldomain.com
```

This tells Apache to proxy all requests for '/app' to the domain https://www.myexternaldomain.com. You would then need to set your CORS proxy URL in your CN1 app to `/app/`.

The syntax is the same if you've multiple domains, but keep attention to the order of the lines to make the proxy working. For example:

```
SSLProxyEngine on
ProxyPass /app https://www.myexternaldomain1.com
ProxyPassReverse /app https://www.myexternaldomain1.com
ProxyPass /storage https://www.myexternaldomain2.com
ProxyPassReverse /storage https://www.myexternaldomain2.com
```

This tells Apache to proxy all requests for '/app' to the domain https://www.myexternaldomain1.com and all requests for '/storage' to the domain https://www.myexternaldomain2.com

80.6. Customizing the splash screen

Since your application may include many resource files, videos, etc., the build-server will generate a splash screen for your app to display while it's loading. This basically shows a progress indicator with your app's icon.

You can customize this splash screen by modifying the HTML source inside the **cn1-splash** `div` tag of your app's index.html file:

```
<div id="cn1-splash">
    <img class="icon" src="icon.png"/>

    <img class="progress" src="progress.gif"/>
    <p>...Loading...</p>
</div>
```

80.7. Debugging

If you run into problems with your app that occur in the JavaScript version, you may need to do a little bit of debugging. Many debugging tools for JavaScript, but the preferred tool for debugging Codename One apps is Chrome's debugger.

If your application crashes and you don't have a clue where to begin, follow these steps:

1. Load your application in Chrome.
2. Open the Chrome debugger.
3. Enable the "Pause on Exceptions" feature, then click the "Refresh" button to reload your app.
4. Step through each exception until you reach the one you're interested in. Chrome will then show you a stack trace that includes the name of the Java source file and line numbers.

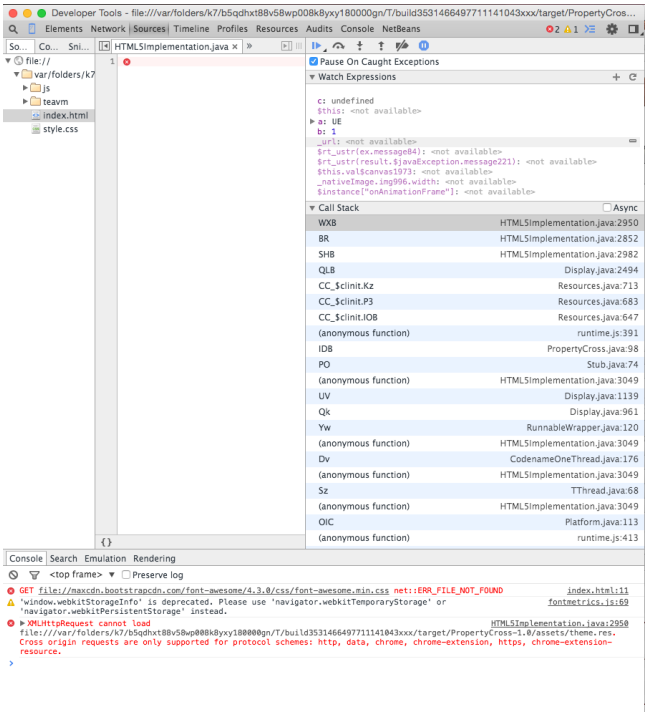


Figure 377. Debugging using Chrome tools

80.8. Including Third-Party JavaScript libraries

Codename One allows you to interact directly with JavaScript using native interfaces. Native interfaces are placed inside your project's `native/JavaScript` directory using a prescribed naming convention. If you want to, include third-party JavaScript libraries in your application you should also place these libraries inside the `native/JavaScript` directory but you must specify which files should be treated as "libraries" and which files are treated as "resources." You can do this by adding a file with extension `.cn1mf.json` file either the root of your `native/JavaScript` directory or the root level of the project's `src` directory.

80.8.1. Libraries vs resources

A **resource** is a file whose contents can be loaded by your application at runtime using `Display.getInstance().getResourceAsStream()`. In a typical Java environment, resources would be stored on the application's classpath (usually inside a Jar file). On iOS, resources are packaged inside the application bundle. In the JavaScript port, resources are stored inside the `APP_ROOT/assets` directory. Historically, JavaScript files have always been treated as resources in Codename One, and many apps include HTML and JavaScript files for use inside the BrowserComponent [https://www.codenameone.com/javadoc/com/codename1/ui/BrowserComponent.html].

With the JavaScript port, it isn't quite so clear whether a JavaScript file is meant to be a resource or a library that the application itself uses. Most of the time you probably want JavaScript files to be used as libraries, but you might also have JavaScript files in your app that are meant to be loaded at runtime and displayed inside a Web View - these would be considered resources.

80.8.2. The JavaScript manifest file

To differentiate libraries from resources, you should provide a **cn1mf.json** file inside your `native/JavaScript` directory that specifies any files or directories that should be treated as libraries.

This file can be named anything you like, as long as its name ends with `cn1mf.json`. Any files or directories that you list in this manifest file will be packaged inside your app's `includes` directory instead of the `assets` directory. It adds appropriate `<script>` tags to include your libraries as part of the `index.html` page of your app.



If you include the `cn1mf.json` file in your project's `src` directory it could be used to add configuration parameters to platform's other than JavaScript (although no other platforms use this feature). If you place it inside your `native/JavaScript` directory, then the JavaScript port will use the configuration contained therein.

A simple manifest file might contain the following JSON:

```
{
    "javascript" : {
        "libs" : [
            "mylib1.js"
        ]
    }
}
```

That is, It contains an object with key **libs** whose value is a list of files that should be treated as libraries. In the above example, you're declaring that the file `native/JavaScript/mylib1.js` should be treated as a library. This will result in the following `<script>` tag being added to the `index.html` file:

```
<script src="includes/mylib1.js"></script>
```



This also caused the `mylib1.js` file to be packaged inside the `includes` directory instead of the `assets` directory.



A project may contain more than one manifest file. This allows you to include manifest files with your cn1libs also. You need to make sure that each manifest file has a different name.

How to NOT generate the `script` tag

Sometimes you may want a JavaScript file to be treated as a library (that is, packaged in the `includes` directory) but not automatically included in the `index.html` page. Rather than specifying the name of the file in the `libs` list, you can provide a structure with multiple options about the file. For example,

```
{
    "JavaScript" : {
        "libs" : [
            "mylib1.js",
            {
                "file" : "mylib2.js",
```

```
                "include" : false
            }
        ]
    }
}
```

In the above example, the `mylib2.js` file will be packaged inside the `includes` directory, but the build server won't insert its `<script>` tag in the `index.html` page.

Library directories

You can also specify directories in the manifest file. In this case, the entire directory will be packaged inside the `includes` directory of your app.



If you're including JavaScript files in your app that are contained inside a directory hierarchy, you should specify the root directory of the hierarchy in your manifest file and use the sub "includes" property of the directory entry to specify which files should be included with `<script>` tags. Specifying the file directly inside the "libs" list will result in the file being packed directly in the your app's `includes` directory. This may or may not be what you want.

For example,

```
{
    "javascript" : {
        "libs" : [
            "mylib1.js",
            {
                "file" : "mylib2.js",
                "include" : false
            },
            {
                "file" : "mydir1",
                "includes" : ["subfile1.js", "subfile2.js"]
            }
        ]
    }
}
```

In this example the entire `mydir1` directory would be packed inside the app's `includes` directory, and the following `script` tags would be inserted into the `index.html` file:

```
<script src="includes/mydir1/subfile1.js"></script>
<script src="includes/mydir1/subfile2.js"></script>
```



Libraries included from a directory hierarchy may not work with the single file preview that the build server generates. For that version, it will embed the

> contents of each included JavaScript file inside the `index.html` file, but the rest of the directory contents will be omitted. If your the library depends on the directory hierarchy and supporting files and you require the single-file preview to work, then you may consider hosting the library on a separate server, and including the library directly from there, rather than embedding it inside your project's "native/JavaScript" directory.

Including remote libraries

The examples so far have demonstrated the inclusion of libraries that are part of the app bundle. But, you can also include libraries over the network by specifying the URL to the library directly. This is handy for including common libraries that are hosted by a CDN.

For example, The Google Maps library requires the Google Maps API to be included. This is accomplished with the following manifest file contents:

```
{
    "javascript" : {
        "libs" : [
            "//maps.googleapis.com/maps/api/js?v=3.exp"
        ]
    }
}
```



> This example uses the "//" prefix for the URL instead of specifying the protocol directly. This allow the library to work for both http and https hosting. You could but specify the protocol as well:

+

```
{
    "javascript" : {
        "libs" : [
            "https://maps.googleapis.com/maps/api/js?v=3.exp"
        ]
    }
}
```

Including CSS files

CSS files can be included using the same mechanism as is used for JavaScript files. If the file name ends with `.css`, then it will be treated as a CSS file (and included with a `<link>` tag instead of a `<script>` tag. For example,

```
{
    "JavaScript" : {
        "libs" : [
```

```
            "mystyles.css"
        ]
    }
}
```

or

```
{
    "JavaScript" : {
        "libs" : [
            "https://example.com/mystyles.css"
        ]
    }
}
```

Embedding variables in URLs

Sometimes the URL of a library may depend on the values of some build hints in the project. For example, in the Google Maps cn1lib, the API key must be appended to the URL of the API as a GET parameter. For example, `https://maps.googleapis.com/maps/api/js?v=3.exp&key=SOME_API_KEY`, but the developer of the library doesn't want to put his own API key in the manifest file for the library. It would be better for the API key to be supplied by the developer of the actual app that uses the library and not the library itself.

The solution for this is to add a **variable** into the URL as follows:

```
{
    "JavaScript" : {
        "libs" : [
            "//maps.googleapis.com/maps/api/js?v=3.exp&key={{javascript.googlemaps.key}}"
        ]
    }
}
```

The `{{javascript.googlemaps.key}}` variable will be replaced with the value of the `javascript.googlemaps.key` build hint by the build server, so the resulting include you see in the index.html page will be something like:

```
<script src="//maps.googleapis.com/maps/api/js?v=3.exp&key=XYZ"></script>
```

80.9. Browser environment variables

Native interfaces allow you to interact with the JavaScript environment in unlimited ways, but Codename One provides a simpler method of obtaining some common environment information from the browser via the `Display.getInstance().getProperty()` method. The following environment variables are currently available:

Table 18. Property hints for the JavaScript port

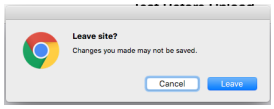
Name	Description
`browser.window.location.href`	A String, representing the entire URL of the page, including the protocol (like http://)
`browser.window.location.search`	A String, representing the querystring part of a URL, including the question mark (?)
`browser.window.location.host`	A String, representing the domain name and port number, or the IP address of a URL
`browser.window.location.hash`	A String, representing the anchor part of the URL, including the hash sign (#)
`browser.window.location.origin`	A String, representing the protocol (including ://), the domain name (or IP address) and port number (including the colon sign (:) of the URL. For URLs using the "file:" protocol, the return value differs between browsers
`browser.window.location.pathname`	A String, representing the pathname
`browser.window.location.protocol`	A String, representing the protocol of the current URL, including the colon (:)
`browser.window.location.port`	A String, representing the port number of a URL. + Note: If the port number isn't specified or if it's the scheme's default port (like 80 or 443), an empty string is returned
`browser.window.location.hostname`	A String, representing the domain name, or the IP address of a URL
`User-Agent`	The User-agent string identifying the browser, version etc.
`browser.language`	The language code that the browser is currently set to. (for example, en-US)
`browser.name`	the name of the browser as a string.
`Platform`	a string that must be an empty string or a string representing the platform on which the browser is executing. + For example: `MacIntel`, `Win32`, `FreeBSD i386`, "WebTV OS"
`browser.codeName`	the internal name of the browser
`browser.version`	the version number of the browser
`javascript.deployment.type`	Specifies the deployment type of the app. This will be "file" for the single-file preview, "directory" for the zip distribution, and "war" for the war distribution.

80.10. Changing the native theme

The JavaScript port selects a bundled native theme from the detected browser platform. You can override this behavior by setting the `javascript.native.theme` Display [https://www.codenameone.com/javadoc/com/codename1/ui/Display.html] property to the path of a theme included with your app. Place additional JavaScript-only themes in the `native/javascript` directory so they aren't bundled into other platforms.

80.11. Disabling the 'OnBeforeUnload' handler

By default, apps will display warning/confirm dialog when the user attempts to leave the page.



You can explicitly enable or disable this behaviour by setting the "platformHint.javascript.beforeUnloadMessage" display property. Setting the property to `null` will disable this behaviour, so that users won't be harassed by this dialog when they navigate away from the app. Setting it to a string value, like "leaving so soon?," will re-enable this behaviour.



Some browsers don't allow you to specify the message that's displayed in this dialog. In those browsers, this property can be viewed as boolean: A null value will result in no prompt being shown, and a non-null value will result in a prompt being shown.

80.11.1. Example: Toggling the BeforeUnload prompt On/Off

```
Form f = new Form("Test Before Unload", BoxLayout.y());
CheckBox enableBeforeUnload = new CheckBox("Enable Before Unload");
enableBeforeUnload.setSelected(true);
enableBeforeUnload.addActionListener(e->{
    if (enableBeforeUnload.isSelected()) {
        CN.setProperty("platformHint.javascript.beforeUnloadMessage", "Are you sure you want to leave this page?  It
might be bad");
    } else {
        CN.setProperty("platformHint.javascript.beforeUnloadMessage", null);
    }
});
f.add(enableBeforeUnload);
f.show();
```

80.12. Deploying as a progressive web app

Out of the box, your app is ready to be deployed as a progressive web app (PWA). That means that users can access the app directly in their browser, but once the browser determines that the user is frequenting the app, it will prompt the user to install the app on their home screen. Once installed on the home screen, the app will behave like a native app. It will continue to work while offline, and if the user launches the app, it will open without the browser's navigation bar. If you were to install the native and PWA versions of your app side by side, you would be hard-pressed to find the

difference - especially on newer devices.

Below is a screenshot from Chrome for Android where the browser is prompting the user to add the app to their home screen.

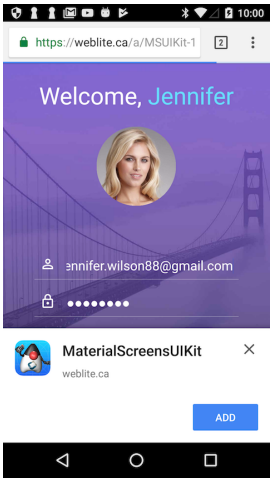


Figure 378. Add app to homescreen banner

If the app is available as a native app, in the Play store, you can indicate this using the `javascript.manifest.related_applications` and `javascript.manifest.prefer_related_applications` build hints. Then, instead of prompting the user to add the web app to their home screen, they'll be prompted to install the native app from the Play store, as shown below.

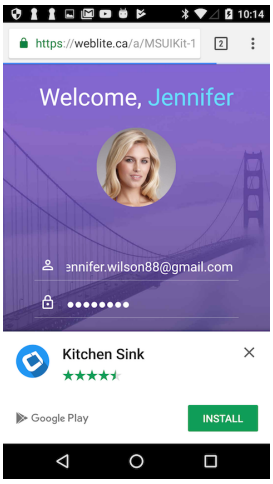


Figure 379. Add native app banner



The PWA standard requires that you host your app on over HTTPS. For testing purposes, it will also work when accessed at a `localhost` address. You can use the Lighthouse PWA analysis tool [https://developers.google.com/web/ilt/pwa/lighthouse-pwa-analysis-tool] to ensure compliance.

For more information about Progressive Web Apps see Google's introduction to the subject [https://developers.google.com/web/progressive-web-apps/].

80.12.1. Customizing the app manifest file

At the heart of a progressive web app is the web app manifest [https://developer.mozilla.org/en-US/docs/Web/Manifest]. It specifies things like the app's name, icons, description, preferred orientation,

display mode (for example, whether to display browser navigation or to open with the full screen like a native app), associated native apps, etc. The Codename One build server will automatically generate a manifest file for your app but you can (and should) customize this file via build hints.

Build hints of the form `javascript.manifest.XXX` will be injected into the app manifest. For example, To set the app's description, you could add the build hint:

```
javascript.manifest.description=An app for doing cool stuff
```

You can find a full list of available manifest keys here [https://developer.mozilla.org/en-US/docs/Web/Manifest]. The build server will automatically generate all the icons so you don't need to worry about those. The "name" and "short_name" properties will default to the app's display name, but they can be overridden via the `javascript.manifest.name` and `javascript.manifest.short_name` build hints respectively.



The `javascript.manifest.related_applications` build hint expects a JSON formatted list, like in the raw manifest file.

80.12.2. Related applications

One nice feature (discussed above) of progressive web apps, is the ability to specify related applications in the app manifest. Browsers that support the PWA standard use some heuristics to "offer" the user to install the associated native app when it's clear that the user is using the app on a regular basis. Use the `javascript.manifest.related_applications` build hint to specify the location of the native version of your app. For example, `javascript.manifest.related_applications=[{"platform":"play", "id":"my.app.id"}]`

You can declare that the native app is the preferred way to use the app by setting the `javascript.manifest.prefer_related_applications` build hint to "true."



According to the app manifest documentation [https://developer.mozilla.org/en-US/docs/Web/Manifest], this should be used if the related native apps do offer something that the web application can't do.

80.12.3. Device/Browser support for PWAs

Chrome and Firefox both support PWAs on desktop and on Android. iOS doesn't support the PWA standard, however, many aspects of it are supported. For example, On iOS you can add the app to your home screen, after which time it will appear and behave like a native app - and it will continue to work while offline. However, many other nice features of PWA like "Install this app on your home screen" banners, push notifications, and invitations to install the native version of the app, aren't supported. It's unclear when, or even, whether Apple will ever add full support; but most experts predict that they will join the rest of the civilized world and add PWA support in the near future.

On the desktop, Chrome provides an analogous feature to "add to your homescreen": `Add to shelf`. If it looks like the user is using the app on a regular basis, and it isn't yet installed, it will show a

banner at the top of the page asking the user if they want to add to their shelf.

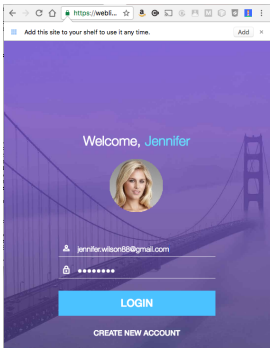


Figure 380. Add to shelf banner

Clicking the "Add button" prompts the user for the name they wish the app to appear as:

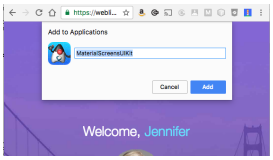


Figure 381. Add to shelf prompt

Upon submission, Chrome will generate a real application (on Mac, it will be a `.app`, on Windows, an "exe," etc.) which the user can double-click to open the app directly in the Chrome. And, importantly, the app will still work when the user is offline.

The app will also appear in their "Shelf" which you can always access at `chrome://apps`, or by opening the "Chrome App Launcher" app (on OS X this is located in "~/Applications/Chrome Apps/Application Launcher").

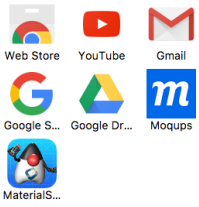


Figure 382. Chrome App Launcher



The Chrome App Launcher lists apps installed both via the Chrome Web Store and via the "Add to Shelf" feature discussed here. The features described in this article are orthogonal to the Chrome Web Store and won't be affected by its closure.

80.13. Playing media and opening links

People don't like it when the browser automatically starts playing sounds, or opening links without their permission. For this reason, modern browsers restrict your ability to programmatically do these things, unless they're in response to a user action, like a mouse click.

If your app needs to play media (for example, `Media.play()`) without the user actually interacting physically (for example, key press or pointer press), then it will display a popup dialog confirming

that the user actually wants to perform this action. Opening a link (for example, `Display.execute("…")`) used to do the same, but now defaults to navigating the current page instead of prompting — see Opening links below.

Sometimes this dialog may affect the utility of the app. For example, suppose you want to play a video in response to a voice command. Having to press an "OK" button after the command, may be annoying. For such cases, you can use the `platformHint.javascript.backsideHooksInterval` property to **poll** for media play requests on an authorized event.

For example:

```
CN.setProperty("platformHint.javascript.backsideHooksInterval", "1000");

// Now your app will process media.play() and Display.execute(null) calls
// once per second (1000ms). If play() or execute() has been called anytime
// in that second (since the last poll), it will seamlessly process the
// request.


// To disable polling, just set it to an interval 0 or lower.
// for example, CN.setProperty("platformHint.javascript.backsideHooksInterval", "0");
```



Don’t abuse this feature. You should enable this polling when necessary. For example, If your app enables the user to listen for voice commands, enable polling for the period of time that it’s listening. When the user wants to stop listening, you should also stop the polling by setting the interval to "0."

80.13.1. Choosing where a link opens

Opening a link deserves its own note, because the restriction above bites almost every app that calls `Display.execute()`. Codename One dispatches events on its own EDT, so by the time your action listener runs, the browser no longer sees the tap as an ongoing user gesture and refuses to open a new window. Navigating the page the app already runs in has no such restriction: it always works, at the cost of unloading the app.

The port picks between those two with the `javascript.execute.target` property:

Value	Behavior
`auto`	The default. Opens a new tab while the page still has user activation, and navigates the current page when it doesn’t. No confirmation sheet is ever shown.
`_blank`	Only ever opens a new tab. When the browser blocks that, the port shows a confirmation sheet whose OK button supplies the gesture the browser wanted. This was the behavior before the property existed.
`_self`	Always navigates the page the app runs in.

```
// The default. Opens a new tab while the browser still considers a
// user gesture to be in progress, and navigates the page the app runs
// in when it doesn't. Never shows a confirmation sheet.
CN.setProperty("javascript.execute.target", "auto");

// Always navigate the page the app runs in. Never blocked and never
// prompts, but it unloads the app.
CN.setProperty("javascript.execute.target", "_self");

// Only ever open a new tab. When the browser blocks that, the port
// shows a confirmation sheet whose OK button supplies the gesture the
// browser wanted. This was the behavior before the property existed.
CN.setProperty("javascript.execute.target", "_blank");

CN.execute("https://www.codenameone.com/");
```



Under `auto` and `_self` the app is unloaded whenever the current page navigates away, so persist anything you need before you call `execute()`. Pick `_blank` when keeping the app alive matters more than avoiding the sheet.

The property applies to any URL the browser can hand off, which means anything carrying a URI scheme. Custom deep links such as `imdb:///find?q=godfather` count, and so do protocol-relative `//host/path` URLs. The exceptions keep their existing meaning: a `javascript:` URL still evaluates on the page, and a `data:` URL, a `file:` URL or a path into local storage still becomes a download.

80.14. BrowserComponent lifecycle

On the JavaScript port, `BrowserComponent` uses an iframe that's added to the DOM when the component is initialized. Understanding the lifecycle is important when showing `Dialog`s over forms that contain `BrowserComponent` instances.

80.14.1. Dialogs and BrowserComponent

When a `Dialog` is shown, the parent `Form` is deinitialized. This causes any `BrowserComponent` instances on that form to also be deinitialized, which removes their iframes from the DOM. When the `Dialog` is dismissed and the parent form is re-shown, the `BrowserComponent` is reinitialized with a **new iframe**, losing all JavaScript state.

This affects:

- **Stateful embedded editors** - content and cursor position lost
- **WebRTC** connections - broken when removed from DOM
- **Interactive visualizations** - charts, maps, diagrams with JavaScript state
- **Any JavaScript state** - event listeners, variables, etc.

80.14.2. Preserving BrowserComponent state

To preserve the iframe and its JavaScript state across deinitialization cycles, set the `HTML5Peer.removeOnDeinitialize` client property to `Boolean.FALSE`:

```
BrowserComponent browser = new BrowserComponent();
browser.putClientProperty("HTML5Peer.removeOnDeinitialize", Boolean.FALSE);
```

This keeps the iframe in the DOM even when the component is deinitialized, preserving all JavaScript state.

80.14.3. Lifecycle example

Without the property set:

1. `Dialog.show()` is called
2. Parent `Form`'s `deinitialize()` is called
3. `BrowserComponent` is deinitialized, iframe removed from DOM
4. Dialog is dismissed
5. Parent `Form`'s `initComponent()` is called
6. `BrowserComponent` creates a **new iframe** - JavaScript state is lost

With `removeOnDeinitialize=false`:

1. `Dialog.show()` is called
2. Parent `Form`'s `deinitialize()` is called
3. `BrowserComponent` is deinitialized, but iframe **stays in DOM**
4. Dialog is dismissed
5. Parent `Form`'s `initComponent()` is called
6. `BrowserComponent` reconnects to the existing iframe - JavaScript state preserved

Chapter 81. Working with macOS

81.1. macOS Desktop Build Options

Launch Codename One Settings from the Maven project root and open **Hints** to configure `desktop.mac.*` build hints such as entitlements and `Info.plist` entries:

```
mvn cn1:settings
```

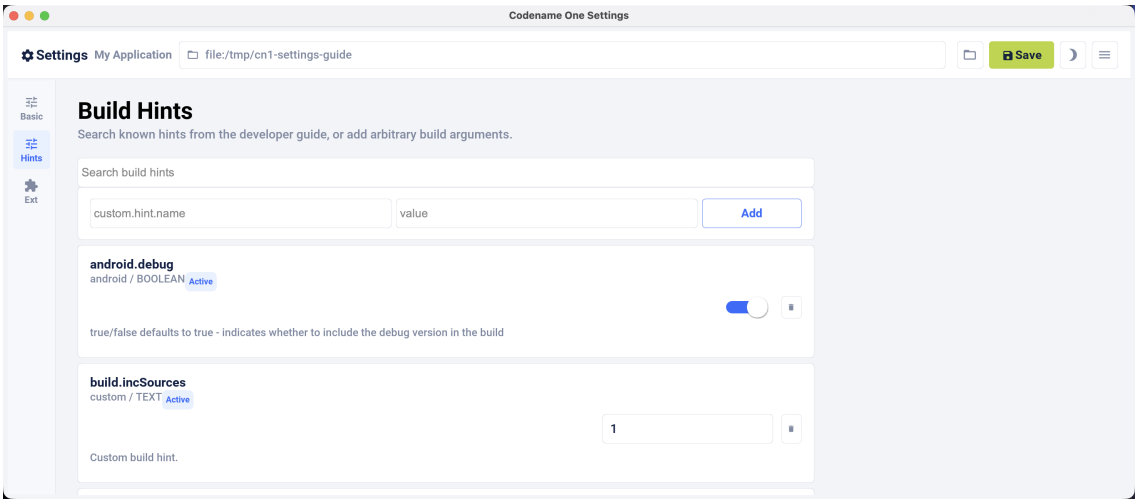


Figure 383. Desktop macOS build hints

Desktop signing values that aren't build hints are top-level entries in `common/codenameone_settings.properties`: `codename1.desktop.mac.certificate` contains the `.p12` path and `codename1.desktop.mac.certificatePassword` contains its password. The Maven desktop macOS build reads these properties together with the `desktop.mac.*` hints.

81.1.1. Bundle Types

Three bundle types dictate what the build server produces for you when you build your project as a Desktop macOS App.

1. **DMG** - Produces a `.DMG` disk image with your app. This is the preferred format for distributing your app outside of the Mac App Store. If you provide a Developer ID Application certificate (see "Understanding Certificates" below), the app will be signed so that users won't receive warnings about "Unidentified developer" when they install your app.
2. **Sandboxed DMG** - Same as **DMG** bundle type except that your app is set up to use the app sandbox. This is typically used to test an app that's being distributed via the App Store, since App Store apps **must** use the sandbox. If you select this bundle type, you're **required** to provide a Mac App Distribution Certificate, and you should also specify entitlements required for your app to function correctly. For more information about the app sandbox, see Apple's documentation on the subject [https://developer.apple.com/app-sandboxing/].
3. **Mac App Store Upload (PKG)** - Produces a `.PKG` file that you can upload to the Mac App Store. This requires that you provide **both** a Mac App Distribution certificate, and a Mac App Installer certificate (see "Understanding Certificates" below). Both of these certificates should be

embedded into a single `.p12` file (See "Exporting Certificates as p12" below).

81.1.2. Understanding Mac Certificates



Currently (as of Jan. 5, 2020) the `mac.desktop-vm` build hint is ignored if you include a certificate with your app. This limitation should be fixed soon.

For the purposes of Mac application distribution, there are 3 types of certificates that you will be interested in. The type(s) of certificate required will depend on the type of bundle you generate. The certificate types are:

1. **Developer ID Application Certificate (Mac applications)**

   This type of certificate is used to sign an app to be distributed **outside** of the Mac App Store as a DMG image. This corresponds to the "DMG" bundle type. You can identify this kind of certificate because its identity will be of the form "Developer ID Application: YOUR COMPANY NAME (SOMECODE)." For example, Developer ID Application: Acme Widgets Corp. (XYSD5YF).

2. **Mac App Distribution Certificate (Mac App Store)**

   This type of certificate is used to sign the.app bundle for an app that's to be distributed in the Mac App Store. This certificate is required for both the "Sandboxed DMG," and "Mac App Store Upload (PKG)" bundle types. You can identify this kind of certificate because its identity will be of the form "3rd Party Mac Developer Application: YOUR COMPANY NAME (SOMECODE)." For example, 3rd Party Mac Developer Application: Acme Widgets Corp. (XYSD5YF).

3. **Mac App Installer Certificate (Mac App Store)**

   This type of certificate is used to sign the.pkg installer for an app that's being submitted to the Mac App Store. This certificate is required for the "Mac App Store Upload (PKG)" bundle type. You can identify this kind of certificate because its identity will be of the form "3rd Party Mac Developer Installer: YOUR COMPANY NAME (SOMECODE)." For example, 3rd Party Mac Developer Installer: Acme Widgets Corp. (XYXD5YF).

81.1.3. Obtaining Certificates



Apple provides documentation on obtaining certificates on its website [https://developer.apple.com/library/content/documentation/IDEs/Conceptual/AppDistributionGuide/MaintainingCertificates/MaintainingCertificates.html#//apple_ref/doc/uid/TP40012582-CH31-SW6] but the process described there involves Xcode. This section describes an alternate process that doesn't require Xcode.

If you have an Apple developer account, you can manage your certificates here [https://developer.apple.com/account/mac/certificate].



Figure 384. Mac Developer portal certificates

The screenshot above shows an account that already has the three kinds of certificates you will require:

1. **Developer ID Application** - Used for the DMG bundle type.
2. **Mac App Distribution** - Used for the Sandboxed DMG and Mac App Store Upload (PKG) bundle types.
3. **Mac App Installer** - Used for the Mac App Store Upload (PKG) bundle type.

If your account doesn't yet have a certificate of the required type, you should begin by pressing the "+" button in the upper right. This will bring you to a page asking "What type of Certificate do you need?" Two options on this page will interest you:

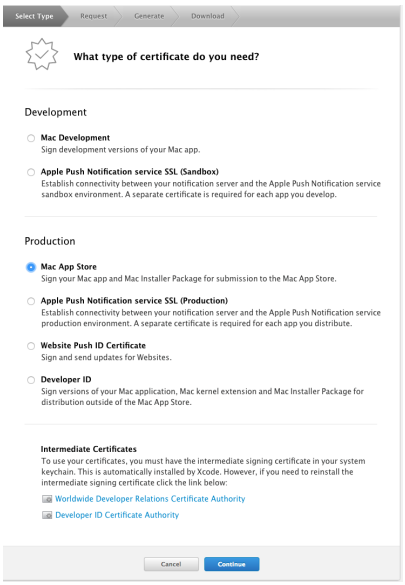


Figure 385. Creating a new certificate

1. **Production** > **Mac App Store** - For both the Mac App Distribution and Mac App Installer certificates.
2. **Production** > **Developer ID** - For the Developer ID Application certificate.

Select the option corresponding to the certificate you wish to generate. In either case, you'll be taken to a form to select whether you want an "Installer" certificate or an "Application" certificate. Select the appropriate type.

You will then be prompted to upload a Certificate Signing Request (CSR) file, and it will provide instructions on how to do this via the Keychain app.



You can reuse the same CSR file for generating all 3 certificates.

After generating the certificates, you should download them to your Mac, and import them into

your keychain. You should be able to accomplish this by simply double-clicking the downloaded ".cer" file, and following the prompts.

81.1.4. Exporting Certificates as P12



The following section requires access to a Mac, and assumes that you've already generated your 3 certificates

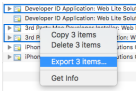
Notice that Mac apps may require three different kinds of certificates, yet `codename1.desktop.mac.certificate` accepts a single P12 file. This isn't a mistake. P12 files may contain more than one certificate, and you're expected to include all the certificates that the build server may require inside a single P12. The build server automatically extracts the certificates it needs according to the bundle type.

When building the "DMG" bundle type, the build server will look for a "Developer ID Application Certificate" inside the P12. If one is found, it will be used to sign the app bundle.

The "Sandboxed DMG" target will look for a "Mac App Distribution Certificate" certificate in the P12.

The "Mac App Store Upload (PKG)" target will require both a "Mac App Distribution Certificate" and a "Mac App Installer Certificate" to be included in the P12.

The easiest way to produce a P12 that includes all 3 kinds of certificates is to export them from the Keychain Access app (Requires a Mac). Select all 3 certificates at once (using CMD-click), then right-click and select "Export 3 Items..."



You will then be prompted to select a location to save the `.p12` file, as well as selecting a password for the file.

81.1.5. Entitlements

When distributing apps in the Mac App Store, or when using the "Sandboxed DMG" bundle type, your app is run inside a sandboxed environment, meaning that it doesn't have access to the outside world. It's provided its own "sandboxed" container for file system access, and it doesn't get any network access. If your app requires access to the "outside world," request the required entitlements with `desktop.mac.entitlement. entries in the *Hints` view.

For more information about the app sandbox, and a full list of entitlements, see Apple's documentation on the subject [https://developer.apple.com/library/content/documentation/Miscellaneous/Reference/EntitlementKeyReference/Chapters/EnablingAppSandbox.html].



You can specify entitlements using the build hint `desktop.mac.entitlement.XXXX` where XXX is the entitlement you wish to set. For example, `desktop.mac.entitlement.com.apple.security.network.client=true`

81.1.6. Custom Info.plist Entries

You can add custom entries to the Info.plist file [https://developer.apple.com/library/archive/documentation/General/Reference/InfoPlistKeyReference/Introduction/Introduction.html] using the following build hints:

desktop.mac.plist.KEYNAME

Injects the entry with key `KEYNAME` into the Info.plist file. For example, `desktop.mac.plist.LSApplicationCategoryType=public.app-category.business`

81.2. Mac Native Build (AOT-compiled macOS App)

The "macOS Desktop" section above covers the JavaSE bundled-JRE wrapper. **Mac Native** is a separate, AOT-compiled target: it routes through the same iOS pipeline (ParparVM bytecode-to-C, native UIKit) so the resulting macOS app shares its rendering code path, its memory profile, and its certificate story with your iPhone / iPad build. The output is a single signed `.app` bundle suitable for Mac App Store or Developer ID distribution.

This target is a good fit when:

- Your app already builds against the iOS slice and you want a Mac version that looks and behaves the same.
- You need native-speed startup and rendering on Mac, not a wrapped JVM.
- You want a single AppKit-grade `.app` bundle without bundling a JRE.

Java source is shared with the iOS target; no separate Mac-specific code is required.

81.2.1. Local Build

To generate an Xcode project locally that you can open and run on the Mac:

```
mvn -B -Dcodename1.platform=ios -Dcodename1.buildTarget=mac-source package
```

The output is written to:

```
target/<finalName>-mac-source/<MainClass>.xcodeproj
```

Open the project in Xcode, select the Mac Catalyst destination, and press Run. The resulting `.app` is identical in shape to one produced by the cloud build server.

81.2.2. Cloud Build

To produce a release-ready signed Mac app on the Codename One build server:

```
mvn -B -Dcodename1.platform=ios -Dcodename1.buildTarget=mac-os-x-native package
```

Mac distribution requires **its own certificates and provisioning profile** — Apple issues separate *Mac App Distribution* (3rd Party Mac Developer Application) and *Developer ID Application* certificates that aren't interchangeable with the *iPhone Distribution* certificate used for iOS App Store builds. Configure them via the parallel `codename1.mac.*` hints:

Hint	Purpose
`codename1.mac.appid`	Mac bundle identifier (the App Store Connect record is distinct from the iOS one).
`codename1.mac.certificate`	Path to the `.p12` containing the Mac signing certificate. Bundle both *Mac App Distribution* and *Developer ID Application* into a single P12 when targeting both channels.
`codename1.mac.certificatePassword`	Password to unlock the P12.
`codename1.mac.provision`	Path to the Mac `.provisionprofile`.

The Apple Developer Team ID itself can stay shared with the iOS slice (single Apple Developer account) — `macNative.teamId` falls back to `ios.release.teamId` when not set. The server returns a signed `.app` bundle.

81.2.3. IDE Shortcuts

The IntelliJ workspace template ships two Mac Native run configurations alongside the iOS ones:

- **Mac Native Project** (Local Builds folder) — equivalent to running `mvn package -Dcodename1.buildTarget=mac-source`. Use this to generate the Xcode project for hands-on debugging.
- **Mac Native Build** (Build Server folder) — equivalent to `mvn package -Dcodename1.buildTarget=mac-os-x-native`. Use this for cloud builds.

81.2.4. Distribution

The Mac Native build supports both Mac App Store and Developer ID distribution. The generated Xcode project includes two `ExportOptions-*-Mac.plist` files in `dist/`:

- `ExportOptions-AppStore-Mac.plist` — App Store distribution (sandboxed, hardened runtime off).
- `ExportOptions-DeveloperID-Mac.plist` — Developer ID distribution (hardened runtime on, sandbox off).

After archiving in Xcode, run:

```
xcodebuild -exportArchive \
  -archivePath build/<MainClass>.xcarchive \
  -exportOptionsPlist dist/ExportOptions-AppStore-Mac.plist \
  -exportPath build/export
```

81.2.5. Entitlements

The Mac Native build emits a `<MainClass>.entitlements` plist (or `<MainClass>-AppStore.entitlements` + `<MainClass>-DeveloperID.entitlements` when both channels are configured). The defaults are sensible: App Sandbox on for the App Store channel, hardened runtime on for the Developer ID channel, network client access enabled, user-selected file access enabled. The signing-certificate identity defaults to *Apple Distribution* for the App Store channel and *Developer ID Application* for the Developer ID channel.

81.2.6. Supported Distribution Channels

Channel	Notes
App Store	Sandboxed, hardened runtime off. Signed with *Apple Distribution*; uploaded via App Store Connect.
Developer ID	Hardened runtime on, sandbox off. Signed with *Developer ID Application*; notarized before distribution.

Chapter 82. App Store Submission and Deployment

Once a build is signed you still have to get it in front of users: upload the binary, fill in the release notes, set the store listing, and send it for review — on every store where your users are. Codename One automates that last mile as a one-stop shop. From the build console you can submit a finished build to the **App Store**, the **Mac App Store**, **Google Play**, and **Huawei AppGallery** with one click, manage the store **listing itself** — title, description, keywords, what's-new, screenshots — as code in your project, and produce per-store Android packages so you can reach the large Android audience that isn't on Google Play.

Reaching those audiences doesn't depend on, and can't be restricted by, any single store. Google Play always receives the standard, unmodified Android App Bundle, so your Play listing follows Google's rules exactly; the additional channels (AppGallery and other Android markets) are separate artifacts delivered to separate stores.

This chapter covers the end-to-end flow, credentials, the metadata-as-code workflow, and multi-store Android distribution. It builds on Signing: submission reuses the same App Store Connect API key the certificate wizard configures.

82.1. What it does, and what it doesn't

Automated submission delivers the binary and applies your release metadata to an **existing** app on the store. It doesn't create the app for you.



You must create the app once, by hand, in App Store Connect [https://appstoreconnect.apple.com] and/or the Google Play Console [https://play.google.com/console] before you can submit to it. Neither store's API can register a brand-new app, set up pricing, in-app purchases, the privacy questionnaire, or the content-rating questionnaire — those remain one-time manual steps in the store consoles. After that, every release's binary and listing can be shipped automatically.

Three things are worth separating, because they happen at different times:

- **The app record** — the listing shell in the store. Created once, manually, and exists before any build.
- **The binary** — produced by your build, delivered when you **submit**.
- **The metadata** — description, what's-new, screenshots. Managed as code and applied when you submit. Metadata attaches to the app record and the new version, not to the binary, so there's no chicken-and-egg problem: the app already exists on the store by the time you build.

82.2. Prerequisites

82.2.1. Store credentials

Apple. Submission uses the App Store Connect API key described under the certificate wizard. If you have already run `mvn cn1:certificatewizard` and stored the key, you are ready — the same key that creates certificates and profiles also uploads builds and manages listings.

Google. Create a Google Play service account with the **Android Publisher** role in the Google Play Console (API access > Service accounts), download its JSON key, and add it to the build console's submission credentials page. The console shows which stores are connected.

Huawei. In AppGallery Connect, create an API client under **Users and permissions** > **API client** and note its **Client ID** and **Client Secret**, then add them on the same submission credentials page. The app must already exist in AppGallery Connect (submission binds to the app resolved from your package name).

The Submissions page shows a connect card for each store you haven't configured yet; fill one in and save to enable one-click submission to that store.

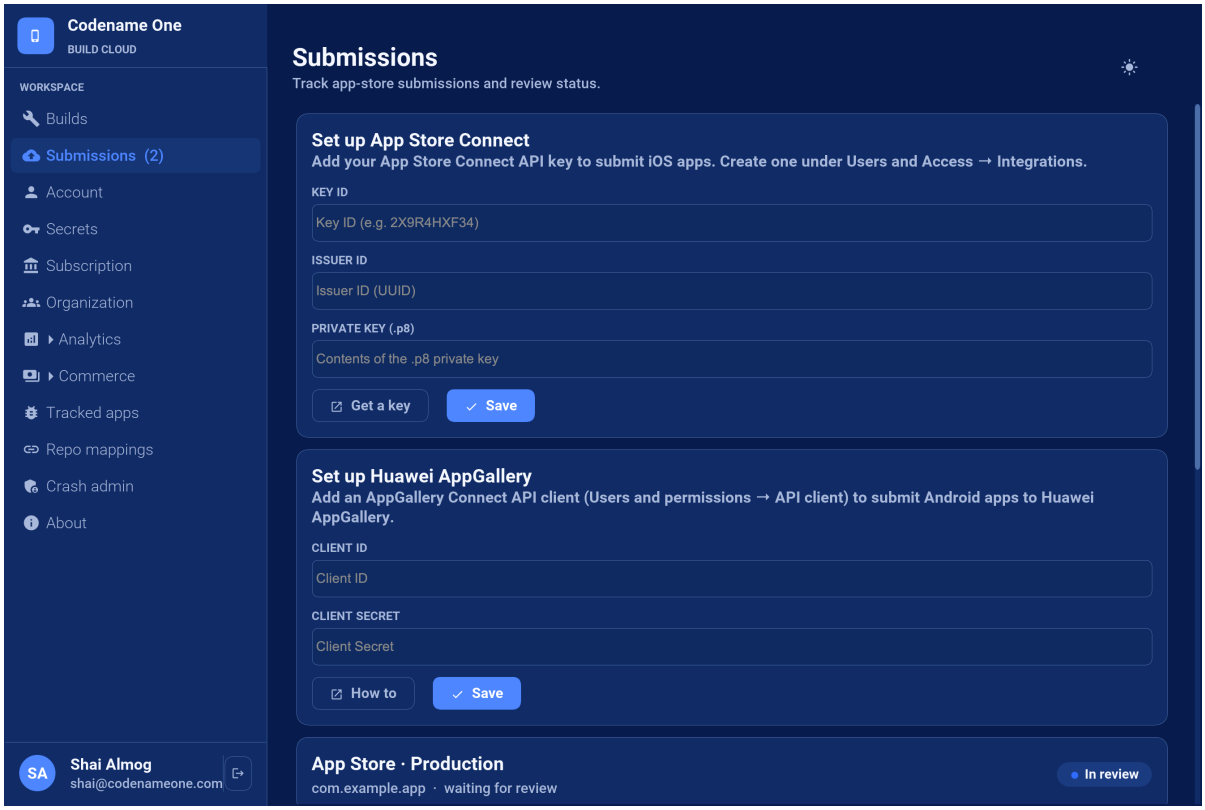


Figure 386. The submission credentials cards in the build console

82.2.2. Beta vs production

When you submit you choose a destination. **Beta** means TestFlight (Apple) or the internal-testing track (Google). **Production** means App Store review (Apple) or the production track (Google).

82.3. Submitting a build from the console

On any successful build the console shows a **Submit** action for each store the build can go to:

- **Submit to App Store** — when the build produced an iOS `.ipa`.

- **Submit to Mac App Store** — when the build produced a native/Catalyst `.pkg` (see the **Working with macOS** chapter). An iOS build that also produced a `.pkg` offers both.
- **Submit to Google Play** — for an Android build.
- **Submit to AppGallery** — for an Android build, to Huawei AppGallery. Production sends the build for review; the beta destination uploads it and leaves it as a draft for you to promote or open-test.

The submit dialog lets you pick the destination (Beta or Production) and add release notes for this submission, then delivers the binary. For an Apple **production** submission the console then tracks the App Store review state (processing → in review → approved/rejected) and can email you when the state changes. Google submissions and TestFlight deliveries complete as soon as the binary is delivered.



Submitting to **Production** on Apple opens an App Store review submission automatically. Submitting to **Beta** delivers to TestFlight without opening a review.

82.4. Metadata as code

Release metadata — the store listing text and screenshots — can live in your project as a folder of plain text and image files, versioned in git alongside your code, and pushed to the stores with a single command. This makes your listing reproducible and CI-drivable: generate release notes from your changelog, push, and the next submission applies it to both stores. It's the alternative to typing the same fields into two different web consoles by hand for every release.

82.4.1. The metadata folder

By default the goal reads a `cn1-metadata` folder next to your `codenameone_settings.properties`. It's organized by store, then locale, with one plain text file per listing field:

```
cn1-metadata/
  apple/
    primary_locale.txt          (optional; defaults to the first locale)
    primary_category.txt        (optional; an App Store Connect category id)
    secondary_category.txt      (optional)
    en-US/
      name.txt                  (app name, max 30 chars)
      subtitle.txt              (max 30 chars)
      description.txt           (max 4000 chars)
      keywords.txt              (comma separated, max 100 chars)
      promotional_text.txt      (max 170 chars)
      whats_new.txt             (release notes for this version)
      marketing_url.txt
      support_url.txt
      privacy_url.txt
      screenshots/
        APP_IPHONE_67/1.png 2.png ...
        APP_IPAD_PRO_129/1.png ...
```

```
  google/
    en-US/
      name.txt                  (listing title, max 30 chars)
      subtitle.txt              (short description, max 80 chars)
      description.txt           (full description, max 4000 chars)
      whats_new.txt             (release notes, max 500 chars)
      support_url.txt
      screenshots/
        phoneScreenshots/1.png 2.png ...
        tenInchScreenshots/1.png ...
```

Every file is optional; a field you don't include keeps whatever value the store already has. The same neutral file names map to each store's fields (for example `subtitle.txt` becomes the App Store **subtitle** and the Google Play **short description**). Fields a store doesn't have (Apple `keywords`/`promotional_text`) are simply ignored for the other store.

The `screenshots/<dir>` sub-folders are named with each store's own type identifiers: App Store `screenshotDisplayType` values (for example `APP_IPHONE_67`) for Apple, and listing `imageType` values (for example `phoneScreenshots`) for Google. Images are uploaded in filename order.



You don't have to create the folder by hand. Run `mvn cn1:metadata-init` to scaffold `cn1-metadata` with an empty template of the field files, a screenshots folder, and a `README`, ready to fill in:

+

```
mvn cn1:metadata-init                        # apple + google, en-US
mvn cn1:metadata-init -Dstore=apple -Dlocale=fr-FR
```

+ It only creates what's missing, so it's safe to re-run (for example to add a new locale) — it never overwrites files you've already filled in.

82.4.2. Pushing the metadata

Run the goal from your project; it authenticates with the same Codename One account your builds use (run `mvn cn1:certificatewizard` once to sign in, or pass `-Dtoken=`):

```
# push both stores from cn1-metadata/
mvn cn1:metadata-push

# or a single store
mvn cn1:metadata-push -Dstore=apple

# override the folder or the package id if needed
mvn cn1:metadata-push -DmetadataDir=path/to/metadata -Dpackage=com.example.app
```

The package id defaults to `codename1.packageName` from `codenameone_settings.properties`. The goal

validates every field against the store's limits before storing it, so an over-long name or a malformed URL fails fast with a clear message. Wire it into CI to keep the stored listing in sync with your repo.



`metadata-push` uploads the descriptor and screenshots to Codename One; it does **not** submit. The stored metadata is applied to the store the next time you **submit** a build for that package.

82.4.3. When and how it's applied

At submission time the stored descriptor is applied to the store automatically:

- **Apple** — applied to the **production** App Store version (App Store review), while the version is still editable. TestFlight has no store listing, so beta Apple submissions don't apply listing metadata.
- **Google** — applied to the listing on the edit that publishes the binary, on any track.

Applying metadata is **best-effort**: the binary is delivered first, so a rejected field (say a name that's too long, or a locale the app hasn't enabled) never blocks the submission — it's recorded as a warning you can see in the console, and the rest of the listing still applies. You can always fix a field by hand in the store console afterward.

82.4.4. Screenshots aren't stored long-term

Screenshots are a transient pass-through: `metadata-push` uploads them, the next submission delivers them to the store, and they're then deleted from Codename One's servers. Keep the canonical images in your repo (that's the point of managing them as code) and re-push when they change. Text descriptors are retained until you delete them, your account is deleted, or they go unused for about a year.

Per-image limits: up to 10 MB, PNG or JPEG, and at most 10 screenshots per locale and display type (the stores' own limit).

82.5. Reaching Android users beyond Google Play

A large share of Android users — most of the market in China, and many devices elsewhere — can't install from Google Play. Reaching them means shipping to other Android stores: Huawei AppGallery, and the many app markets run by device makers and carriers (Xiaomi, OPPO, VIVO, Tencent MyApp, Baidu, 360 and others). Codename One makes your app available on those stores without changing how you build or locking you to any of them.

Huawei AppGallery is a first-class submission target, exactly like the App Store or Google Play (see Submitting a build above). For the remaining Android markets, Codename One produces **distribution-channel packages**.

82.5.1. Distribution-channel packages

Most of these stores accept a plain, self-signed APK rather than an App Bundle, and they attribute

each download and its revenue to the store it came from. A distribution-channel package is your signed release APK stamped with a store identifier — one APK per store, all built from the same release.

You choose the stores; the build produces one package per store, each carrying its channel id. Your app reads that id at runtime with `Display.getInstance().getProperty("DistributionChannel", "")` — an empty string for a normal Google Play build — so you can report the install source (to your own analytics or the store's) **without adding any third-party channel SDK**. Codename One writes the id and reads it back for you.



Google Play is never a channel package. Play always receives the standard, unmodified Android App Bundle, so nothing about this feature affects your Play listing or its compliance — the channel packages are separate APKs for separate stores.

Sensible defaults keep this simple: enable it and you get packages for the major Android markets out of the box, and you can narrow or extend the channel list to match the stores you actually publish to. Each package is signed with your app's release key, so every store receives a signed, installable APK.

82.6. Benefits over editing the stores by hand

Managing submission and metadata through Codename One pays off most for teams, frequent releases, many locales, or many apps:

- **Automated release notes.** Generate what's-new from your changelog and ship it to both stores on every release — the one field that always changes.
- **One source, two stores.** A single descriptor fans out to App Store Connect and Google Play, which have different fields and limits.
- **Reproducible and reviewable.** The listing lives in git, versioned and code-reviewed like everything else. Console edits have no history.
- **Localization at scale.** Localized listings are just files, and can be generated or translated in a pipeline.
- **Fewer console logins.** Team members who ship releases don't each need App Store Connect or Play Console access; they ship through the build pipeline instead.

For a single app with infrequent updates, editing the store console by hand is fine — the payoff scales with how often you release and how many listings you maintain.

Chapter 83. Desktop Integration

Codename One is mobile-first, but the same app can run on the desktop: as a Java SE “Run as desktop app” / executable jar, and as a native macOS app via the local `mac-source` (Mac Catalyst) build target. By default a desktop build looks like a phone app in a window - a mobile-style `Toolbar` at the top of every `Form`, fading touch scrollbars, and commands tucked into a hamburger side menu.

The desktop integration features make a desktop build **feel** like a desktop app:

- A native window title bar that shows the `Form` title (instead of the in-app `Toolbar`).
- A native menu bar populated from your `Command` objects, with per-command placement hints (About, File, Edit, Quit, …).
- Interactive scrollbars - a draggable thumb that highlights on hover and while dragged, click-to-page on the track, a minimum thumb size so it stays easy to grab on long content, a reserved gutter so content isn’t drawn under the bar, and an always-visible (rather than fading) scrollbar.
- Keyboard accelerators on the native menu items (a Save shortcut, a Quit shortcut).
- Native desktop notifications driven by the standard `LocalNotification` API.

These behaviors are **opt-in**. They’re switched on by default for newly generated projects (both the Maven archetype and the start.codenameone.com initializr) and stay completely inert on phones/tablets - all the new behavior is gated on `CN.isDesktop()`, so a mobile build of the same project is unaffected.

83.1. Title-bar modes

The window chrome is selected by the `desktop.titleBar` setting, which takes one of three values:

`native`	Hide the CN1 `Toolbar`. The `Form` title goes into the OS window title bar and the toolbar’s commands are bridged to a native menu bar. This is the default for newly generated apps.
`custom`	Undecorated window (no OS title bar) where the CN1 `Toolbar` itself acts as the window’s title bar: it shows the `Form` title and side menu, dragging it moves the window, and the window is resized by dragging its edges. The commands are bridged to the native menu bar **and** remain in the toolbar’s side menu.
`toolbar`	Legacy behavior - the in-app CN1 `Toolbar` is shown exactly as on mobile.

The screenshots below show the same notes app, with File / Edit / View / Help commands, in each mode on macOS.

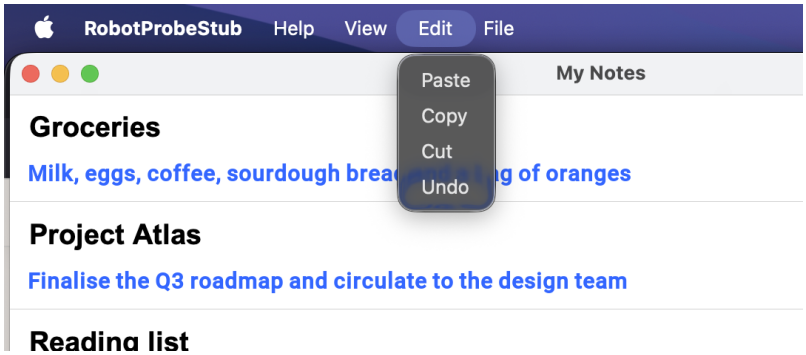


Figure 387. `native` mode: the `Form` title is in the OS window title bar, the commands populate the native menu bar, and there is no in-app `Toolbar`

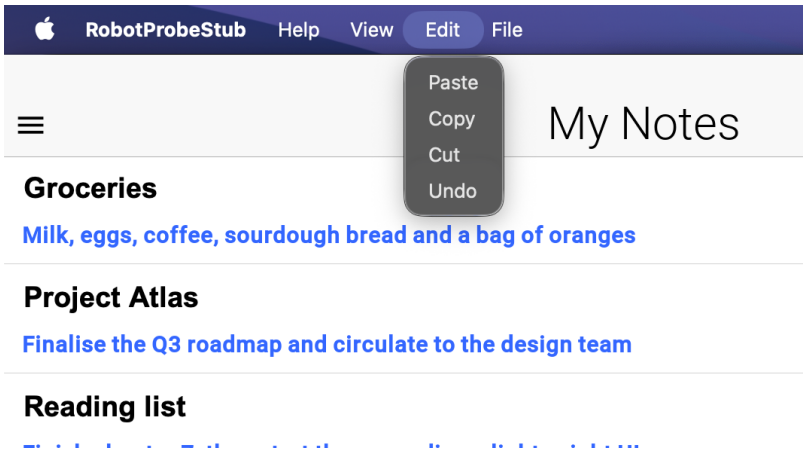


Figure 388. `custom` mode: the window is undecorated and the CN1 `Toolbar` (title + hamburger side menu) is the window's title bar; the commands are also in the native menu bar

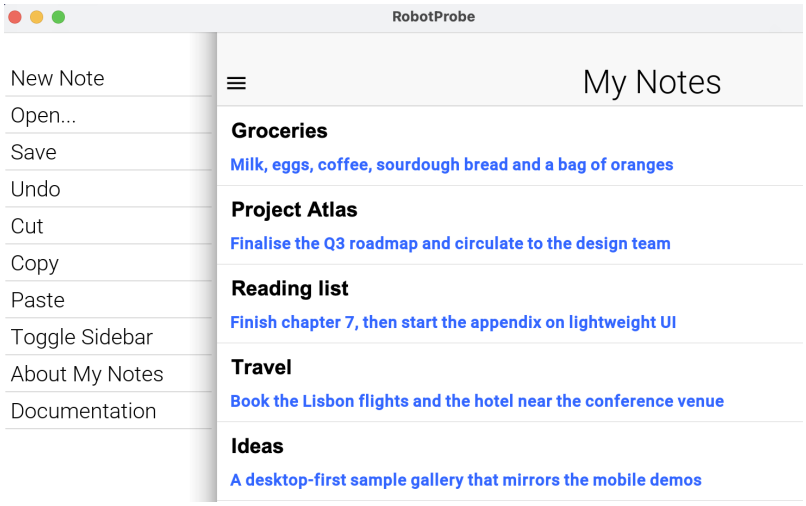


Figure 389. `toolbar` mode: the legacy look - the in-app `Toolbar` sits below a generic OS title bar and the commands live in the hamburger side menu

83.1.1. Enabling it on the Java SE desktop build

For the desktop (Java SE) target the mode is a build hint that the generated desktop launcher applies for you. In `codenameone_settings.properties`:

```
codename1.arg.desktop.titleBar=native
codename1.arg.desktop.interactiveScrollbars=true
```

Newly generated projects already contain these lines. The generated `<MainName>Stub` reads them at startup, so no code is required. Because the stub is the desktop entry point, the settings only ever take effect on the desktop.

83.1.2. Enabling it on the macOS (Mac Catalyst) build

The local `mac-source` build runs through ParparVM and the iOS port, where the chrome is provided by native `UIWindowScene` / `UIMenuBuilder` code. There it's read as a runtime property and defaults to `toolbar`, so existing Catalyst apps are unaffected. Opt in from your app before the first `Form` is shown:

```
Display.getInstance().setProperty("desktop.titleBar", "native");
Display.getInstance().setProperty("desktop.interactiveScrollbars", "true");
```



The native title bar and native menu bar on macOS require iOS 13 / macOS 10.15 (Catalyst) or newer. On a phone/tablet iOS build these properties are ignored.

83.2. Native menu bar and command placement hints

When the `Toolbar` is hidden (`native` or `custom` mode), the commands you add through the `Toolbar` / `Form` command API are bridged to the platform's native menu bar - a Swing `JMenuBar` (which becomes the macOS screen menu) on the Java SE build, and a `UIMenuBuilder` menu on Catalyst. Selecting a menu item runs the command's `actionPerformed` on the Codename One EDT, exactly as a button press would.

By default every command lands under a single **Commands** menu. To place commands into the conventional desktop menus, tag them with a desktop-menu hint using `Command.setDesktopMenu(String)` and the `Command.DESKTOP_MENU_*` constants:

```
Form hi = new Form("My App", BoxLayout.y());

Command about = new Command("About My App");
about.setDesktopMenu(Command.DESKTOP_MENU_ABOUT);   // application menu

Command open = new Command("Open Filenull");
open.setDesktopMenu(Command.DESKTOP_MENU_FILE);     // File menu

Command quit = new Command("Quit");
quit.setDesktopMenu(Command.DESKTOP_MENU_QUIT);     // application menu

hi.addCommand(about);
hi.addCommand(open);
hi.addCommand(quit);
hi.show();
```

The recognised values are (case-insensitive):

`DESKTOP_MENU_APP`, `DESKTOP_MENU_ABOUT`, `DESKTOP_MENU_PREFERENCES`, `DESKTOP_MENU_QUIT`	Placed in the application menu (the macOS app menu / a leading menu).
`DESKTOP_MENU_FILE`, `DESKTOP_MENU_EDIT`, `DESKTOP_MENU_VIEW`, `DESKTOP_MENU_WINDOW`, `DESKTOP_MENU_HELP`	Placed in the matching standard top-level menu.
Any other string	Becomes a top-level menu with that literal title.
Unset	Grouped under a default **Commands** menu.

Each port maps the hint to its native structure: on Catalyst the command is inserted into the real `UIMenuApplication` / `UIMenuFile` / `UIMenuEdit` / … menus; on the Java SE build a top-level `JMenu` is created per hint group.



The hint is a plain client property (`Command.DESKTOP_MENU`), so it's inert on platforms that don't have a native menu bar - it's safe to set it unconditionally in shared code.

83.2.1. Keyboard accelerators

Give a command a keyboard accelerator with `Command.setDesktopShortcut(…)`. The shortcut is shown next to the menu item and triggers the command from the keyboard:

```
Command save = new Command("Save");
save.setDesktopMenu(Command.DESKTOP_MENU_FILE);
save.setDesktopShortcut('S');   // Cmd+S on macOS, Ctrl+S on Windows/Linux

Command saveAs = new Command("Save Asnull");
saveAs.setDesktopMenu(Command.DESKTOP_MENU_FILE);
saveAs.setDesktopShortcut('S',
        Command.DESKTOP_SHORTCUT_MODIFIER_PRIMARY | Command.DESKTOP_SHORTCUT_MODIFIER_SHIFT);
```

The single-argument form uses the platform's **primary** modifier - Command on macOS, Control on Windows/Linux - which is the right default for most menu shortcuts. The two-argument form takes a bit-mask of `Command.DESKTOP_SHORTCUT_MODIFIER_PRIMARY`, `Command.DESKTOP_SHORTCUT_MODIFIER_SHIFT` and `Command.DESKTOP_SHORTCUT_MODIFIER_ALT`. On the Java SE build the accelerator becomes a Swing `KeyStroke`; on Catalyst it becomes a `UIKeyCommand`. Like the placement hint, the accelerator is inert on platforms without a native menu bar, so it's safe to set unconditionally.

83.3. Interactive scrollbars

Enabling `desktop.interactiveScrollbars` turns the scrollbar from a fading touch indicator into a desktop-style control:

- The thumb can be grabbed and dragged, and it **highlights** while the pointer hovers it and while

it's being dragged (the `DesktopScrollThumb` selected / pressed styles).

- Clicking the track pages toward the pointer by about one viewport.
- The thumb has a **minimum size** so it stays easy to grab even when the content is far taller than the viewport. The minimum (in pixels) is the `scrollThumbMinSizeInt` theme constant; when unset it defaults to the equivalent of 4mm.
- A **gutter** is reserved for the scrollbar so content isn't laid out underneath it. The gutter width comes from the `DesktopScroll` / `DesktopHorizontalScroll` UIID padding (see below).
- The scrollbar stays visible rather than fading out.

This is a cross-platform core feature driven by the `interactiveScrollBool` theme constant, which the desktop ports inject only when `CN.isDesktop()` is true. Mouse-wheel scrolling continues to work as before.

83.4. Theme UIIDs

When interactive scrollbars are active, the desktop scrollbar is styled through dedicated UIIDs so the mobile `Scroll` / `ScrollThumb` styling is never affected:

`DesktopScroll`, `DesktopScrollThumb`, `DesktopHorizontalScroll`, `DesktopHorizontalScrollThumb`	The interactive desktop scrollbar track and thumb. The horizontal-/vertical-scroll UIID's padding sets the scrollbar gutter width. The thumb UIID's `.selected` and `.pressed` styles render the hover and drag highlight respectively.

The modern native themes ship platform-conventional defaults for these UIIDs (macOS-style in the iOS theme, Material-style in the Android theme), including the `.selected` / `.pressed` thumb highlight states; see Chapter 5, *Native modern themes*. Override them in your app's `theme.css` like any other UIID. The minimum thumb length is the `scrollThumbMinSizeInt` theme constant (pixels). In `custom` mode the title bar is just the regular CN1 `Toolbar`, so it's themed through the usual `Toolbar` / `Title` UIIDs.

83.5. Desktop notifications

The standard `com.codename1.notifications.LocalNotification` API works on the desktop. On the Java SE desktop build a scheduled notification surfaces as a native OS notification through the system tray (Notification Center on macOS, the notification area on Windows/Linux); on the macOS / Mac Catalyst build it goes through the same `UNUserNotificationCenter` path as iOS. Clicking the notification delivers it to your app's `LocalNotificationCallback`, exactly as on mobile, so the same scheduling and handling code is shared across phone and desktop:



On the Java SE build desktop notifications require an OS with system-tray support (`java.awt.SystemTray`); where it's unavailable the notification is logged and skipped.

Chapter 84. Working with the native Windows port

The native Windows port compiles a Codename One app to a real, standalone Win32 executable — a single `.exe` with **no JVM and no runtime dependencies**. It's the Windows analog of the iOS port: the same ParparVM pipeline that turns your Java/ Kotlin bytecode into C and then into a native binary, here targeting Windows with the LLVM toolchain and rendering through Direct2D / DirectWrite.

This is distinct from the long-standing **Windows desktop** build, which packages the app to run on a Java Virtual Machine (see Section 84.7, "Native Windows vs. the desktop (JVM) app vs. the jar" for a practical comparison). The native port needs no JVM on the target machine.

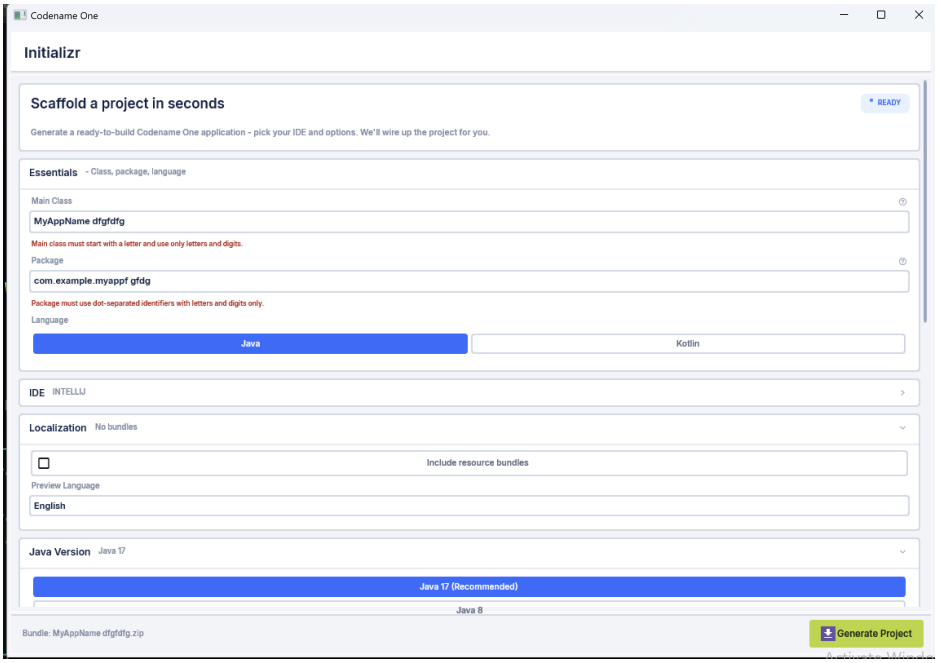


Figure 390. The Codename One Initializr app running as a native Windows `.exe` (ParparVM + Direct2D/DirectWrite, no JVM)



The native Windows port is a young port. It renders, runs and edits text, but several platform services are still being filled in. Treat it as a foundation to build on rather than a feature-complete replacement for the desktop build, and consult the port's `status.md` for the current gap list.

84.1. How it works (the technology stack)

The port reuses Codename One's portable architecture and swaps in a Win32 / Direct2D implementation layer:

- **ParparVM "clean" C target.** The same VM that powers iOS translates your app the Codename One core + the minimal Java runtime to C. A CMake project is generated and compiled with **clang-cl** (LLVM on the MSVC ABI) + **Ninja** into a native `.exe`. There is no bytecode interpreter and no JNI — your code **is** the native binary.

- **Concurrent garbage collector.** ParparVM's non-blocking GC runs natively, the same collector used on iOS.
- **Direct2D + DirectWrite + WIC.** All 2D graphics (primitives, gradients, clipping, affine and perspective transforms, images) go through Direct2D; glyph layout, measurement and rasterization use DirectWrite; image decode/encode uses Windows Imaging Component (WIC). Rendering is GPU-accelerated.
- **Media Foundation** backs `Media` playback, and **WebView2** backs `BrowserComponent` (when the WebView2 SDK is present at build time).
- **WinHTTP** backs networking; raw sockets and WebSockets use Winsock.
- **Single self-contained executable.** There is no `.app`-style bundle directory on Windows, so the app's classpath resources — the theme `.res`, images, localization, the material icon font — are embedded directly into the executable's PE resource section and read back at runtime through `getResourceAsStream`. The result is one file you can copy and run.

84.2. Building a native Windows app

A native Windows build runs through the `codenameone-maven-plugin` like every other target; the project produced by the Codename One Initializr already knows how to build it. The heavy lifting — translate to C, generate the CMake project, configure and build with clang-cl/Ninja, collect the `.exe` — is handled by the builder, so the configuration on your side is trivial.

The build can run two ways, and the builder picks automatically based on the host:

- **On a Windows host** it uses clang-cl + CMake + Ninja inside a Visual Studio developer environment (the CRT and Windows SDK reach the compiler through `vcvarsall`).
- **On any other host — for example a Linux build server** it cross-compiles with clang-cl + lld-link + llvm-rc against a Windows SDK laid out by `xwin` [https://github.com/Jake-Shadle/xwin]. clang is a cross-compiler, so the PE it emits is identical to a Windows-host build. Point the `windows.sdkRoot` build hint (or the `CN1_XWIN_SYSROOT` environment variable) at the `xwin splat` directory. This is how the build cloud produces Windows binaries without a Windows machine.

Both **x64** and **arm64** are produced this way — clang-cl cross-compiles to either architecture from either host; pick the target with the `windows.arch` build hint.

What you **can't** do off Windows is **run** the result: Direct2D/DirectWrite need a real Windows GPU/display stack (emulation such as Wine is too incomplete to trust), so rendering can only be verified on Windows. A Windows 11 VM (Parallels, Hyper-V, …) or a real machine is therefore the place to **test** the app, even when the binary itself was built on Linux.

84.3. Cloud builds (the `win32` target)

On the Codename One build cloud the native Windows build is the **win32** target, and it builds on Linux (the Android/JavaScript build group) using the cross-compile above. Trigger it like any other cloud target:

```
mvn -pl common package -Dcodename1.platform=windows -Dcodename1.buildTarget=windows-device
```

The convenience goal `mvn -pl common cn1:buildWin32` does the same thing. A regular (release) build returns **two** binaries — x64 and arm64, both stripped release exes. Setting the `windows.debug` build hint instead returns a **single** x64 exe with symbols, for diagnosis. (To build locally on a Windows box, use `-Dcodename1.buildTarget=local-windows-device`.)

84.4. Build hints

The native Windows port adds the build hints below. They use the `windows.` prefix; don't confuse them with the older `win.` hints, which configure the **JVM-based Windows desktop installer** (`win.installDirName`, `win.shortcutName`, …) and have no effect on the native port.

Table 19. Native Windows port build hints

Name	Description
windows.arch	Target CPU architecture for the `.exe`: `x64` (default) or `arm64`. Synonyms `x86_64`/`amd64` and `aarch64` are accepted. clang-cl cross-compiles to the chosen architecture from either host.
windows.debug	`true`/`false` (default `false`). When `false`, the `.exe` is optimized and **stripped** — no PDB, with the linker dead-stripping unreferenced code (`/OPT:REF`) and folding identical functions (`/OPT:ICF`). Set `true` to keep debug symbols (a `.pdb` next to the exe) so a native crash address can be symbolized while you are developing. Optimizations stay on in either case.

The full cross-platform build-hints reference is in the Advanced topics chapter [# _sending_arguments_to_the_build_server].

84.5. Optimized, with debug info in a separate file

The shipping build is optimized (`/O2`): the linker removes unused functions and folds identical ones (`/OPT:REF` and `/OPT:ICF`), and the debug information isn't embedded in the executable — it's written to a separate `.pdb` companion. The `.exe` therefore stays as small as the translated code allows, while a faulting address can still be turned back into a function name and source line with `llvm-symbolizer` against the `.pdb`. As a concrete data point, the Codename One Initializr sample builds to an ~8 MB `.exe`; the matching `.pdb` is several times larger but ships beside the binary rather than inside it, and is only needed to symbolize a crash. Setting `windows.debug=true` switches to an unoptimized debug build with full symbols.

84.6. Code signing

Unlike iOS and Android, Windows does **not** require a signature to **run** a desktop exe — an unsigned

win32 build launches fine. But an unsigned binary that a user downloads shows **"Unknown publisher"** in the UAC prompt and trips **SmartScreen** ("Windows protected your PC"), so anything you actually distribute should be **Authenticode-signed**. Signing is therefore optional in the port (the default is unsigned) but strongly recommended for release.

The port signs with `osslsigncode` [https://github.com/mtrojnar/osslsigncode], which signs Windows PE files on **any** OS — so signing happens right in the Linux build cloud, no Windows machine needed. Provide a code-signing certificate and the build hints below, and each produced exe (x64 and arm64) is signed and timestamped.

84.6.1. Signing with a PKCS#12 certificate

Put your code-signing certificate and key in a PKCS#12 (`.pfx`/`.p12`) file and point your project settings at it:

```
# codenameone_settings.properties
codename1.windows.signing.certificate=/secure/path/codesign.p12
codename1.windows.signing.password=your-pkcs12-password
# optional signature metadata + timestamping (shown with their defaults):
codename1.arg.windows.signing.name=Acme Corp
codename1.arg.windows.signing.url=https://acme.example
codename1.arg.windows.signing.timestampUrl=http://timestamp.digicert.com
codename1.arg.windows.signing.digest=sha256
```

You configure the certificate once and it's used for **both local and cloud builds** — a cloud build automatically sends it with the build request (the same way the iOS and Android signing certificates are uploaded), so there is nothing extra to do for the build cloud. The signature is **timestamped** (RFC 3161) by default, so it keeps verifying after the certificate expires; set `windows.signing.timestampUrl` empty to turn that off. Set `windows.signing=false` to force an unsigned build even when a certificate is configured.

84.6.2. Hardware-backed and cloud keys

Since June 2023 the CA/Browser Forum requires code-signing private keys to live in a hardware module, so a brand-new OV/EV certificate is no longer a plain `.pfx`. The usual answer is a managed signing service — **Azure Trusted Signing**, **DigiCert KeyLocker**, **eSigner** from SSL.com — which exposes the key through PKCS#11. `osslsigncode` speaks PKCS#11, so the build host points at the service's provider per its docs; the rest of the flow (digest, timestamp, the resulting signed exe) is identical. Plain `.pfx` signing remains fine for internal/test certificates and for certs issued before the cutoff.



`osslsigncode` must be on the build host's `PATH` (or set `CN1_OSSLSIGNCODE` to its location). If signing is requested but the certificate or tool is missing, the build fails with a clear message rather than shipping an unsigned exe by surprise.

84.7. Native Windows vs. the desktop (JVM) app vs. the jar

Codename One can produce three different Windows artifacts from the same project. They trade off self-containment, size, startup and maturity differently:

	Native Windows `.exe`	**Desktop app (JVM)**	**Executable jar**
Summary	A real Win32 binary (ParparVM → C → native), Direct2D/DirectWrite rendering.	Your app running on a JVM with the Java SE port (Swing-hosted rendering), packaged as a Windows installer.	A runnable `.jar` of your app plus the Codename One jars.
JVM required on the target?	No — fully self-contained.	No if the installer bundles a JRE; otherwise it relies on one.	Yes — a compatible JDK (11–25) must be installed on the machine.
Artifact size	One `.exe`, on the order of 5–10 MB (everything — VM, GC, your code, resources — is inside; ~8 MB for the Initializr sample), plus a separate `.pdb` for crash symbolication.	Largest: app + a bundled JRE (the JRE alone is ~40–70 MB).	Smallest: a few MB (just bytecode); the JVM lives outside it.
Startup	Native process start — no VM warm-up.	JVM start-up cost.	JVM start-up cost.
CPU architecture	Native x64 **and** arm64 binaries (`windows.arch`).	Intel x64 only — it ships with a bundled **Intel Java 8 JRE** (no arm64 build; on an arm64 machine it runs through Windows' x64 emulation).	Whatever the installed JVM supports.
Look & rendering	Direct2D/DirectWrite, GPU-accelerated; the material native theme.	Swing-hosted Codename One rendering (the simulator's pipeline).	Same as the desktop app.
Portability of the artifact	Windows-only, and per-architecture.	Windows-only installer.	Cross-platform — the same jar runs anywhere a supported JVM exists.

	Native Windows `.exe`	Desktop app (JVM)	Executable jar
Maturity	Young — a foundation, with gaps (see `status.md`).	Mature, battle-tested.	Mature, battle-tested.

In practical terms: reach for the **native `.exe`** when you want to ship a single self-contained, JVM-free, architecture-native Windows app that starts instantly; reach for the **desktop (JVM) app** when you want the mature, fully featured desktop experience and don't mind shipping (or depending on) a JVM; and reach for the **jar** when you want the smallest, most portable artifact for machines that already have a JDK. == Working with the native Linux port

The native Linux port compiles a Codename One app to a real, standalone Linux executable — a single ELF binary that runs with **no JVM**. It's the Linux analog of the iOS and native Windows ports: the same ParparVM pipeline that turns your Java/Kotlin bytecode into C and then into a native binary, here targeting Linux and rendering through GTK3 / Cairo / Pango.

This is distinct from packaging the app as an executable jar that runs on a Java Virtual Machine on Linux (see Section 84.13, "Native Linux vs. the executable jar" for a practical comparison). The native port needs no JVM on the target machine.

84.8. How it works (the technology stack)

The port reuses Codename One's portable architecture and swaps in a GTK3 / Cairo implementation layer:

- **ParparVM "clean" C target.** The same VM that powers iOS and Windows translates your app + the Codename One core + the minimal Java runtime to C. A CMake project is generated and compiled with `cc`/Clang + **Ninja** into a native ELF. There is no bytecode interpreter and no JNI — your code **is** the native binary.
- **Cairo + Pango + GdkPixbuf.** All 2D graphics (primitives, gradients, clipping, affine transforms, images) go through Cairo; glyph layout, measurement and rasterization use Pango (with FontConfig/FreeType, so bundled fonts — including the material icon font — register and render); image decode/encode uses GdkPixbuf. Windowing and native peers (text editing, browser, video) are GTK3 widgets layered over the Cairo drawing surface.
- **OpenGL ES (EGL)** backs the portable 3D API (`com.codename1.gpu`); **GStreamer** backs `Media` playback, audio recording and the camera; **WebKitGTK** backs `BrowserComponent`; **libsecret** backs secure storage; **libnotify** backs local notifications; **GeoClue** backs location; **libcurl** and POSIX sockets back networking.
- **Single self-contained executable.** There is no `.app`-style bundle directory, so the app's classpath resources — the theme `.res`, images, localization, the material icon font — are embedded directly into the ELF and read back at runtime through `getResourceAsStream`. The result is one file you copy and run.

84.9. Runs on essentially any Linux desktop

Two choices keep the binary portable, so in practice it just runs:

- **GTK3, dynamically linked.** GTK3 has shipped on every Linux desktop since 2011, so its runtime is already present on virtually any machine with a graphical session, and the binary resolves `libgtk-3`, Cairo and Pango from the system at startup. (GTK4 is newer and not yet universal, so a GTK4 build would be a separate future target.)
- **An old glibc.** The build compiles your app against an old glibc (via `zig cc`), so the resulting ELF needs only an ancient, universally present glibc — in practice around `GLIBC_2.17` (2013) — and starts on essentially any glibc desktop from the last decade, no matter how new the build machine is.

The result is one small ELF that runs out of the box across mainstream distros. Alpine, which uses musl instead of glibc, is an opt-in target — see the `linux.libc` build hint below.

84.10. Building a native Linux app

A native Linux build runs through the `codenameone-maven-plugin` like every other target. Build it with:

```
mvn -pl common package -Dcodename1.platform=linux -Dcodename1.buildTarget=local-linux-device cn1:build
```

The builder translates the app to C with ParparVM's `linux` app type, generates a CMake project that links the GTK stack via `pkg-config`, builds it with CMake
Ninja, and produces the ELF.

84.10.1. Build-host dependencies

The native compile needs the GTK development stack and the capability libraries the port links. On Debian/Ubuntu:

```
sudo apt-get install -y \
  cmake ninja-build pkg-config \
  libgtk-3-dev libcairo2-dev libpango1.0-dev libgdk-pixbuf-2.0-dev libglib2.0-dev \
  libfontconfig1-dev libfreetype-dev libcurl4-openssl-dev \
  libgstreamer1.0-dev libgstreamer-plugins-base1.0-dev \
  libwebkit2gtk-4.1-dev libsecret-1-dev libnotify-dev libgeoclue-2-dev \
  libepoxy-dev libegl1-mesa-dev libgles2-mesa-dev
```

The `-dev` packages are needed on the *build host. The machine that **runs** the binary only needs the corresponding runtime libraries, which any standard desktop already has.

84.10.2. Architecture and cross-compiling

Both **x64** and **arm64** are supported; pick the target with the `linux.arch` build hint. The build is a **native** compile — it produces a binary for the host architecture by default, so the straightforward way to get both is to build on a machine of each.

You can also **cross-compile** — and you may well want to, for example producing an arm64 binary

from a build host that itself runs x64. The translated C is architecture-independent, and the default toolchain (`zig cc`) is itself a cross-compiler, so it can emit a binary for a different CPU from the one it runs on. Point `linux.cc` at a cross `zig cc` wrapper for the target triple (or set `linux.arch` and let the builder generate one) to cross-build.

84.11. Build hints

The native Linux port adds the build hints below (all use the `linux.` prefix).

Table 20. Native Linux port build hints

Name	Description
linux.arch	Target CPU architecture for the ELF: `x64` (default) or `arm64`. The synonyms `x86_64`/`amd64` and `aarch64` are accepted.
linux.debug	`true`/`false` (default `false`). When `false`, the ELF is optimized and **stripped** (no debug info). Set `true` to keep symbols (`RelWithDebInfo`) so a native crash address can be turned back into a function name and source line while developing. Optimizations stay on either way.
linux.libc	`glibc` (default) or `musl`. The default builds with `zig cc` against an old glibc (`-linux-gnu.2.28`), so the binary runs on essentially any mainstream distro. `musl` targets `-linux-musl` for **Alpine** (where the GTK stack is itself musl-built); a musl binary doesn't run on glibc distros and vice versa.
linux.toolchain	Set to `zig` to force the `zig cc` toolchain (the default already uses it for the glibc target). zig is a self-contained cross-compiler, so a single host can build for either `linux.arch` and either `linux.libc`.
linux.cc	Override the C compiler CMake uses (for example a `zig cc` wrapper for a specific target triple, a `musl-gcc`, or a cross `cc`). Takes precedence over `linux.libc`/`linux.toolchain`; also settable through the `CN1_CC` environment variable.

The full cross-platform build-hints reference is in the Advanced topics chapter [# _sending_arguments_to_the_build_server].

84.12. Optimized and stripped by default

The shipping build is optimized and stripped: no debug information is embedded, so the single self-

contained executable stays as small as the translated code allows. Setting `linux.debug=true` keeps the symbols so a faulting address can be symbolized during development; the build stays optimized either way.

84.13. Native Linux vs. the executable jar

Before the native port, the way to run a Codename One app on Linux was to package it as an executable jar that runs on a JVM. The two differ in what ships and how it runs:

	Native Linux port	**Executable jar**
Needs a JVM on the target?	**No** — a native ELF	Yes (any installed JRE)
What ships	One small ELF + the system GTK libraries	A single `.jar`
Rendering	GTK3 / Cairo / Pango (native)	Java2D on the JVM
Startup / footprint	Native, small	JVM startup
How it's built	ParparVM → C → ELF (CMake/Ninja)	`javase` packaging

The native port is the right choice when you want a true native Linux binary with no JVM dependency; the executable jar remains the convenient path for development and for environments where a JVM is acceptable.

84.14. Blocking native calls and the concurrent GC

ParparVM's garbage collector is concurrent: before it can traverse a lightweight (CN1-created) thread's stack it sets that thread's `threadBlockedByGC` flag and spins until the thread parks itself (clears `threadActive`). A native that parks a CN1 thread in a long blocking syscall — a socket `read`/`write`/`connect`, an `curl_easy_perform`, a `sleep` — **must** drop `threadActive` across the call (`CN1_YIELD_THREAD` / `CN1_RESUME_THREAD`), exactly as every other port does. Forget it on even one path (for example the cn1ss screenshot reader parked in `read()` waiting on the server) and a GC that happens to fire during that window never completes, deadlocking every thread waiting on it. The Linux port wraps all of its blocking I/O this way; keep that invariant when adding new native calls.

Chapter 85. Wearables (Apple Watch and Wear OS)

Codename One can build and run your application UI on smartwatches: Apple Watch (watchOS) and Android Wear OS. The same Java/Kotlin code base that drives your phone app drives the watch app — you write Codename One UI as usual, and the build pipeline produces the appropriate watch artifact for each platform.

The two platforms reach the watch through different mechanisms, and understanding the difference explains why the build hints and the supported feature sets differ:

- **Wear OS is Android.** A Wear OS app is an ordinary Android app that declares the watch hardware feature. The existing Codename One Android port renders the UI through exactly the same pipeline it uses on phones and tablets, so almost everything that works on Android works on the watch.
- **watchOS isn't iOS.** watchOS has no UIKit view hierarchy, no OpenGL ES and no Metal. The Codename One watchOS port therefore ships a dedicated **Core Graphics** rendering backend and a separate watch application target that hosts the Codename One runtime (the ParparVM-translated app) inside a SwiftUI shell. The graphics-heavy, GPU-bound and UIKit-peer APIs that have no watchOS equivalent are unavailable on the watch (see [watch-supported-apis]).

In both cases the build is **additive**: with the watch hints turned off your phone build is byte-for-byte unchanged.

85.1. Detecting the Watch Form Factor

Use `CN.isWatch()` (or `Display.getInstance().isWatch()`) to branch your UI for the watch at runtime. This is the wearable analog of the existing `isTablet()` and `isDesktop()` checks:

```
Form f = new Form(BoxLayout.y());
if (CN.isWatch()) {
    // Compact, single-column layout suited to a small round/square screen
    f.add(new Label("Hi Watch"));
    f.getToolbar().setVisible(false);
} else {
    // Full phone/tablet layout
    f.add(new SpanLabel("Welcome to the full size application"));
}
f.show();
```

On Android `isWatch()` is derived from `PackageManager.FEATURE_WATCH` (`android.hardware.type.watch`); on iOS it's reported natively by the watchOS runtime. On every other platform it returns `false`.



`isWatch()` describes the device form factor, not the screen shape. A Wear OS device can be round or square; query the display safe-area insets (see [designing-for-the-watch]) rather than assuming a rectangle.

85.2. Designing for the Watch

A watch screen is small and is frequently round. A few practical guidelines:

- Prefer a single vertical column (`BoxLayout.y()` inside a `Form` that scrolls on the Y axis). The Digital Crown on Apple Watch and the rotary input on Wear OS scroll the focused scrollable container.
- Keep interactive targets large and few. There is little room for a `Toolbar`, side menu or tabs — hide the title area when you don't need it.
- On round screens, keep content away from the corners. Use the form's safe-area insets so important content isn't clipped by the rounded bezel.
- Use the `"watch"` theme/constant overrides if you want a distinct watch look without forking your code (the override layer activates on watch devices the same way platform overrides do elsewhere).



You can lay out and iterate on a watch UI in the simulator by guarding the watch layout with `CN.isWatch()` and exercising both branches; the device build then renders the same code on the real watch.

85.3. Apple Watch (watchOS)

The watchOS build adds a second Xcode target to the generated project. It compiles the shared, translated application sources for the watch architecture (`arm64_32` on device), renders through the Core Graphics backend, and — in the default *companion* distribution — embeds the watch app inside your iOS app so the pair installs together. The watch app is rooted in a generated SwiftUI `@main` shell that hosts the Codename One frames and forwards Digital Crown and tap input into the runtime.



Figure 391. Codename One UI rendered on the watchOS simulator via the Core Graphics backend

85.3.1. Enabling the watchOS Build

Set the build hint:

```
watchNative.enabled=true
```

Alternatively, declare a watch entry point and the watch slice is produced automatically as part of the regular iOS build:

```
codename1.watchMain=com.mycompany.myapp.MyWatchMain
```

If you don't declare a distinct `watchMain`, the watch app reuses your phone main class as its lifecycle entry point.



`codename1.watchMain` (and `watchNative.enabled`) affect only the Apple Watch (watchOS) build. They have no effect on Android: a Wear OS build is never produced implicitly — you enable it explicitly with `android.wear=true` (see [wear-os-android]). A project can target both wearables at once by setting a `watchMain` (or `watchNative.enabled=true`) and `android.wear=true` together.

85.3.2. watchOS Build Hints

Build hint	Default	Description
`watchNative.enabled`	`false`	Force the watch target on even without a distinct `watchMain`.
`codename1.watchMain` (a.k.a. `watchMain`)	*(none)*	Fully-qualified watch lifecycle entry class. Setting it also turns on the watch build.
`watchNative.distribution`	`companion`	`companion` embeds the watch app in the iOS app; `standalone` builds a watch-only app with no paired phone app.
`watchNative.bundleId`	`<package>.watchkitapp`	Bundle identifier of the watch app.
`watchNative.minDeploymentTarget`	`10.0`	`WATCHOS_DEPLOYMENT_TARGET` for the watch target.
`watchNative.displayName`	*(app display name)*	The watch app name shown on the watch.
`watchNative.teamId`	*(falls back to the iOS team id)*	Apple Developer Team ID used to sign the watch target.
`watchNative.embedCompanion`	`false`	Embed the watch app into the iOS app as a build dependency. Off by default so the iOS build is unaffected; enable it for a packaged companion submission.

85.3.3. Supported and Unsupported APIs on watchOS

Because watchOS lacks UIKit views, GPU rendering and several iOS frameworks, the APIs that depend on them aren't available on the watch slice. They're guarded so that the shared sources still link, and they degrade rather than crash:

- **Unavailable:** `BrowserComponent` / web view, camera capture, `MediaPlayer` video, inline native text editors (text input routes through the watch text input controller), `MapComponent` native maps, and StoreKit in-app purchase.
- **Available:** the full Codename One UI and layout system, drawing and `Graphics` (rendered via Core Graphics, including gradients, transforms, clipping and Gaussian blur via Accelerate/vImage), `FontImage`/material icons, images and mutable images, networking and storage, JSON/XML, and the property and binding frameworks.



The watch slice runs the same garbage-collected ParparVM runtime as the iOS app. Memory and CPU budgets on the watch are far smaller than on the phone, so keep watch screens light.

85.3.4. Building and Debugging

A `companion` build produces an iOS `.ipa` that carries the embedded watch app; a `standalone` build produces a watch-only product. The generated project is a standard Xcode project, so you can open it and debug/profile the watch target with the native Xcode tools as usual. Cloud builds support the watch target through the same iOS build — set the hints above and build for iOS.

85.4. Android (Wear OS)

A Wear OS app is a regular Android app. The Codename One Android port renders the UI with the same pipeline it uses on phones, so no special rendering backend is required — you only need to mark the build as a watch app.

85.4.1. Enabling the Wear OS Build

```
android.wear=true
```

This injects the watch hardware feature into the manifest:

```
<uses-feature android:name="android.hardware.type.watch" android:required="true" />
```

By default it also declares the app **standalone**, so it installs and runs directly on the watch without a paired phone app:

```
<meta-data android:name="com.google.android.wearable.standalone" android:value="true" />
```

Setting `android.wear=true` also raises the minimum SDK to API 23 (the Wear OS 2.0 standalone baseline) if your project requests a lower level.

85.4.2. Wear OS Build Hints

Build hint	Default	Description
`android.wear`	`false`	Mark the build as a Wear OS app (manifest feature + standalone meta-data minimum SDK floor).
`android.wear.standalone`	`true`	Declare the app standalone. Set to `false` for a watch app that requires a companion phone app.
`android.playService.wearable`	`false`	Add the `play-services-wearable` dependency (only needed if you use the Wearable Data Layer / message APIs directly).



Because a Wear OS app is an ordinary Android app, you can also declare any additional manifest features and permissions with the generic `android.uses_feature.<name>` and `android.uses_permission.<name>` hints.

85.5. Summary

	Apple Watch (watchOS)	Wear OS (Android)
Enable	`watchNative.enabled=true` or `codename1.watchMain`	`android.wear=true`
Rendering	Dedicated Core Graphics backend + separate watch target	Standard Android rendering pipeline
Distribution	Companion (embedded in iOS app) or standalone	Standalone (default) or companion
Runtime detection	`CN.isWatch()`	`CN.isWatch()`

The wearable build is additive on both platforms: with the hints off, your phone builds are unchanged.

Chapter 86. Television (Apple TV and Android TV)

Codename One can build and run your application UI on the living-room TV platforms: Apple TV (tvOS) and Android TV / Google TV. As with the phone and wearable targets, the same Java/Kotlin code base drives the TV app — you write Codename One UI as usual and the build pipeline produces the appropriate TV artifact for each platform.

The two platforms reach the TV through different mechanisms:

- **Android TV / Google TV is Android.** An Android TV app is an ordinary Android app (the **same APK**) that adds a small amount of manifest metadata: a Leanback launcher category so the app appears on the TV home screen, the `android.software.leanback` feature, and an optional banner. The existing Codename One Android port renders the UI through exactly the same pipeline it uses on phones and tablets.
- **Apple TV runs tvOS.** tvOS is a UIKit relative of iOS reached through the same ParparVM (Java bytecode → C → native) pipeline used for iPhone/iPad, built against the `appletvos` SDK with a tvOS application target. tvOS has no touchscreen — navigation is driven by the Siri remote and the focus engine.

In both cases the build is **additive**: with the TV hints turned off your phone/tablet build is unchanged.

86.1. Detecting the TV Form Factor

Use `CN.isTV()` (or `Display.getInstance().isTV()`) to branch your UI for the TV at runtime. This is the living-room analog of the existing `isTablet()`, `isDesktop()` and `isWatch()` checks:

```
Form f = new Form(BoxLayout.y());
if (CN.isTV()) {
    // 10-foot UI: larger fonts, generous spacing, focus-driven navigation
    f.add(new Label("Hello TV"));
} else {
    // Full phone/tablet layout
    f.add(new Label("Hello"));
}
f.show();
```

Codename One's existing D-pad/arrow focus traversal (the same mechanism that powers `GAME_UP`/`GAME_DOWN`/`GAME_LEFT`/`GAME_RIGHT` and component focus) drives remote navigation on the TV, so focusable components such as `Button` are navigable with the remote out of the box.

86.2. Adapting Styling with CSS Media Queries

The TV (and watch) form factors integrate with the CSS `@media` device-type mechanism. Rules inside a `device-tv` (or `device-watch`) media block are selected at runtime when `Display.isTV()` (respectively

`isWatch()`) returns `true` — exactly the way `device-tablet` / `device-desktop` already work:

```
Label {
    color: black;
}

@media device-tv {
    Label {
        /* Larger type for the 10-foot UI */
        font-size: 3mm;
        color: white;
    }
}
```

86.3. Building for Android TV / Google TV

Enable the Android TV manifest metadata with a single build hint in `codenameone_settings.properties`:

```
codename1.arg.android.tv=true
```

This makes the build:

- add `<category android:name="android.intent.category.LEANBACK_LAUNCHER"/>` to the launcher activity so the app appears on the Android TV home screen;
- declare `<uses-feature android:name="android.software.leanback" android:required="false"/>` and make `android.hardware.touchscreen` optional so the app installs on touchless TVs;
- generate a 320×180 launcher banner (`@drawable/tv_banner`) from the app icon.

The resulting APK still installs and runs on phones and tablets.

86.3.1. Android TV Build Hints

Build hint	Default	Description
`android.tv`	`false`	Mark the build as an Android TV / Google TV app: adds the `LEANBACK_LAUNCHER` intent category, declares the `android.software.leanback` feature, makes `android.hardware.touchscreen` optional, and generates the 320×180 launcher banner. The same APK still runs on phones and tablets.

86.4. Building for Apple TV (tvOS)

Enable the tvOS application target with the `tvNative. build hints (analogous to the watchNative.` hints used for Apple Watch):

```
codename1.arg.tvNative.enabled=true
```

The tvOS build produces a standalone Apple TV app. Because tvOS uses a separate App ID and provisioning profile from your iOS app, you supply tvOS signing material the same way you do for the other Apple targets.



`codename1.tvMain` (and `tvNative.enabled`) only add the tvOS target; the iPhone/iPad build is unchanged. In particular enabling the tvOS target does **not** switch the iOS app's renderer — tvOS runs on Metal (it has no OpenGL ES), but the iOS app continues to honor its own `ios.metal` setting.

86.4.1. tvOS Build Hints

Build hint	Default	Description
`tvNative.enabled`	`false`	Force the tvOS target on even without a distinct `tvMain`.
`codename1.tvMain` (a.k.a. `tvMain` / `tvNative.mainClass`)	*(none)*	Fully-qualified tvOS lifecycle entry class. Setting it also turns on the tvOS build. When omitted (with `tvNative.enabled=true`) the tvOS app reuses your phone main class.
`tvNative.bundleId`	`<package>.tvos`	Bundle identifier of the tvOS app.
`tvNative.minDeploymentTarget`	`13.0`	`TVOS_DEPLOYMENT_TARGET` for the tvOS target.
`tvNative.displayName`	*(app display name)*	The tvOS app name shown on Apple TV.
`tvNative.teamId`	*(falls back to the iOS team id)*	Apple Developer Team ID used to sign the tvOS target.

Chapter 87. Working with Codename One sources

The Codename One SDK is published as a Maven multi-module project. Building the aggregator in the `/maven` directory compiles every module, installs the artifacts into your local Maven repository, and lets you depend on the exact snapshot you cloned from GitHub.

87.1. Prerequisites

Codename One uses multiple JDKs and Maven profiles during the build. Make sure your development machine includes the following tooling (see `BUILDING.md` for platform-specific installation tips):

- JDK 8 (required for the core build of the framework itself)
- JDK 11 through 25 (required at **runtime** to launch the simulator or "Run as desktop app" target)
- JDK 17 (required when compiling the Android port)
- Apache Maven 3.6 or newer
- macOS with Xcode (if you plan to build or test the iOS port)

87.2. Preparing the Maven workspace

Clone the repository and run the Maven aggregator from inside the `maven` subdirectory. This installs the complete SDK (core runtime, simulator, and ports) into your local Maven cache:

```
$ git clone https://github.com/codenameone/CodenameOne
$ cd CodenameOne/maven
$ mvn install
```

The default build runs every module and its unit tests. Append `-DskipTests` if you want to skip the test phases to speed up local builds.

The version printed by Maven will end with `-SNAPSHOT` when you build from the `master` branch. Release builds from Maven Central omit the suffix.

87.3. Installing the Maven archetypes

Codename One ships its project archetypes in a companion repository. Install them once so that `mvn archetype:generate` (and the Codename One Initializr) can reference your locally built SDK artifacts:

```
$ git clone https://github.com/shannah/cn1-maven-archetypes
$ cd cn1-maven-archetypes
$ mvn install
```

87.4. Running tests selectively

The Maven aggregator exposes modules for the different automated test suites. Run any of these from the `CodenameOne/maven` directory:

- Core unit tests:

```
$ mvn -pl core-unittests test
```

- Integration tests (simulator and build client):

```
$ mvn -pl tests -am verify
```

Refer to the READMEs in `maven/core-unittests` and `maven/tests` for more configuration flags, platform requirements, and environment variables.

87.5. Using your local build in application projects

After running the `mvn install` commands above, the Codename One artifacts are available in your local Maven repository. To consume that build in an application generated from the Codename One Initializr (or any other Maven project), adjust the `pom.xml` properties to reference your snapshot version:

```
<properties>
    <!-- Replace with the snapshot printed during mvn install -->
    <cn1.version>{cn1-snapshot-version}</cn1.version>
    <cn1.plugin.version>{cn1-snapshot-version}</cn1.plugin.version>
</properties>
```

Open the project in your IDE and build or run it. Maven will resolve the local snapshot instead of downloading the latest release from Maven Central.

87.6. Why build from source

Building the SDK yourself gives you:

- Immediate access to fixes and features before they reach Maven Central.
- The ability to inspect, debug, and change the framework when you need custom behavior.
- A path to contribute improvements back to the Codename One core.

Once you're comfortable with the baseline build, continue with the scripts in `scripts/` or the `BUILDING.md` guide to compile specific ports (Android or iOS) or to automate CI workflows.

Chapter 88. Skin Designer

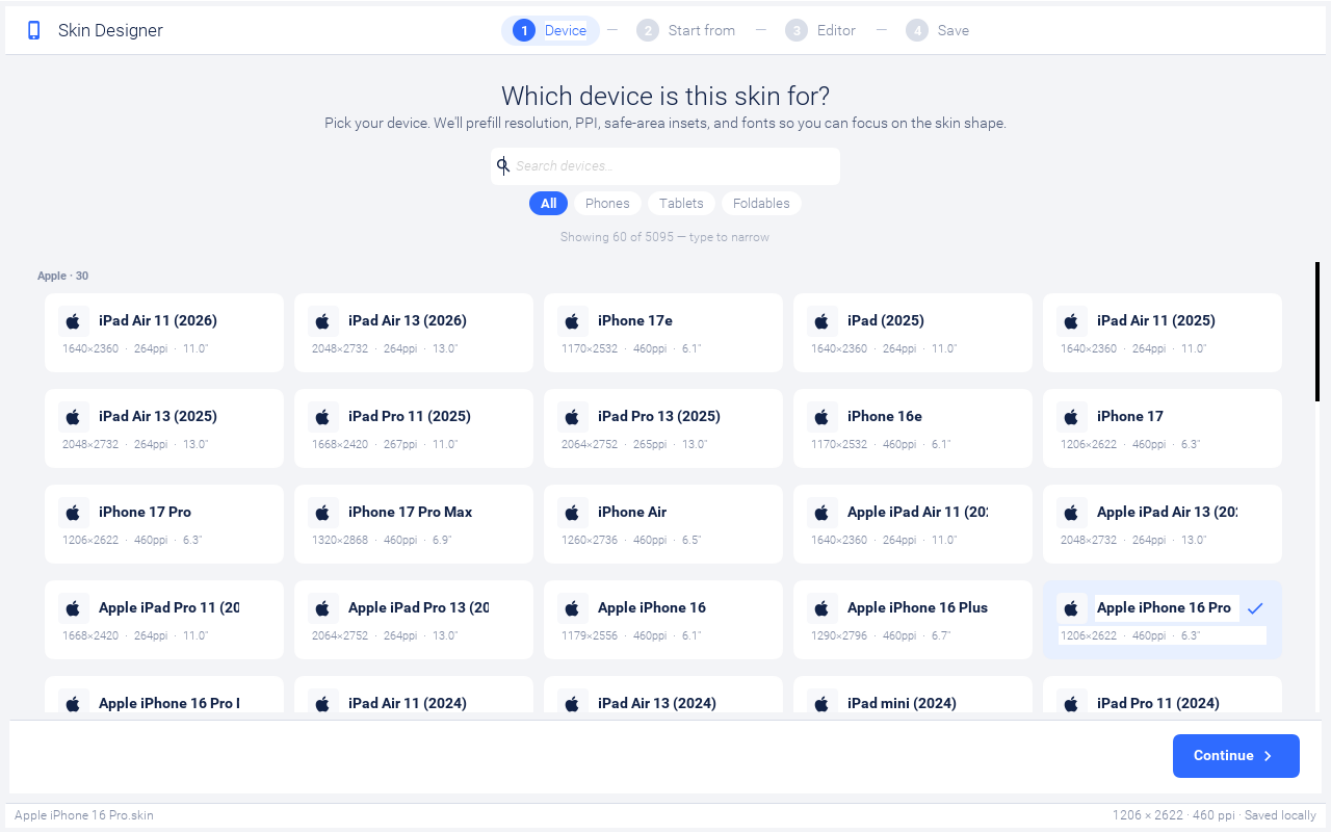
The Skin Designer turns a device specification (resolution, PPI, fonts, safe-area insets, cutouts) into a `.skin` file that the JavaSE simulator can load. It runs in your browser at https://www.codenameone.com/skindesigner — there is nothing to install.

This chapter walks through the wizard and explains what each step contributes to the generated `.skin`. If you only want a skin and don't care how it's built: pick a device, accept the defaults, click **Finish**, then **Download skin**. The file is ready to load via **Add** in the simulator's **Skins** menu.



The wizard is intentionally opinionated. It ships with a curated device catalog, generates the device frame procedurally, and writes a skin layout that matches the `Themes/iPhoneTheme.res`, `Themes/iOS7Theme.res`, and `Themes/android_holo_light.res` themes shipped with Codename One. For a custom skin from a hand-painted device render, jump straight to the "Upload an image" source.

88.1. Stage 1 — pick a device



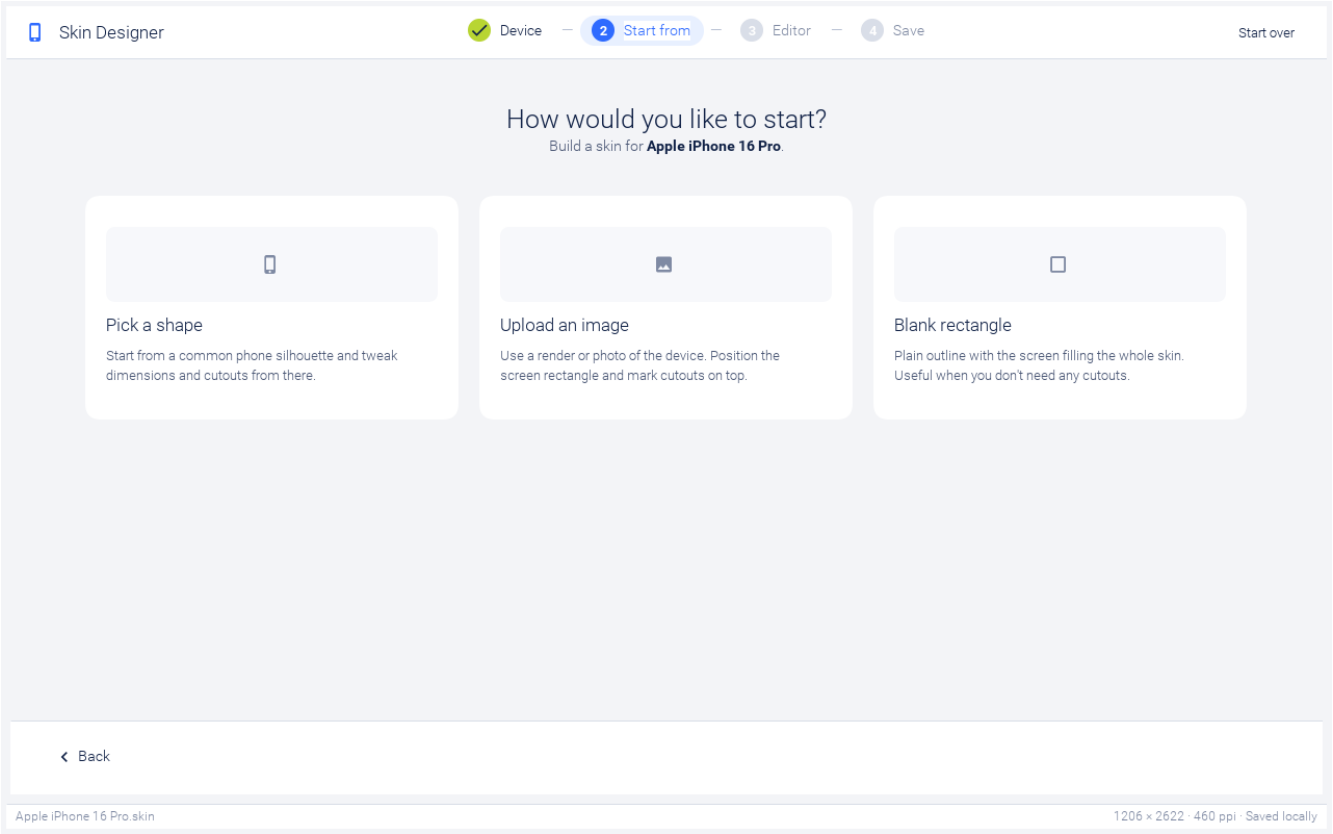
The first step shows a card per device from the bundled catalog. Use the search box to filter by name (it matches both the model and the brand) and the chips below to narrow by form factor: **All** / **Phones** / **Tablets** / **Foldables**. Tapping a card selects the device and enables the **Continue** button in the footer.

Picking a device pulls in its resolution, PPI, screen size, default safe-area insets, and the iOS / Android system font names from the device catalog, then seeds a sensible starting frame: notch, island, or hole presets are applied automatically based on the device's hardware.



The catalog is large. The grid is capped to the most recent matches by default; type into the search field to find older or less-common devices. The `Showing X of Y` hint between the chips and the grid tells you when you're looking at a clamped subset.

88.2. Stage 2 — pick a starting source



Three ways exist to seed the skin's body image:

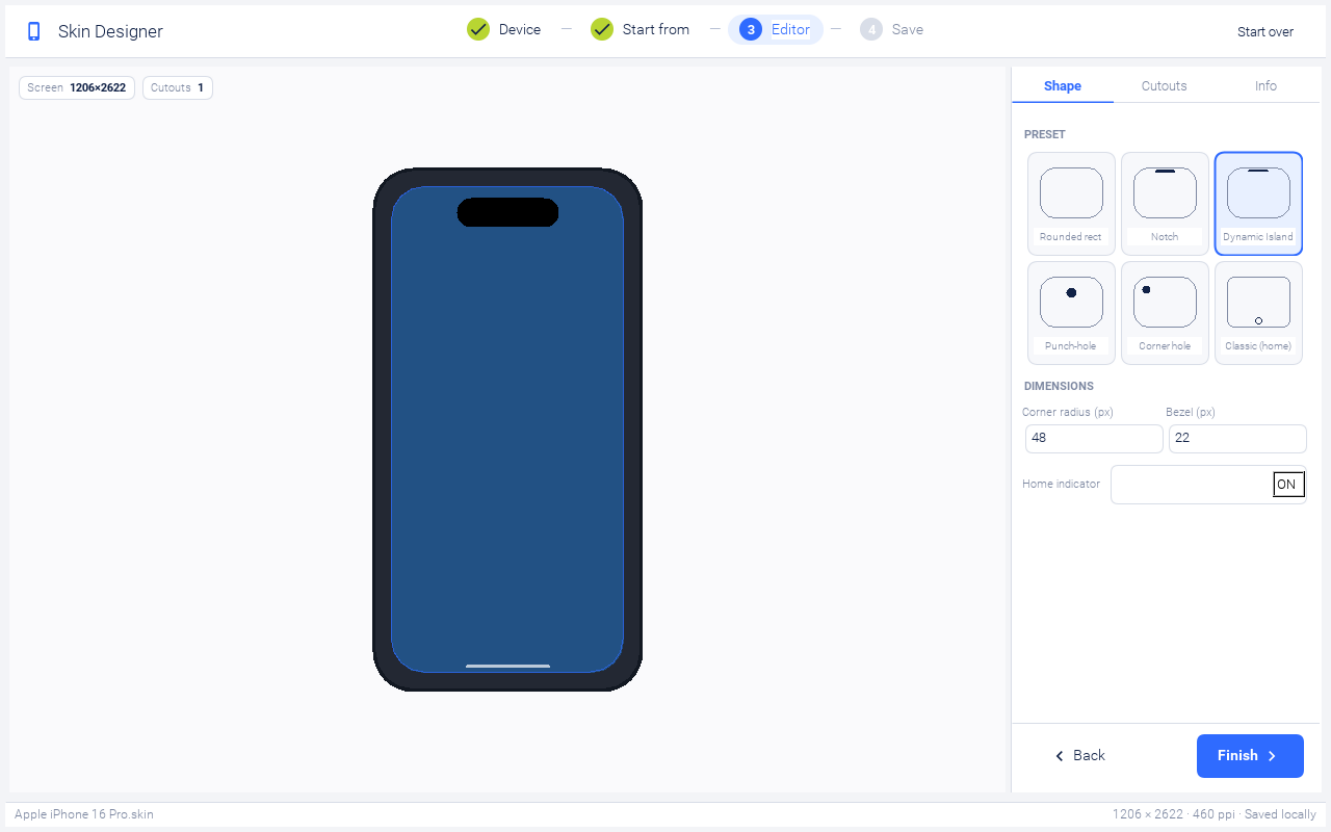
Pick a shape — generates the device frame procedurally from a small preset library (rounded rect, notch, dynamic island, punch-hole, corner hole, classic home-button). The frame is rendered as a dark gradient with the screen rect (and any cutouts) carved into it. Best when you want a generic-looking iPhone or Android frame and don't care about exact hardware fidelity.

Upload an image — opens an image picker. The wizard scales the image into the device's resolution, then carves the screen rect (and cutouts) on top. Use this when you have a marketing render of the specific device you're targeting.

Blank rectangle — collapses the bezel and corner radius to almost nothing, drops every cutout, and turns the home indicator off. The screen fills the entire skin. Useful for desktop / web simulators where the device frame would just be visual noise.

Hover over a card on desktop to see the blue selection ring; click to advance to the editor.

88.3. Stage 3 — the editor



The editor is split into two panes: a live preview on the left that paints the device frame, screen tint, cutouts, and home indicator, and a sidebar on the right with three tabs.

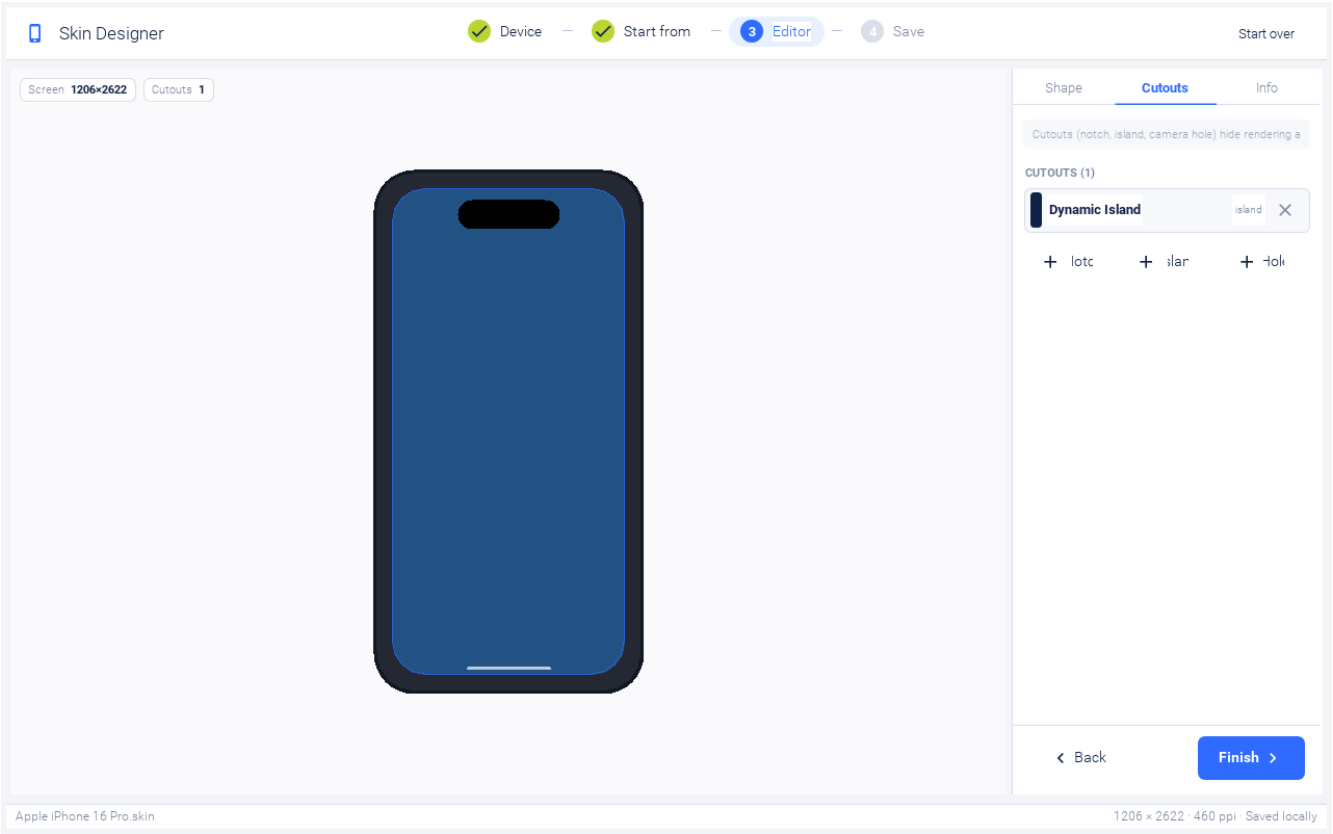
88.3.1. Shape tab

When you started from **Pick a shape**, the Shape tab shows a preset grid (**Rounded rect** / **Notch** / **Dynamic Island** / **Punch-hole** / **Corner hole** / **Classic home**) and dimension fields:

- **Corner radius** — the device's outer corner radius in viewbox pixels (the design uses a 320 × 620 viewbox; the actual skin image is scaled to the device's resolution).
- **Bezel** — frame thickness in viewbox pixels. The screen rect is positioned at `(bezel, bezel)` inside the skin image.
- **Home indicator** — toggles the lime pill drawn at the bottom edge. iPhones from X onward and most modern Androids should leave this on; classic devices with a hardware home button should turn it off.

When you started from **Upload an image**, the Shape tab swaps the preset grid for a **Replace image** button so you can swap the body image without restarting the wizard.

88.3.2. Cutouts tab



The Cutouts tab lists every cutout currently on the skin. Tap a row to expand its width / height / offset fields; the X button on the right removes it. The three add buttons at the bottom seed a sensible default of each type:

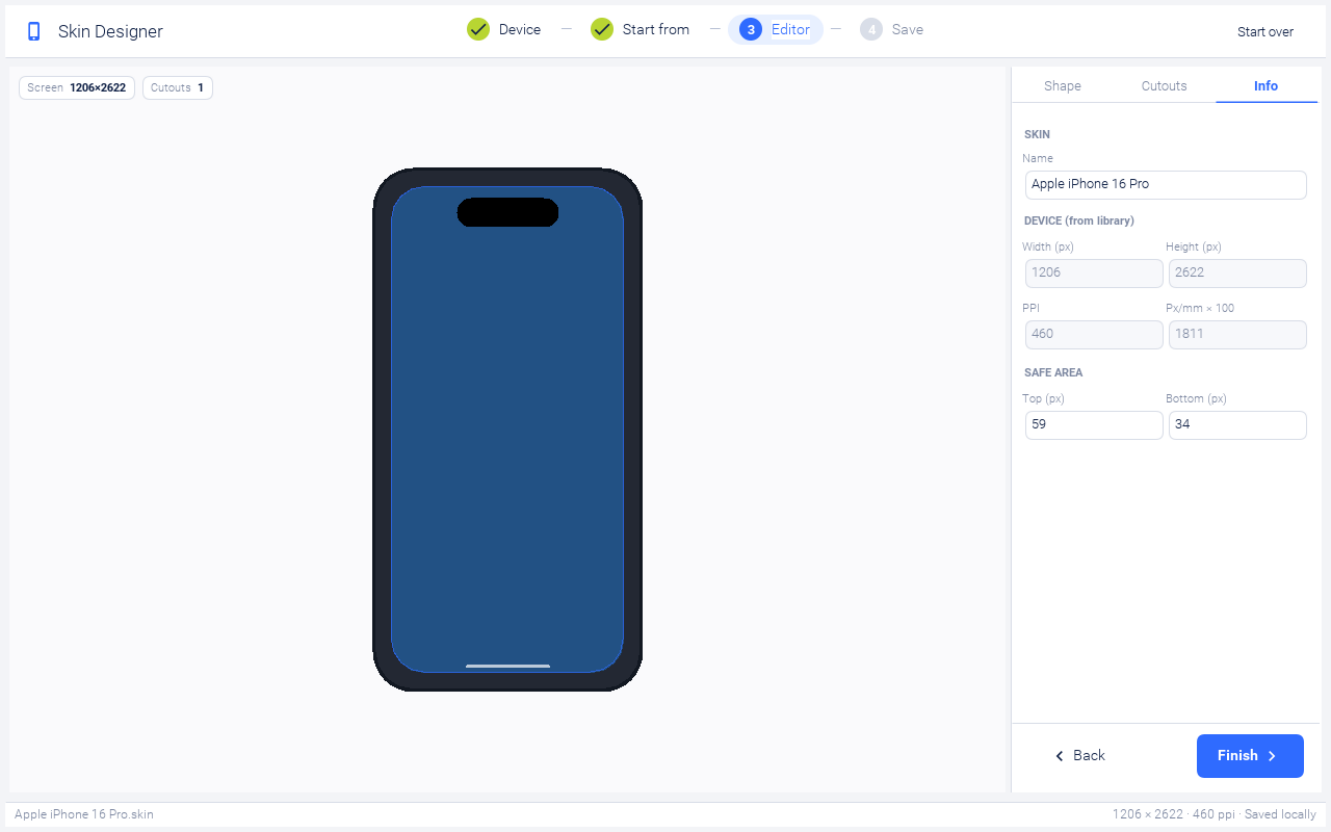
- **Notch** (180 × 30 viewbox px) — physical hardware cutout. Notches are drawn in the device frame **above** the screen rect, with the bottom edge touching the screen top. Mirrors iPhone X / 11 / 12 / 13 hardware.
- **Island** (120 × 35) — Dynamic Island. Software-reserved space rendered as an opaque pill **inside** the screen rect, floating on top of the iOS status bar.
- **Hole** (28 × 28) — Android punch-hole camera. Rendered like the island: opaque circle inside the screen rect.

Offsets are relative to the top-center of the screen, in viewbox pixels. For islands and holes, `Offset Y` is the gap between the cutout's top edge and the screen top. For notches, `Offset Y` is ignored (the notch is anchored to the frame).



When the wizard generates the `.skin` file, it automatically extends `safePortraitTop` to cover any in-screen cutouts (islands, holes), so app content lands below the floating shape. You don't need to manually adjust safe area for the cutout itself; only adjust it if you need extra padding above your title bar.

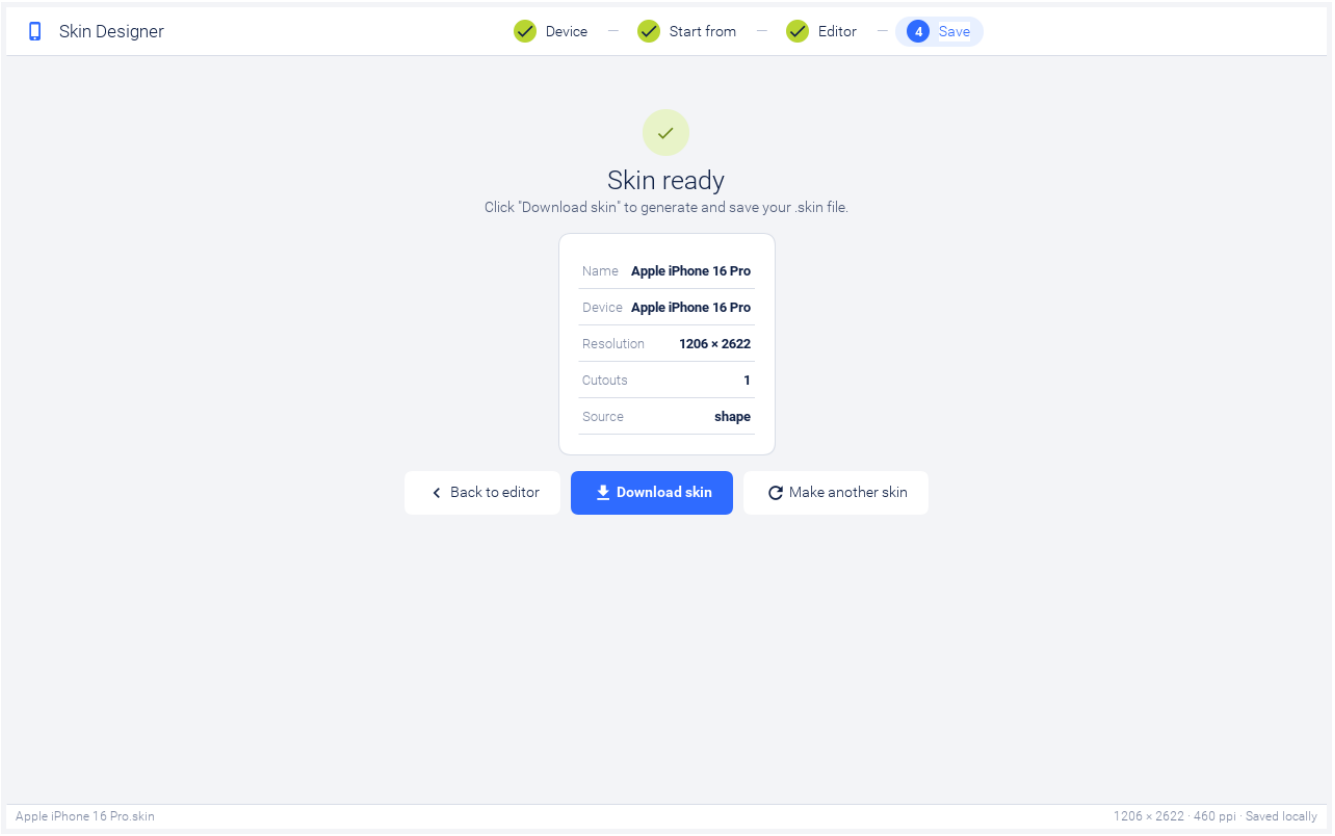
88.3.3. Info tab



The Info tab is mostly read-only and shows what's about to be written into `skin.properties`:

- **Name** — the only editable field. Determines the saved file name (sanitised: spaces become hyphens, special characters dropped, appended with `.skin`).
- **Width** / **Height** — display resolution from the device catalog.
- **PPI** / **Pixels per millimeter** — the simulator uses `pixelMilliRatio = ppi / 25.4` to derive font sizes when none are specified in the skin.
- **Safe area Top** / **Bottom** — the user-editable safe-area insets in viewbox pixels. The wizard converts these to display-relative pixels at save time.

88.4. Stage 4 — finish and download



Clicking **Finish** in the editor's footer:

1. Renders the portrait skin image (`skin.png`) at the device's actual resolution + bezel, with rounded corners, transparent screen, opaque cutouts, and a home indicator if enabled.
2. Synthesises the landscape skin (`skin_l.png`) by 90° rotation.
3. Writes `skin_map.png` / `skin_map_l.png` overlays that mark the screen rectangle for the simulator's screen-position detection.
4. Bundles the appropriate native theme (`iOS7Theme.res`, `android_holo_light.res`, or `winTheme.res`) inside the skin zip.
5. Writes `skin.properties` with `roundScreen=true`, `displayX/Y/Width/Height`, `safePortrait*` / `safeLandscape*`, the platform name, override names, fonts, and PPI.

Clicking **Download skin** on the done page hands the file to the browser's download dialog. The button is the only reliable trigger because the browser's user-gesture window expires during the heavy image-generation step that **Finish** runs.

After the file is on disk, drop it into your simulator's skins folder (or use the **Add** command in the simulator's **Skins** menu) and your new device should appear in the picker.

88.5. File layout of a generated skin

A generated `.skin` is just a renamed zip. Unzip it to see the layout:

```
Apple-iPhone-16-Pro.skin/
  skin.png            # portrait body (device frame + transparent screen + cutouts)
  skin_l.png          # 90° rotated portrait
```

```
skin_map.png        # black rect = screen, white = frame, used for hit-testing
skin_map_l.png      # rotated map
iOS7Theme.res       # bundled native theme (or android_holo_light.res / winTheme.res)
skin.properties     # platform metadata, safe-area, PPI, display rect
```

The `skin.properties` file is a normal Java `.properties`. The most important keys the wizard writes:

```
touch=true
platformName=ios
tablet=false
ppi=460
pixelRatio=18.110236220472443

# roundScreen=true makes the simulator paint skin.png *over* the rendered
# UI rather than clipping the UI to non-frame pixels. That overlay step is
# what makes the Dynamic Island / punch-hole shapes appear "floating" on
# top of the iOS status bar instead of being carved out of the display.
roundScreen=true
displayX=89
displayY=89
displayWidth=1206
displayHeight=2622

# Safe area in display-relative coordinates (origin = screen top-left).
safePortraitX=0
safePortraitY=237
safePortraitWidth=1206
safePortraitHeight=2272
safeLandscapeX=237
safeLandscapeY=0
safeLandscapeWidth=2272
safeLandscapeHeight=1206

# These get composed by overrideNames(device) — they let users layer
# device-specific styling on top of the platform theme via
# UIManager.addThemeProps()-style theme inheritance.
overrideNames=phone,ios,iphone

systemFontFamily=SF Pro
proportionalFontFamily=SF Pro
monospaceFontFamily=SF Mono
```



The wizard intentionally does **not** write `smallFontSize`, `mediumFontSize`, or `largeFontSize`. When those are absent the simulator automatically derives them from `pixelMilliRatio` (`med = round(2.6 * ppmm)`, `sm = 2 * ppmm`, `la = 3.3 * ppmm`), which is what you want on high-PPI screens. Writing iOS-style point values (12 / 15 / 22) into the file used to make text render at sub-millimeter sizes on a 460 PPI device.

Appendix A: Project archetypes

A.1. Codename One application project archetype (cn1app-archetype)

The `cn1app-archetype` is the basis for all maven Codename One application projects. It provides a multimodule project with the following modules:

See Getting Started with the Bare-Bones Java App Template [https://shannah.github.io/cn1-maven-archetypes/cn1app-archetype-tutorial/getting-started.html] for details on using this archetype.

Also see Section 1.2.2.2.1, “Bare-bones Java project” for an example using this archetype from the command-line to create a bare-bones Java project.

A.2. Codename One library project archetype (cn1lib-archetype)

The `cn1lib-archetype` provides a project for building Codename One libraries. See Section 1.5, “Codename One libraries” for details on using this archetype.

Appendix B: Maven goals

B.1. Codename One build (`build`)

The `build` goal is used to send builds to the Codename One build server. It also supports a few local build targets, such as `ios-source`, which generates an Xcode project, and `android-source` which generates an Android Gradle project.

This goal is bound to the `package` phase of the Section A.1, "Codename One application project archetype (cn1app-archetype)", so you don't need to run this directly.

B.1.1. Example

Listing 26. Submitting a Mac Desktop build directly using this goal:

```
mvn cn1:build -Dcodename1.platform=javase -Dcodename1.buildTarget=mac-os-x-desktop
```

B.1.2. Properties

codename1.platform

Specifies the platform to build for. Values include Java SE, android, ios, JavaScript, and win.

codename1.buildTarget

The build target. Different platforms support different build sets of build targets.

Build Target	Platform	Description
mac-os-x-desktop	Java SE	Sends a Mac desktop build to the Codename One build server.
windows-desktop	Java SE	Sends a Windows desktop build to the Codename One build server.
android-device	android	Sends an Android build to the Codename One build server.
android-source	android	Generates an Android Gradle project locally that can be opened and built in Android Studio.
ios-device	ios	Sends an iOS build to the Codename One build server.
ios-source	ios	Generates an iOS Xcode project locally that can be opened and built in Xcode.

Build Target	Platform	Description
JavaScript	JavaScript	Sends a JavaScript build to the Codename One build server.

automated

Set to `true` to submit build as an automated build. When using this flag, the goal will wait for the build server to complete the build, then download it and save it in the target directory (of the associated module) using standard maven artifact file naming conventions. This allows you to set up automated CI workflows more.

Default value is `false`.

Requires a Codename One enterprise subscription.

open

If set to `true`, this will automatically open the generated Gradle or Xcode project in Android studio or Xcode. Only applicable to the `ios-source` and `android-source` buildTargets.

Default is `false`

B.2. Generate legacy cn1lib (`cn1lib`)

This goal generates a legacy.cn1lib file. It's bound to the `package` phase of `cn1lib-archetype` projects so you will never need to run this goal directly.

B.2.1. Output location

This will output the cn1lib file inside the common/target directory of the root multimodule maven project.

B.3. Compliance check (`compliance-check`)

Checks API usage in the project's "common" module to ensure that it doesn't use any classes or methods that aren't supported by Codename One.

This goal is bound to the `process-classes` phase of both Section A.1, "Codename One application project archetype (cn1app-archetype)" projects and Section A.2, "Codename One library project archetype (cn1lib-archetype)" projects, so you shouldn't ever need to run this goal directly.

B.4. Compile CSS (`css`)

Compiles the project's CSS files, generating a `theme.res` file which is placed in the build/classes directory.

This goal is bound to the `process-classes` phase of Section A.1, "Codename One application project archetype (cn1app-archetype)" projects, so you shouldn't need to run this goal directly.

B.5. Generate app project (`generate-app-project`)

Generates a Maven project using the Section A.1, “Codename One application project archetype (cn1app-archetype)” as a basis, and applying an optional project template. This goal can also be used to migrate legacy Codename One Ant-based Application projects into maven.

B.5.1. Usage example

This goal shouldn’t be run inside an existing Maven project directory. It will output a project into a directory named after the `artifactId` parameter.

Because there is no existing project, you will need to provide the full maven path to the goal:

```
mvn com.codenameone:codenameone-maven-plugin:$CN1VERSION:generate-app-project \
  -DsourceProject=/path/to/my/ProjectTemplate \
  -DgroupId=com.example \
  -DartifactId=myapp \
  -Dcn1Version=$CN1VERSION
```



Substitute the latest Codename One plugin version [https://search.maven.org/artifact/com.codenameone/codenameone-maven-plugin] for the `$CN1VERSION` in the above example.



This command is formatted for the bash prompt (for example, Linux or Mac). It will work on Windows also if you use bash. If you’re on Windows and are using PowerShell or the regular command prompt, then you’ll need to modify the command slightly. In particular, the entire command would need to be on a single line. (Remove the '\' at the end of each line, and merge lines together, with space between the command-line flags)

In the above example, assuming all went well, it would output your project into a directory named `myapp`.

B.5.2. Parameters

cn1Version

The Codename One version that you want to use for the project. This will be manifested as the `cn1.version` and `cn1.plugin.version` properties in the common/pom.xml file of the generated project. If omitted it will default to the cn1Version that’s hard-coded in the cn1app-archetype artifact.

sourceProject

The path to an optional project template to use. This may be either a legacy Ant project, or a Maven project that follows the structure of Section A.1, “Codename One application project archetype (cn1app-archetype)”.

artifactId

The `artifactId` to use for the new project.

groupId

The `groupId` to use for the new project.

version

The `version` to use for the new project. Optional.

Default value is "1.0-SNAPSHOT"

packageName

The package name to use for the new project.

This is necessary if the `sourceProject` property is a Maven project. If the `sourceProject` is a legacy Ant project, then this property is ignored.

mainName

The main class name to use for the new project.

This is necessary if the `sourceProject` property is a Maven project. If the `sourceProject` is a legacy Ant project, then this property is ignored.

B.5.3. Migrating legacy Ant projects into Maven

When providing an Ant-based Codename One application project as the `sourceProject` parameter, this goal will generate an equivalent Maven project to the Ant project, with the same settings and sources.

See Section 1.2.3, “Migrating an existing project” for examples using this goal to migrate an existing project into Maven.

B.5.4. Project templates

When using a Section A.1, “Codename One application project archetype (cn1app-archetype)” maven project as the `sourceProject` parameter, the project will be treated project template, and perform some basic processing of the source files as necessary convert the template into a **real** project. This includes replacing all occurrences of `${mainName}` and `${packageName}` in project sources with the value of the `mainName` and `packageName` parameters provided on the command-line.

Any occurrences of `mainName` and `packagePath` in file or directory names will be swapped with the values of `mainName` and `packagePath` (which is automatically derived from `packageName` by substituting '.' with file separators).

The `generate-app-project.rpf` file

The (optional) secret sauce that differentiates a regular Maven project from a Maven project template is the existence of a `generate-app-project.rpf` file in the root project directory. This file is in rich property file format, and allows you to define a minimal amount of configuration details

that the `generate-app-project` goal needs to convert the template into a real project.

This file may contain the following properties:

template.mainName

The name of the main class that's used in this project. This property isn't required if the project is already using the `mainName` placeholder in the file name that contains the main class, and the `${mainName}` placeholder in any source code referring to the main class.

Often times it's easier to specify this property here rather than injecting placeholders into the template source base, because that way the template itself can be used as a valid project.

template.packageName

The package name for the app. This property isn't required if the project is already using the `packagePath` placeholder in directories containing your main package files, and the `${packageName}` placeholder in any source code that refers to the main package.

Often times it's easier to specify this property here rather than injecting placeholders into the template source base, because that way the template itself can be used as a valid project.

template.type

Either `maven` or `ant`, depending on the project type.

dependencies

An XML snippet containing any more Maven dependencies that should be added to the project. This is handy of the project template relies on other cn1libs that are on Maven central.

Sample `generate-app-project.rpf` file

```
template.mainName=MyApp
template.packageName=com.example
template.type=maven

[dependencies]
====
<dependency>
  <groupId>com.codenameone</groupId>
  <artifactId>googlemaps-lib</artifactId>
  <version>1.0.1</version>
  <type>pom</type>
</dependency>
====
```

B.5.5. Sample: The bare-bones Kotlin app project

As a fuller example of a project template, see the generate-app-project.rpf [https://github.com/shannah/cn1app-archetype-kotlin-template/blob/master/generate-app-project.rpf] file in the bare-bones Kotlin app template [https://github.com/shannah/cn1app-archetype-kotlin-template].

This is the template that's used in Codename One initializr [https://www.codenameone.com/initializr] for the Bare-bones Kotlin project.

B.6. Clone project (`clone`)

Clones the current project as a new project with a different *groupId*, *artifactId*, *packageName*, and *mainName*. Resulting project will be made available in the *common/target/generated-sources/cn1-cloned-projects* directory.

B.6.1. Usage example

```
mvn cn1:clone \
  -DgroupId=com.example.newgroup \
  -DartifactId=newapp \
```



This command is formatted for the bash prompt (for example, Linux or Mac). It will work on Windows also if you use bash. If you're on Windows and are using PowerShell or the regular command prompt, then you'll need to modify the command slightly. In particular, the entire command would need to be on a single line. (Remove the '\' at the end of each line, and merge lines together, with space between the command-line flags)

In the above example, assuming all went well, it would output your project into a directory named `myapp`.

B.6.2. Parameters

artifactId

The `artifactId` to use for the cloned project.

groupId

The `groupId` to use for the cloned project.

B.7. Create GUI Form (`create-gui-form`)

The `create-gui-form` goal will generate a GUI form that can be edited in the GUI builder.

Basic Usage Example

```
mvn cn1:create-gui-form -DclassName=com.example.MyForm
```

The above example will generate a GUIBuilder form with the provided class name. It effectively generates two files:

1. `common/src/main/guibuilder/com/example/MyForm.gui`

2. `common/src/main/java/com/example/MyForm.java`



The paths above assume that this goal was run from the root multimodule maven project. If it were run inside the "common" submodule, then the files would be generated in the "src/main/..." directory of that module. (It would try to create yet another "common" submodule). See Project Structure [https://shannah.github.io/cn1-maven-archetypes/cn1app-archetype-tutorial/getting-started.html#_project_structure].

You can then open the GUI builder to edit this form using the `cn1:guibuilder` goal:

```
mvn cn1:guibuilder -DclassName=com.example.MyForm
```

B.7.1. Parameters

className

(Required) The class name of the form that you wish to generate. E.g `com.example.MyForm`.

guiType

The kind of GUI component to generate. Supports `Form`, `Dialog`, and `Container`. Default value is "Form"

autoLayout

Whether to use autolayout. This is boolean (true/false), and the default value is `true`.

B.8. Generate cn1lib project (`generate-cn1lib-project`)

Generates a Maven cn1lib project (using the cn1lib-archetype) given an ANT cn1lib project as a template.

This is to assist in migrating Ant projects to Maven projects. This won't make any changes to the source Ant project. It generates a new project using the Section A.2, "Codename One library project archetype (cn1lib-archetype)" and copies all files and configuration from the source project, into the new project.

B.8.1. Usage example



This goal isn't run in an existing maven project directory, so you need to include the absolute Maven coordinates for the goal.

Suppose you have an Ant Codename One library project at `/path/to/MyLegacyAntLibraryProject` and you want to convert it to a Maven project.

You can run the `generate-cn1lib-project` goal as follows:

Listing 27. Maven command to migrate an Ant project to Maven:

```
mvn com.codenameone:codenameone-maven-plugin:$CN1VERSION:generate-cn1lib-project \
```

```
  -DsourceProject=/path/to/MyLegacyAntLibraryProject \
  -DgroupId=com.example \
  -DartifactId=my-maven-lib \
  -Dversion=1.0-SNAPSHOT \
  -U
```



Make sure you substitute the latest codenameone-maven-plugin version [https://search.maven.org/search?q=a:codenameone-maven-plugin] for the `$CN1VERSION` in this command.

Some notes about this command as shown above:

1. This command is formatted for Unix/Mac on multiple lines, using the `\` character to escape the new-lines. On Windows the command will need to be all one line, and you should omit those `\` escape characters.
2. The `-U` flag tells Maven to update its catalogs to ensure that it can find the `$CN1VERSION` that you specify.

If all goes well, you should find a new maven project generated in the "your-maven-lib" directory (named after the `artifactId` that you specified).

To test that the project was generated successfully, try opening the resulting project in your IDE or run its "install" goal on the command-line.

For example:

```
cd my-maven-lib
mvn install
```

After running the "install" command, you should be able to add your library as a dependency to a Section A.1, “Codename One application project archetype (cn1app-archetype)” project using the following dependency:

```
<dependency>
  <groupId>com.example</groupId>
  <artifactId>my-maven-lib-lib</artifactId>
  <version>1.0-SNAPSHOT</version>
  <type>pom</type>
</dependency>
```



Notice that the `artifactId` has an extra "-lib" appended. That is, it's `<artifactId>my-maven-lib-lib</artifactId>` and not `<artifactId>my-maven-lib</artifactId>`. This is because the `artifactId` that you specify in the `generate-cn1lib-project` goal is used for the "root" module of the multimodule maven project. The actual "lib" project that you can use as a Maven dependency is the "lib" submodule, which uses the specified artifactId with a "-lib" suffix.

See Section 1.5, “Codename One libraries” for more information about the resulting maven library project.

B.8.2. Properties

sourceProject

The path to the legacy Ant project that you want to convert to a Maven project.

groupId

The maven `groupId` to use for the resulting project.

artifactId

The maven `artifactId` to use for the resulting project.

version

The maven `version` to use for the resulting project. Default "1.0-SNAPSHOT"

B.9. Generate desktop app wrapper (`generate-desktop-app-wrapper`)

Generates the bootstrapping wrapper class for the Java SE desktop app. This is used by the `Java SE` module of Section A.1, “Codename One application project archetype (cn1app-archetype)” projects.

You shouldn’t need to call this goal directly.

B.10. Generate GUI sources (`generate-gui-sources`)

Generates Java sources from the Codename One GUI builder files. This goal is bound to the `process-sources` phase of Section A.1, “Codename One application project archetype (cn1app-archetype)” projects, so you should never need to run this goal explicitly.

B.11. Generate native interfaces (`generate-native-interfaces`)

Generates stub implementations for all native interfaces defined in the project. This won’t overwrite any existing implementations that may exist.

You should run this goal explicitly after you create a native interface in your class.

See the Codename One Developer guide section on native interfaces [https://www.codenameone.com/developer-guide.html#_native_interfaces] for more information on creating native interfaces.

B.11.1. Usage example

Suppose you’ve created a native interface as the Java interface `com.mycompany.myapp.MyNative`, as described in the example in native interfaces [https://www.codenameone.com/developer-guide.html#

_native_interfaces].

After creating this (and possibly other) native interfaces in your project, run the `generate-native-interfaces` Maven goal as follows:

```
mvn cn1:generate-native-interfaces
```

By default this generates Java/Objective-C stubs. You can optionally include Swift and Kotlin stubs (both off by default):

```
mvn cn1:generate-native-interfaces \
  -Dcn1.generateNativeInterfaces.swift=true \
  -Dcn1.generateNativeInterfaces.kotlin=true
```

This will generate the following files (if they don't exist yet).

Java SE

`Java SE/src/main/java/com/mycompany/myapp/MyNativeImpl.java`

ios

1. `ios/src/main/objectivec/com_mycompany_myapp_MyNativeImpl.h`
2. `ios/src/main/objectivec/com_mycompany_myapp_MyNativeImpl.m`

android

`android/src/main/java/com/mycompany/myapp/MyNativeImpl.java`

android (optional Kotlin)

`android/src/main/java/com/mycompany/myapp/MyNativeImpl.kt`

JavaScript

`JavaScript/src/main/JavaScript/com_mycompany_myapp_MyNativeImpl.js`

ios (optional Swift)

`ios/src/main/objectivec/com_mycompany_myapp_MyNativeImpl.swift`

Open and edit these files to implement your native interface methods as desired.

B.12. Generate OpenAPI client (`generate-openapi`)

Generates a typed Codename One client from an OpenAPI 3.x JSON specification. Writes one `@Mapped` record (Java 17+) or class (Java 8 target) per `components.schemas` entry and one `@RestClient` -annotated interface per OpenAPI tag. The generated files land in `common/src/main/java` so the project owns the contract; the matching networking implementation is emitted into `common/target/generated-sources` by the build-time annotation processor so the project source stays clean.

The mojo is paired with the existing `process-annotations` pipeline: identical schemas across

operations collapse to one record/class, and operationIds become interface methods that resolve at runtime via a `com.codename1.io.rest.RestClients` registry populated by the generated bootstrap class (the same splice pattern as `@Mapped` mappers).

B.12.1. Usage example

```
mvn -pl common cn1:generate-openapi \
    -Dcn1.openapi.spec=petstore.json \
    -Dcn1.openapi.basePackage=com.example.petstore
```

Configuration:

Property	Description
`-Dcn1.openapi.spec=PATH`	Local file path or URL of the OpenAPI 3.x JSON document, for example `petstore.json` or `https://petstore3.swagger.io/api/v3/openapi.json`. YAML isn't supported — convert with `yq` upstream.
`-Dcn1.openapi.basePackage=PKG`	Java package the generated sources are written under. Records / classes go under `<basePackage>.model`; the `@RestClient` interfaces go under `<basePackage>`.
`-Dcn1.openapi.outputDirectory=DIR` (optional)	Defaults to `${project.basedir}/src/main/java`.
`-Dcn1.openapi.overwrite=false` (optional)	Defaults to `true`. Set to `false` to preserve user edits to existing files (only missing files are written).

B.12.2. Generated output

For the Swagger Petstore reference spec the goal emits, under `common/src/main/java`:

```
com/example/petstore/
  PetApi.java          // @RestClient interface, methods addPet, updatePet,
                       // findPetsByStatus, getPetById, deletePet, ...
  StoreApi.java        // @RestClient interface, methods getInventory,
                       // placeOrder, getOrderById, deleteOrder
  UserApi.java         // @RestClient interface
com/example/petstore/model/
  Pet.java             // @Mapped record (Java 17+) or class (Java 8)
  Order.java
  User.java
  Category.java
  Tag.java
```

Each `@RestClient` interface method is annotated with the HTTP verb and path; parameters are annotated `@Path` / `@Query` / `@Header` / `@Body` so the processor knows how to assemble the `Rest` call. The emitted method shape:

```
@RestClient
public interface PetApi {

    @GET("/pet/{petId}")
    void getPetById(@Path("petId") Long petId,
                    @Header("Authorization") String bearerToken,
                    OnComplete<Response<com.example.petstore.model.Pet>> callback);

    @POST("/pet")
    void addPet(@Body com.example.petstore.model.Pet body,
                @Header("Authorization") String bearerToken,
                OnComplete<Response<com.example.petstore.model.Pet>> callback);

    static PetApi of(String baseUrl) {
        return RestClients.create(PetApi.class, baseUrl);
    }
}
```

The model type referenced by the generated API is emitted into the generated model package:

```
public class Pet implements PropertyBusinessObject {
    public final Property<Long, Pet> id = new Property<Long, Pet>("id");
    public final Property<String, Pet> name = new Property<String, Pet>("name");
    private final PropertyIndex index = new PropertyIndex(this, "Pet", id, name);

    @Override
    public PropertyIndex getPropertyIndex() {
        return index;
    }
}
```

Model types are emitted as fully qualified names (the API interface lives in `<basePackage>` and models under `<basePackage>.model`) so the generator never needs to track imports or worry about collisions between an API class name and a same-named model.

Call sites use the static factory:

The `<Tag>ApiImpl` class that actually performs the HTTP call lives in `target/generated-sources` — the project source never references it directly. The build server probes the project zip for the generated `cn1app.RestClientBootstrap` and splices the registry wiring in, mirroring the existing `cn1app.MapperBootstrap` pattern.

B.12.3. Scope

- HTTP verbs: `GET`, `POST`, `PUT`, `DELETE`, `PATCH`.
- Parameter locations: `path`, `query`, `header`, `cookie`. Multiple `cookie` parameters on the same operation are joined into a single `Cookie: a=1; b=2` request header.

- Request bodies: `application/json` — serialized via `Mappers.toJson(body)` before being attached.
- Response schemas: `$ref` resolution, primitives (`string` / `number` / `integer` / `boolean`), arrays, object schemas. `oneOf` / `anyOf` / `allOf` collapse to `Object` — callers cast.
- String `enum` schemas are emitted as Java enums and bound by the JSON mapper through each constant's `name()`, so a model property typed by an enum `$ref` gets the generated enum type. This requires every enum value to be a valid (non-reserved) Java identifier; an enum whose values are not — for example `"two-day"` — degrades to `String` rather than generating an enum that couldn't round-trip. Integer/number enums keep their numeric type.
- Schema unification: two `components.schemas` entries with identical property shapes collapse to a single record/class to avoid an explosion of duplicates.
- Authentication: bearer token is exposed as a `@Header("Authorization") String bearerToken` parameter on every operation. API-key auth declared in the spec is emitted as `@Header` or `@Query` like any other parameter; basic auth and OAuth bearer tokens travel through the same `bearerToken` slot.

B.13. Generate gRPC client (`generate-grpc`)

Generates a typed Codename One gRPC-Web client from a proto3 `.proto` specification. Writes one `@ProtoMessage` record (Java 17+) or class (Java 8 target) per `message`, one `@ProtoEnum` enum per `enum`, and one `@GrpcClient`-annotated interface per `service`. The generated files land in `common/src/main/java` so the project owns the contract; the matching protobuf codec and gRPC-Web call site are emitted into `common/target/generated-sources` by the build-time annotation processors so the project source stays clean.

The mojo is paired with the existing `process-annotations` pipeline: `@ProtoMessage` triggers per-class binary protobuf codec generation and a `cn1app.ProtoBootstrap` registration entry; `@GrpcClient` triggers per-interface `<Service>GrpcImpl` generation chained through `com.codename1.io.grpc.GrpcWeb.invokeUnary(…)` and a `cn1app.GrpcClientBootstrap` that wires everything to the `com.codename1.io.grpc.GrpcClients` registry (same splice pattern as `@Mapped` mappers).

B.13.1. Usage example

```
mvn -pl common cn1:generate-grpc \
    -Dcn1.grpc.proto=helloworld.proto \
    -Dcn1.grpc.basePackage=com.example.hello
```

Configuration:

Property	Description
`-Dcn1.grpc.proto=PATH`	Local path to the `.proto` file (proto3 syntax).
`-Dcn1.grpc.basePackage=PKG`	Java package the generated sources are written under. Messages, enums, and the `@GrpcClient` interface all land directly in `<basePackage>` (no nested model package — proto namespaces are flat by design).

Property	Description
-Dcn1.grpc.outputDirectory=DIR (optional)	Defaults to ${project.basedir}/src/main/java.
-Dcn1.grpc.overwrite=false (optional)	Defaults to true. Set to false to preserve user edits to existing files (only missing files are written).

B.13.2. Generated output

For a helloworld.proto declaring message HelloRequest, message HelloReply, enum Mood, and service Greeter { rpc SayHello(HelloRequest) returns (HelloReply); }, the goal emits under common/src/main/java:

```
com/example/hello/
  HelloRequest.java     // @ProtoMessage record / class
  HelloReply.java
  Mood.java             // @ProtoEnum enum with `number` accessor
  GreeterGrpc.java      // @GrpcClient interface
```

Each @ProtoMessage carries one @ProtoField(tag = N) per field; scalar fields use the default varint wire kind, sint32 / sint64 use wireType = ProtoField.WireKind.SINT, and fixed32 / fixed64 / sfixed32 / sfixed64 use wireType = ProtoField.WireKind.FIXED. Renamed fields (e.g. snake_case → camelCase) carry the original proto name via the optional name attribute so introspection tooling can recover it.

The @GrpcClient interface looks like:

Call sites use the static factory:

The <Service>GrpcImpl class that actually performs the gRPC-Web POST lives in target/generated-sources — the project source never references it directly. The build server probes the project zip for cn1app.ProtoBootstrap and cn1app.GrpcClientBootstrap and splices the registry wiring in, mirroring the existing cn1app.MapperBootstrap pattern.

B.13.3. Wire protocol

The runtime speaks **gRPC-Web binary** (application/grpc-web+proto) over HTTP/1.1, which is the standard mobile / browser variant of gRPC supported by Envoy, the official grpcweb Go proxy, and the gRPC-Web filter in modern gRPC server implementations. Plain HTTP/2 gRPC requires trailers that ConnectionRequest doesn't expose; gRPC-Web carries grpc-status in a synthetic trailer frame in the response body instead.

A successful call:

1. Encodes the request message via the build-time-generated ProtoCodec for the request type.
2. Wraps the encoded bytes in a 5-byte frame header (flag + length) and posts to <baseUrl>/<service-fqn>/<method> with Content-Type: application/grpc-web+proto and X-Grpc-Web: 1.
3. Reads the response body, iterates frames, accumulates data payload, and parses the trailer

frame for `grpc-status` / `grpc-message`.

4. Decodes the accumulated payload via the response codec and invokes the supplied `OnComplete<GrpcResponse<T>>`.

B.13.4. Scope

- Unary RPCs (single request, single response). Streaming RPCs (`stream` keyword on request or response) are rejected at generation time — gRPC-Web client streaming requires HTTP/2.
- proto3 syntax. proto2 `required` is rejected; `optional` is accepted but treated as a nullable field.
- Scalar types: `int32`, `int64`, `uint32`, `uint64`, `sint32`, `sint64`, `fixed32`, `fixed64`, `sfixed32`, `sfixed64`, `float`, `double`, `bool`, `string`, `bytes`.
- `enum` declarations (nested or top-level) emit `@ProtoEnum` enums with a `public final int number` field and a `static T forNumber(int n)` lookup.
- Nested `message` declarations are emitted as top-level siblings in the same package so the generated codec layout stays flat.
- `repeated` fields land as `java.util.List<T>`. Scalar lists are encoded packed on the wire (proto3 default); the reader accepts both packed and unpacked forms.
- `oneof` collapses to a group of nullable fields on the parent message — the mutual-exclusion guarantee is lost but round-trips work. Use application code to enforce the invariant if needed.
- Not yet supported: `map<K, V>`, well-known types (Timestamp, Empty, etc.), file `import`, streaming RPCs. These produce a clear error from the parser pointing at the offending line.
- Authentication: a bearer token is exposed uniformly as a `@Header("Authorization") String bearerToken` parameter on every RPC method, mirroring the OpenAPI bearer-token slot.

B.14. Generate GraphQL client (`generate-graphql`)

Generates a typed Codename One GraphQL client from a GraphQL SDL schema, optionally driven by a set of operation documents. Writes one `@Mapped` record (Java 17+) or class (Java 8 target) per response shape and input type, a Java `enum` per GraphQL `enum`, and one `@GraphQLClient`-annotated interface carrying one method per operation. The generated files land in `common/src/main/java` so the project owns the contract; the matching client implementation and JSON mappers are emitted into `common/target/generated-sources` by the build-time annotation processors so the project source stays clean.

The mojo is paired with the existing `process-annotations` pipeline: the generated `@Mapped` types trigger per-class JSON mapper generation and a `cn1app.MapperBootstrap` entry, while `@GraphQLClient` triggers per-interface `<Name>Impl` generation chained through `com.codename1.io.graphql.GraphQL.execute(…)` (queries and mutations) or `com.codename1.io.graphql.GraphQL.subscribe(…)` (subscriptions), plus a `cn1app.GraphQLClientBootstrap` that wires everything to the `com.codename1.io.graphql.GraphQLClients` registry (same splice pattern as `@Mapped` mappers and `@GrpcClient` clients).

B.14.1. Usage example

```
mvn -pl common cn1:generate-graphql \
    -Dcn1.graphql.schema=schema.graphqls \
    -Dcn1.graphql.operations=operations.graphql \
    -Dcn1.graphql.basePackage=com.example.starwars
```

Configuration:

Property	Description
`-Dcn1.graphql.schema=PATH`	Local path to the GraphQL SDL schema (`.graphqls` / `.graphql`). Required: it supplies the type graph used to resolve field types.
`-Dcn1.graphql.basePackage=PKG`	Java package the generated sources are written under. Models, enums, input types, and the `@GraphQLClient` interface all land directly in `<basePackage>`.
`-Dcn1.graphql.operations=PATH` (optional)	Path to a `.graphql` operation-document file, or a directory of them. When supplied the generator runs in the precise operations mode (one method and one response-type tree per named operation). When omitted it falls back to the schema-only quick-start mode.
`-Dcn1.graphql.clientName=NAME` (optional)	Simple name of the generated `@GraphQLClient` interface. Defaults to `GraphQLApi`.
`-Dcn1.graphql.endpoint=URL` (optional)	Default endpoint URL recorded on the generated annotation. The effective endpoint is always the argument passed to `of(String endpoint)`.
`-Dcn1.graphql.maxDepth=N` (optional)	Maximum selection-set depth used by the schema-only quick-start mode. Defaults to `2`.
`-Dcn1.graphql.outputDirectory=DIR` (optional)	Defaults to `${project.basedir}/src/main/java`.
`-Dcn1.graphql.overwrite=false` (optional)	Defaults to `true`. Set to `false` to preserve user edits to existing files (only missing files are written).

B.14.2. Generation modes

Unlike OpenAPI paths or gRPC service methods, a GraphQL schema doesn't enumerate the operations a client wants — the client chooses which fields to select. The goal therefore supports two modes:

- **Operations mode** (recommended): supply `cn1.graphql.operations`. For each named `query` / `mutation` / `subscription` the generator emits one interface method plus a precise `@Mapped` response-type tree that mirrors exactly that operation's selection set (nested selections become nested generated types). Referenced fragment definitions are merged into the request document automatically.
- **Schema-only quick-start mode**: omit `cn1.graphql.operations`. For each field of the root `Query` / `Mutation` / `Subscription` type the generator emits one method whose selection set is automatically expanded to `cn1.graphql.maxDepth` levels, stopping at recursive types. This is a

convenience for getting started; it may over- or under-fetch, so prefer operations mode for production code.

B.14.3. Generated output

For a Star Wars schema and an `operations.graphql` declaring `query HeroName`, `mutation AddReview`, and `subscription OnReview`, the goal emits under `common/src/main/java`:

```
com/example/starwars/
  Episode.java               // GraphQL enum -> Java enum
  ReviewInput.java           // @Mapped input type
  HeroNameData.java          // @Mapped query response root
  HeroNameData_Hero.java     // nested selection type
  AddReviewData.java         // @Mapped mutation response root
  OnReviewData.java          // @Mapped subscription response root
  StarWarsApi.java           // @GraphQLClient interface
```

GraphQL enums map to a generated Java `enum` in response types, input types, and method variables alike — the JSON mapper binds enums by their `name()`. Built-in scalars map to their boxed Java type (`Int` → `Integer`, `Float` → `Double`, `Boolean` → `Boolean`, `String` / `ID` → `String`); custom scalars fall back to `String`.

The `@GraphQLClient` interface looks like:

Call sites use the static factory. Queries and mutations report through an `OnComplete<GraphQLResponse<T>>`:

A subscription returns a `GraphQLSubscription` handle whose `cancel()` ends the stream:

The `<Name>Impl` class that performs the request lives in `target/generated-sources` — the project source never references it directly. The build server probes the project zip for `cn1app.GraphQLClientBootstrap` and splices the registry wiring in, mirroring the existing `cn1app.MapperBootstrap` pattern.

B.14.4. Wire protocol

Queries and mutations are sent as an HTTP `POST` to the endpoint with a JSON body `{"query":…,"operationName":…,"variables":{…}}` and a `Content-Type: application/json` header. The response envelope (`{"data":…,"errors":[…]}`) is parsed into a typed `GraphQLResponse<T>`; because a GraphQL response can carry both data and errors, `GraphQLResponse` exposes `getData()` and `getErrors()` independently and `isOk()` reports whether the `errors` array was empty.

Subscriptions run over a WebSocket using the `graphql-transport-ws` protocol (offered via the `Sec-WebSocket-Protocol` handshake header): the client sends `connection_init`, waits for `connection_ack`, sends a `subscribe` message carrying the operation, and maps each `next` payload's `data` to `T` before delivering it to the handler. The WebSocket endpoint defaults to the query endpoint with its scheme rewritten to `ws` / `wss`; pass a `ws://` / `wss://` URL to `of(…)` to override it. All handler callbacks are dispatched on the EDT.

B.14.5. Scope

- Queries, mutations, and subscriptions.
- GraphQL enums map to generated Java enums in every position (response fields, input fields, and method variables); unknown enum names decode to `null` rather than throwing.
- Input object types map to generated `@Mapped` classes; variables of scalar, enum, input-object, and list types are supported.
- Named and inline fragments are resolved against the schema and merged into the generated response types and the request document.
- Built-in scalars (`Int`, `Float`, `Boolean`, `String`, `ID`) map to their boxed Java types. Custom scalars fall back to `String`.
- `union` / `interface` selections are emitted with their common fields in the quick-start mode; use explicit operation documents with inline fragments for full polymorphic typing.
- Authentication: a bearer token is exposed uniformly as a `@Header("Authorization") String bearerToken` parameter on every method, mirroring the OpenAPI and gRPC bearer-token slots.

B.15. GUIBuilder goal (`guibuilder`)

The `guibuilder` goal opens the Codename One GUI builder to edit a specified GUIBuilder form.

B.15.1. Usage

```
mvn cn1:guibuilder -DclassName=com.example.MyForm
```

This will open the GUI builder to edit the form whose class is `com.example.MyForm`.

B.15.2. Parameters

className

The fully qualified name to the form class that you wish to edit. This must have been earlier generated using Section B.7, “Create GUI Form (`create-gui-form`)”.

B.16. Generate archetype (`generate-archetype`)



This goal isn’t used. It was originally developed to simplifying the maintenance of multiple application archetypes, but this approach was later abandoned in favor of application project templates.

The `generate-archetype` mojo will generate a new archetype maven project based on an existing archetype project. It’s designed to ease the maintenance of several similar archetypes which differ in some dependencies or default project source code. It takes as input a template file which references the base archetype project and specifies what to customize.

Example Usage

```
mvn cn1:generate-archetype -Dtemplate=/path/to/mytemplate.java
```

B.16.1. Parameters

template

Required. The path to a template file that should be used to generate the archetype project.

outputDir

Optional. The output directory where the archetype project should be written to. The project will be created at `outputDir/artifactId`, where the `artifactId` is as specified in the `[archetype]` section of the template. By default this will be the current working directory.



This goal doesn't require a project to run. You can run it directly using the full goal coordinates:

```
mvn com.codenameone:codenameone-maven-plugin:7.0-SNAPSHOT:generate-archetype \
-Dtemplate=/path/to/mytemplate.java
```

B.16.2. Template syntax

Here is a sample template file:

Listing 28. A sample template, which is a.Java file:

```
#set( $symbol_pound = '#' )
#set( $symbol_dollar = '$' )
#set( $symbol_escape = '\' )
package ${package};
/*
[archetype] ①
----
extends=../cn1app-archetype ②
groupId=com.codenameone.archetypes
artifactId=helloworld2-archetype
version=7.0-SNAPSHOT
----

[dependencies] ③
----
<dependency>
<groupId>com.codenameone.libs</groupId>
<artifactId>filechooser-lib</artifactId>
<version>1.0-SNAPSHOT</version>
<type>pom</type>
</dependency>
----
 */
import static com.codename1.ui.CN.*;
import com.codename1.ui.*;
import com.codename1.ui.layouts.*;
import com.codename1.io.*;
import com.codename1.ui.plaf.*;
import com.codename1.ui.util.Resources;
```

```
/**
 * This file was generated by <a href="https://www.codenameone.com/">Codename One</a> for the purpose
 * of building native mobile applications using Java.
 */
public class ${mainName} {

 private Form current;
 private Resources theme;

 public void init(Object context) {
 // use two network threads instead of one
 updateNetworkThreadCount(2);

 theme = UIManager.initFirstTheme("/theme");

 // Enable Toolbar on all Forms by default
 Toolbar.setGlobalToolbar(true);

 // Pro feature
 Log.bindCrashProtection(true);

 addNetworkErrorListener(err -> {
 // prevent the event from propagating
 err.consume();
 if(err.getError()!= null) {
 Log.e(err.getError());
 }
 Log.sendLogAsync();
 Dialog.show("Connection Error", "There was a networking error in the connection to " +
err.getConnectionRequest().getUrl(), "OK", null);
 });
 }


 public void start() {
 if(current!= null){
 current.show();
 return;
 }
 //WebSocket sock;
 Form hi = new Form("Hi World", BoxLayout.y());
 hi.add(new Label("Hello World"));

 hi.show();
 }

 public void stop() {
 current = getCurrentForm();
 if(current instanceof Dialog) {
 ((Dialog)current).dispose();
 current = getCurrentForm();
 }
 }

 public void destroy() {
 }

}
```

① The `[archetype]` section specifies details about the archetype project to be generated. For example, the `groupId`, `artifactId`, etc.

② The `extends` property in the `[archetype]` section is required, and points to the location of the archetype project that this template is based on. Path is relative to the location of the template

file.

③ The `[dependencies]` section includes content that should be injected into the `<dependencies>` section of the pom.xml file. Note that this isn't the pom file for the archetype project itself. It's the pom.xml file for the project that the archetype project is to generate.

Notice that the above template is velocity template for a Java file. It will be used as the source code for the main class in the resulting project. It's a velocity template because maven's `archetype:generate` goal will process it to replace properties such as package and the main class name.

Template sections

archetype

This is a required section and specifies both the location of the base archetype project from which this project is to be derived, and the coordinates of the output archetype project, such as `groupId`, `archetypeId`, and `version`.

Example

```
[archetype]
----
extends=../cn1app-archetype
groupId=com.codenameone.archetypes
artifactId=helloworld2-archetype
version=7.0-SNAPSHOT
----
```

Properties

extends

Required. The path to the archetype project that this is based on. This should usually be the `cn1app-archetype` project as it includes the placeholders that this generator relies on for injecting content into the pom.xml file and its project structure was used as a basis for developing this mojo.

groupId

The groupID of the resulting archetype project. You can or use the `id` property to specify `groupId`, `artifactId` and `version` in a single string.

artifactId

The artifactId of the resulting archetype project. You can or use the `id` property to specify `groupId`, `artifactId` and `version` in a single string.

version

The version of the resulting archetype project. You can or use the `id` property to specify `groupId`, `artifactId` and `version` in a single string.

id

A colon-separated string in the format `groupId:artifactId:version` that can be used as an alternative to `groupId`, `artifactId`, and `version`.

parentGroupId

If the output archetype project should be part of a multi-module project, then this will specify the parent `groupId` for the `<parent>` tag in the pom.xml file.

parentArtifactId

If the output archetype project should be part of a multi-module project, then this will specify the parent `archetypeId` for the `<parent>` tag in the pom.xml file.

parentVersion

If the output archetype project should be part of a multi-module project, then this will specify the parent `archetypeId` for the `<parent>` tag in the pom.xml file.

parent

A colon-delimited string in the format `parentGroupId:parentArtifactId:parentVersion` that can be used as an alternative to the `parentGroupId`, `parentArtifactId`, and `parentVersion` properties.

dependencies

Specify more dependencies that should be injected into the `<dependencies>` section of the pom.xml file for the common module of the maven project that the archetype will generate. The content of this section will be injected into the `src/main/resources/archetype-resources/common/pom.xml` of the archetype project.

Example

```
[dependencies]
---
<dependency>
    <groupId>com.codenameone.libs</groupId>
    <artifactId>filechooser-lib</artifactId>
    <version>1.0-SNAPSHOT</version>
    <type>pom</type>
</dependency>
---
```

css

CSS content that should be injected into the theme.css file of the project. This CSS will be injected into the `src/main/resources/archetype-resources/common/src/main/css/theme.css` file of the archetype project.

Example

```
[css]
---
#Constants {
    includeNativeBool: true;
}
Button {
    color:green;
    border:1px solid green;
    border-radius: 2mm;
    margin: 5mm;
}
---
```

properties

Properties that should be appended to the `codenameone_settings.properties` file. These will be added to the `src/main/resources/archetype_resources/common/codenameone_settings.properties` file of the archetype project.

Example

```
[properties]
---
codename1.arg.win.desktop-vm=zuluFx8-32bit
codename1.arg.win.desktopExtractDll=true
codename1.arg.win.launchOnStart=true
codename1.arg.win.runAfterInstall=true
---
```

files

Files contains a list of files that should be created in the archetype project. All file paths are relative to the "common" project root directory. Each path should have a corresponding section with the heading `[file:path/to/file]`.

Example

```
[files]
----
src/main/guibuilder/__mainName__MainForm.gui
src/main/java/__mainName__MainForm.java
----

[file:src/main/guibuilder/__mainName__MainForm.gui]
----
<?xml version="1.0" encoding="UTF-8"?>

<component type="Form" layout="LayeredLayout" layeredLayoutPreferredWidthMM="0.0"
layeredLayoutPreferredHeightMM="0.0"  autolayout="true" title="${mainName}MainForm" scrollableY="true"
```

```
name="${mainName}MainForm">
<component type="Button" text="Click Me" name="Button">
<layoutConstraint insets="auto auto auto auto" referenceComponents="-1 -1 -1 -1" referencePositions="0.0 0.0
0.0 0.0" />
</component>
</component>
----

[file:src/main/java/__mainName__MainForm.java]
----
package ${package};
public class ${mainName}MainForm extends com.codename1.ui.Form {
public ${mainName}MainForm() {
this(com.codename1.ui.util.Resources.getGlobalResources());
}

    public ${mainName}MainForm(com.codename1.ui.util.Resources resourceObjectInstance) {
        initGuiBuilderComponents(resourceObjectInstance);
    }

//-- DON'T EDIT BELOW THIS LINE!!!
    protected com.codename1.ui.Button gui_Button = new com.codename1.ui.Button();


// <editor-fold defaultstate="collapsed" desc="Generated Code">
    private void initGuiBuilderComponents(com.codename1.ui.util.Resources resourceObjectInstance) {
        setLayout(new com.codename1.ui.layouts.LayeredLayout());
        setInlineStylesTheme(resourceObjectInstance);
        setScrollableY(true);
                setInlineStylesTheme(resourceObjectInstance);
        setTitle("MyForm");
        setName("MyForm");
        gui_Button.setText("Click Me");
                gui_Button.setInlineStylesTheme(resourceObjectInstance);
        gui_Button.setName("Button");
        addComponent(gui_Button);
        ((com.codename1.ui.layouts.LayeredLayout)gui_Button.getParent().getLayout()).setInsets(gui_Button,
"auto auto auto auto").setReferenceComponents(gui_Button, "-1 -1 -1 -1").setReferencePositions(gui_Button, "0.0
0.0 0.0 0.0");
    }// </editor-fold>

//-- DON'T EDIT ABOVE THIS LINE!!!
}
----
```

B.17. Install Codename One (`install-codenameone`)

Installs/Updates Codename One into the user's home directory. This goal is automatically run if a Maven build is attempted and it detects that Codename One isn't yet installed (in the `$HOME/.codenameone`) directory.

B.18. Install legacy Cn1lib (`install-cn1lib`)

Installs a legacy cn1lib file as a dependency in this application project.

See Section 1.2.4, "Adding project dependencies" for fuller coverage of project dependencies.



Using this goal explicitly is a last resort. The best solution for installing add-ons

into your project is Section 1.2.4.5, “Managing add-ons in Codename One Settings”.

B.18.1. Usage example

```
mvn cn1:install-cn1lib -Dfile=/path/to/MyLegacyLib.cn1lib
```

This will generate a Maven pom project for this lib inside the "cn1libs" directory, and it will add a dependency inside the common/pom.xml file.



The generated project isn’t a cn1lib-archetype project, and it isn’t meant to change. It’s merely a Maven wrapper around the cn1lib file that provides compatibility with Maven’s dependency mechanism.

B.18.2. Removing the Cn1lib later

If you need to remove the cn1lib later (that is, revert an installation), you can remove the `<dependency>` tag that was added to the common/pom.xml file for the cn1lib.

You can also remove the directory that was created inside the cn1libs folder for this cn1lib - but this step isn’t necessary.

B.18.3. Parameters

file

Path to the cn1lib file you want to install.

groupId

The groupId to use for the generated pom project. Optional. If not specified, it will use the same groupId as the root project.

artifactId

The artifactId to use for the generated pom project. Optional. If not provided, it will generate an artifactId derived from the project’s artifactId.

version

The version to use for the generated pom project. Optional. If not provided, it will use the project version.

updatePom

A boolean flag indicating whether it should automatically update the pom.xml file with the dependency.

Default is `true`

overwrite

A boolean flag indicating whether it should overwrite an existing project of the same name. Default false.

Default is `false`

B.19. Open Designer (`designer`)

Opens the legacy Codename One designer app to edit the project "theme.res" file.



If your project is using CSS it may not have a theme.res file, thus this goal will fail in this case.

B.19.1. Usage example

```
mvn cn1:designer
```

This will open the Codename One designer.

B.20. Prepare simulator classpath (`prepare-simulator-classpath`)

Sets up some properties and environment variables required for the Codename One simulator to run properly.

This goal is bound to the `initialize` phase of the `Java SE` module of the Section A.1, “Codename One application project archetype (cn1app-archetype)” and shouldn’t be executed directly.

B.21. Open settings (`settings`)

Downloads and opens the standalone Codename One Settings editor bound to the current Maven project. The editor manages basic project properties, build hints, and extensions.

```
mvn cn1:settings
```

See Appendix D, *Codename One Settings*

B.22. Run tests (`test`)

Runs Codename One unit tests.

This goal is bound to the `test` phase of Section A.1, “Codename One application project archetype (cn1app-archetype)” projects, so that running it directly isn’t necessary. If you build a cn1app-archetype project using:

```
mvn install
```

or

```
mvn package
```

Then this goal will be executed automatically.



Prevent tests from running using the `skipTests` flag. For example, `mvn install -DskipTests`

B.23. Update Codename One (`update`)

Updates Codename One. This updates tools that live inside `$HOME/.codenameone`, such as the GUI builder, and tries to update the `cn1.version` and `cn1.plugin.version` properties in the project's `pom.xml`. The standalone Settings editor is resolved from Maven using `cn1.plugin.version` when you run `mvn cn1:settings`.

B.23.1. Usage example

```
mvn cn1:update
```

Parameters

newVersion

(Optional) The version to update to. This should be a version number available on Maven central. Will accept a value of "LATEST" to cause it to resolve to the latest version available on Maven central.

If this parameter is omitted, then it will be implicitly set to `LATEST`, but it won't update the `cn1.version` or `cn1.plugin.version` properties if they're set to a SNAPSHOT version.

See Section 1.3, "Updating Codename One" for more information about updating Codename One.

Appendix C: API



This section isn't a comprehensive treatment of the Codename One API. For a more in-depth discussion of Codename One and it's supported APIs, refer to the Developer Guide [https://www.codenameone.com/developer-guide.html].

Codename One apps support a subset of the Java SE 8 API as well as its own light-weight runtime and UI component library that includes support for everything that would expect in a mobile application platform.

See the JavaDocs [https://www.codenameone.com/javadoc/] for a full list of supported classes, and check out the source in the git repository [https://github.com/codenameone/CodenameOne].



The Codename One source is open source. Released under GPLv2 with Classpath Exception.

Codename One is much more than an API library. It provides a full tool-chain and eco-system for developing beautiful, performant native mobile apps with a single codebase in Java and Kotlin. Please see the introduction in the Developer guide [https://www.codenameone.com/developer-guide.html#_introduction] for a proper overview of Codename One.

C.1. Limitations

No reflection

Codename One apps don't support reflection because reflection makes it impossible to keep app-size down.

C.2. Kotlin API support

Codename One apps support the Kotlin Standard Library [https://kotlinlang.org/api/latest/jvm/stdlib/]. Kotlin support is still new, so are still discovering which libraries are compatible. APIs that rely on reflection won't be supported, but most other libraries should work.

C.3. Add-on libraries

Add-on libraries can be added to your library in the common/pom.xml file, but, if you use APIs that aren't supported by Codename One (for example, which use reflection), then your app will fail to build.

Codename One supports its own library format (cn1lib) which sort of "certifies" that it's compatible with Codename One. Run `mvn cn1:settings` and open **Extensions** to browse the growing catalog of available cn1libs. See Codename One Settings.

For more information about cn1libs, see the cn1libs section [https://www.codenameone.com/developer-guide.html#_libraries_cn1lib] of the developer guide.

Appendix D: Codename One Settings

Codename One Settings is a standalone editor for the current Maven project. Use it to edit the application's basic properties, manage build hints, and browse, install, update, or uninstall extensions. Signing assets are managed by the separate Certificate Wizard, and cloud builds are monitored on the Codename One website.

D.1. Opening Codename One Settings

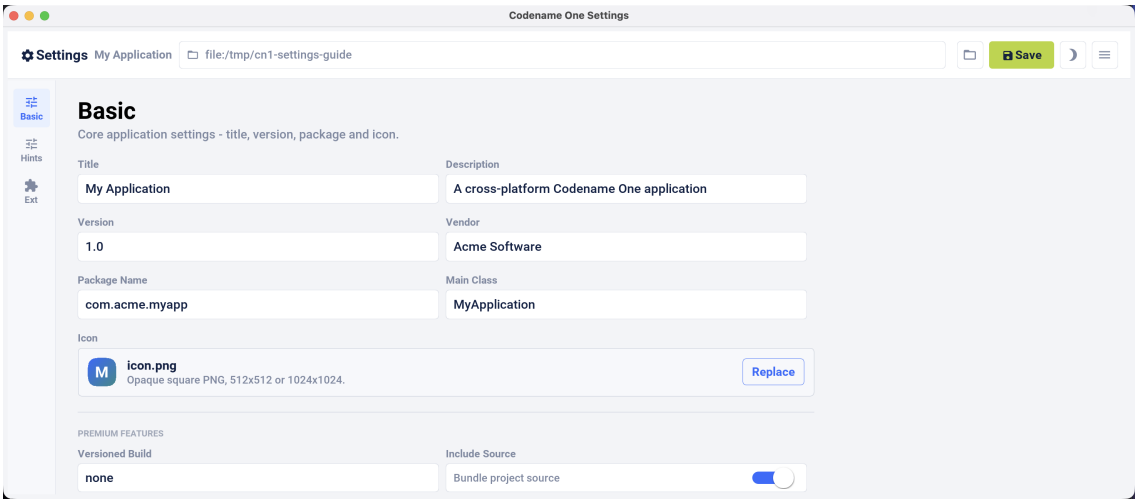
Run the following command from the root directory of a Codename One Maven project:

```
mvn cn1:settings
```

Maven downloads the Settings artifact that matches the Codename One Maven plugin version and opens it as a desktop application bound to the current project. The same command works in a terminal or an IDE's Maven runner.

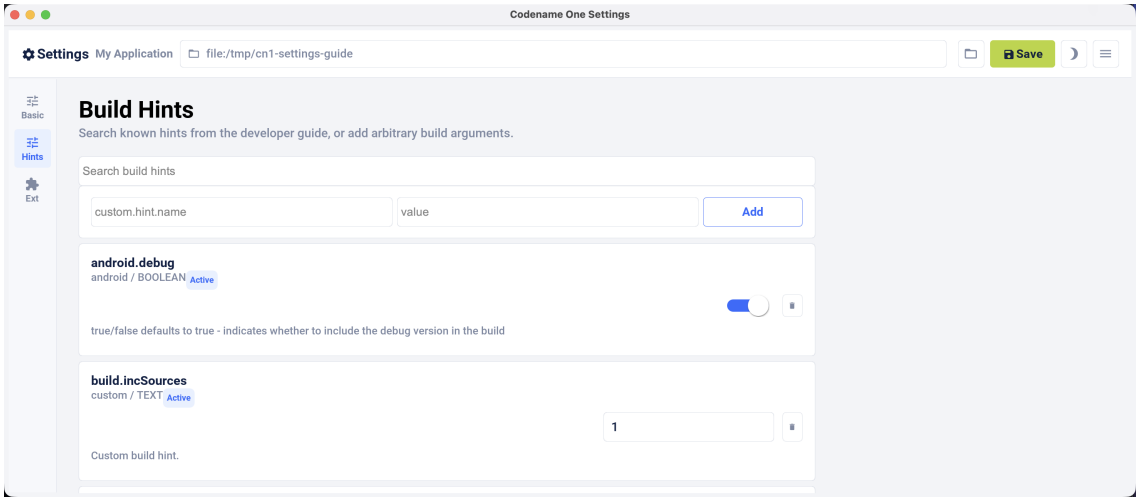
D.2. Basic settings

The **Basic** view edits the application title, description, version, vendor, package name, main class, icon, versioned-build selection, and include-source option. Changes are written to `common/codenameone_settings.properties` when you click **Save**.



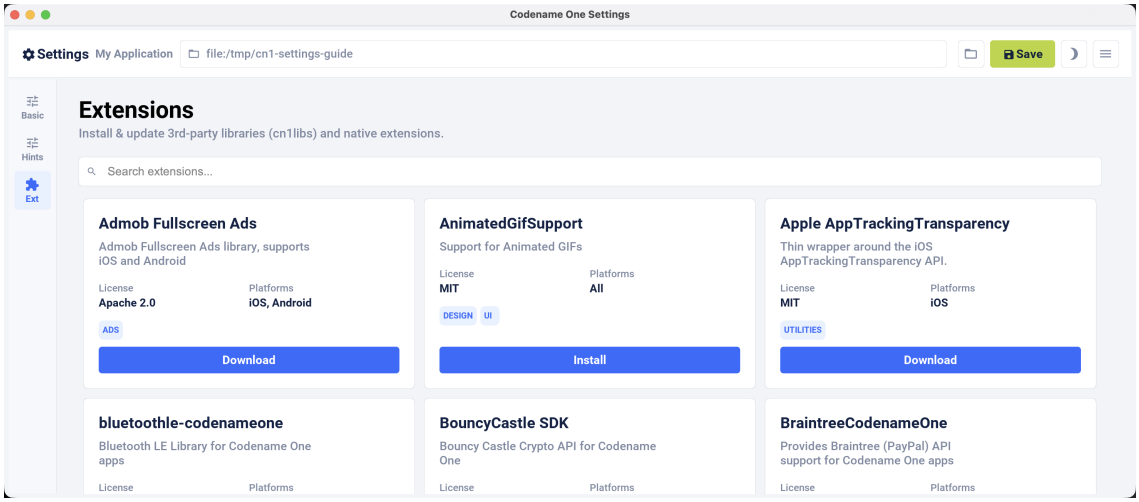
D.3. Build hints

The **Build Hints** view searches the build-hint catalog from this guide. It provides a control appropriate to each known hint and also accepts arbitrary key/value hints. The editor adds and removes the `codename1.arg.` prefix when reading and writing `codenameone_settings.properties`.



D.4. Extensions

The **Extensions** view searches the Codename One library catalog. Depending on how an extension is distributed, installation either adds a Maven dependency to `common/pom.xml` or installs a legacy cn1lib. Click the action for an installed extension to uninstall it. Settings displays a compatibility warning before installing a catalog entry known to target an older Codename One release.



Appendix E: Rich properties file (rpf) format

The rich properties file (rpf) format is used to store configuration for the `generate-app-project` goal. The format is the same as a regular properties file except that it can accommodate properties whose values are "rich" and lengthy.

"Rich" properties use the following syntax:

```
[keyname] ①
=== ②
Key value ③
=== ④
```

① The property key appears on a new line wrapped in square brackets.

② A separator consisting of 3 or more equals signs on the next line.

③ The property value, which can be one or more lines of content.

④ The separator with the same number of equals signs on a line of its own.

E.1. Example rich properties file

```
firstName=Bob
lastName=Smith
[bio]
====
Bob is a hard worker.
He attended Harvard and is looking for opportunities in the fast food industry.
====
age=23

# A comment that it is ignored
[xmldata]
========
<person>
    <name>Bob Smith</name>
    <age>23</age>
</person>
========

favoriteColor=Brown
```

Many of the properties of this file are regular properties. Two rich properties exist: `bio` and `xmldata`.

Historical reference

Chapter 89. Working with UWP



The Codename One UWP (Universal Windows Platform) target was discontinued in release 7.0.229. This chapter is retained to document historical context for legacy projects and older blog posts. New projects should target the supported Codename One platforms instead.

If you're maintaining an older codebase or reading archived material that mentions UWP, treat those instructions as historical. The current Maven build templates, initializr output, and IDE shortcuts no longer include a UWP target.