Introduction to Game of Trees (EuroBSDcon 2026)

Contents

Chapter 1

About Game of Trees

Game of Trees (Got) is a version control system which prioritizes ease of use and simplicity over flexibility.

Got uses Git repositories to store versioned data. Git can be used for any functionality which has not yet been implemented in Got. It will always remain possible to work with both Got and Git on the same repository.

Got is being developed on OpenBSD by OpenBSD developers and other contributors. While Got's main target audience are OpenBSD developers, it is being used for general version control tasks by users of OpenBSD and other Unix-like operating systems.

The software is freely usable and re-usable by everyone under a BSD-style license.

Chapter 2

Introduction

This tutorial has the goal of familiarizing participants with Game of Trees.

After completing this tutorial, participants should be able to collaborate on shared text files with other users of Game of Trees, and users of Git, across an online Git hosting service or a self-hosted Git server setup (the creation of which is possible with Game of Trees, but is out of scope for this tutorial).

No prior knowledge of Git is required. However, a basic understanding of general version control concepts and the Git repository format is required to use Game of Trees effectively.

See the *Version Control* section for a quick introduction to the basics of version control.

See the *Git Repository Format* section for a quick introduction to the Git repository format.

Skip to the *Installing Game of Trees* section if you are already familiar with the topic.

Chapter 3

Version Control

Version control is a process which keeps track of changes made to digital artifacts over time. It is used to support any creative process which involves the creation and maintenance of digital artifacts. Either with multiple contributors who need to coordinate the work they do on a shared set of artifacts, or even a single person collaborating with their past self by building on top of work they did days or weeks before.

Version control is most widely used with text files. However, it can be used when creating art (images, music) or documents which are stored in binary file formats (office documents, spreadsheets).

In software development in particular, version control provides a record of the changes made to program source code over time. It allows for changes to be replicated between multiple machines used for development of the same software. And version control allows for changes to be applied selectively to different versions of the software. For example, there might be a **stable** version of the software which end users run in production, and which only receives changes which repair newly discovered defects and bugs. In parallel, a **development** version can be in progress, which aggregates new functionality and features over time and eventually becomes the next stable version, once the work on the new set of features is considered complete. It is usually possible to mark an arbitrary state of development of the project as a **release**. Releases serve as points of reference in the evolutionary history of the software, and bundle a bunch of changes which are deemed safe for use in production by end users.

Several version control systems exist. Some of them trace their roots back to the 1970s and 1980s. Some version control systems are released under open-source licences, others are proprietary. Today, the most commonly used open-source version control system is Git. Also in use are Mercurial, Subversion, and CVS. And there are proprietary systems which we won't go into.

Most version control systems were designed with the assumption that text files represent one of the basic units involved in version control operations, with each such file being composed of lines of text, delineated by the ASCII sequences LF (0xa) or CR/LF (0xd 0xa). These assumptions correspond to the way most programming languages are stored in files.

If multiple developers make changes to such files in parallel, the changes might overlap and need to be reconciled in order to produce a correct program. There are two basic approaches for solving this problem:

One approach is **lock-modify-unlock**, where a single developer reserves (locks) an entire file for the period of time used to modify the file, and releases (unlocks) the file again once modifications are complete. This process is easy to implement, provided there is a central place to register locks on files. Overlapping changes are avoided by never allowing concurrent modifications to the same file. However, in practice, locks might be ignored, or stolen, or can become stale, requiring manual workarounds outside the intended workflow.

Another approach is **copy-modify-merge**, where every developer obtains their own copies of all files and is free to modify any file, at any time. When synchronizing changes, the version control system uses a merge algorithm to detect modifications made to overlapping sections of files, and marks such overlapping sections for conflict resolution. Changes which do not overlap are merged automatically, which can sometimes yield wrong results because merge algorithms are (generally) unaware of the semantics of the program code being merged. To catch such problems, developers can employ code review, manual testing, and automated testing. Since such steps must be part of a software development quality assurance process anyway, regardless of version control, the trade-off between an occasional bad merge and much improved developer productivity is usually worth it.

Most version control systems today use the copy-modify-merge approach, with some also providing support for the lock-modify-unlock approach (which usually requires a central server to store locks). The merge algorithm used for text files is usually the

diff3 algorithm, or a close derivative thereof. See https://www.cis.upenn.edu/~bcpierce/papers/diff3-short.pdf for details of how it works.

As part of their basic operation, version control systems need to assign stable identifiers to versions of files, and even to collections of files (directories).

One possibility is to assign unique **version numbers** which increment with each change recorded by the version control system. This approach requires a central server responsible for assigning numbers to versions, and storing these numbers as meta-data. Numeric identifiers make it inherently easy to tell whether a given change is older or newer than another change, which makes the use of some version control operations very intuitive, such as when comparing versions of a file.

When **distributed** version control systems arrived in the early 2000s, they introduced the concept of an **intrinsic identifier**. Such identifiers are derived from the content of a file, rather than being assigned to the version of a file as meta-data. This is usually done by running a checksum algorithm on file content, perhaps with some meta-data mixed in, and using the resulting checksum as the identifier for a specific version. A manifest, which lists multiple files with their names and their intrinsic identifiers, can likewise be assigned an intrinsic identifier, in order to represent collections of files in the version control system. This approach has the advantage that stable identifiers can be assigned to versions without relying on a central registry. However, it carries the risk of checksum collisions: It is impossible to store two different sets of content which result in the same checksum, should such content exist. In practice, modern checksum algorithms provide enough entropy such that this risk can simply be ignored. However, future research in cryptography could invalidate this assumption. Maintainers of version control systems which use intrinsic identifiers must remain aware of relevant research and be prepared to tweak the version control system in case the risk of collisions is no longer negligible.

For usability reasons, it is usually possible to set human-readable aliases for unique identifiers. Such aliases can then be used in place of version numbers of checksums while interacting with the version control system. For example, such aliases may appear similar to paths in a Unix filesystem.

Chapter 4

Git Repository Format

A Git repository is a database which stores a collection of artifacts that are tracked under version control.

From the version control system's point of view, the repository is the largest unit of things which exist in its universe. Everything which is under version control lives somewhere in a repository.

Software projects may use more than one such repository, each of which may have a specific scope or purpose within the larger context of the software project.

Note
Some projects even create virtual links between repositories, treating a collection of repositories as a single unit. While Git supports such use cases via its "submodules" feature, Game of Trees does intentionally not support this feature because of the added complexity it creates.

4.1 Git Repository Objects

Conceptually, the repository's data model is a directed acyclic graph which contains four types of objects as nodes.

Each object is referred to by an intrinsic identifier, or **object ID**, which is a checksum calculated over both the object's header (type and size information) and the data stored in the object itself.

The 4 object types are as follows:

Blobs

The content of files is stored in objects of type **blob**.

Trees

A **tree** object is a manifest which lists any number of blobs or other trees along with their path-names and object IDs, in order to represent a hierarchy of files and directories.

Commits

A **commit** object points to the root element of one tree, and thus records the state of this entire tree as a snapshot. Commit objects are chained together to form lines of version control history. Most commits have just one successor commit, but commits may be succeeded by more than one subsequent commits so that diverging lines of version control history, known as **branches**, can be represented. A commit which precedes another commit is referred to as that other commit's **parent commit**. A commit with multiple parents unites disparate lines of history and is known as a **merge commit**.

Tags

A **tag** object associates a user-defined label with another object, which is typically a commit object. Tags are usually used to represent releases of the software managed in the Git repository.

4.2 Git Repository References

A Git **reference** associates a path-like name with an object ID. References act as user-friendly aliases for intrinsic identifiers.

A prominent use of references is providing names to branches in the repository. Such references point at commit objects which represent the current tip commit of a branch. Because references may point to arbitrary object IDs, their use is not limited to branches.

A **symbolic reference** associates a name with the name of another reference, much like a symbolic link in a Unix-like filesystem. The most prominent example is the HEAD reference which points at the name of the repository's default branch reference, which is usually, but not always, *refs/heads/main*. Game of Trees resolves the HEAD reference to find the default branch to use if no branch is provided by the user.

4.3 Git Repository Work Tree

A typical Git repository exposes a **work tree** which allows the user to make changes to versioned files and create new commits. When a Git work tree is present, the actual repository data is stored in a *.git* subfolder of the repository's root directory.

A Git repository without a work tree is known as a **bare repository**. Game of Trees does not make use of Git's work tree and treats every repository as if it was bare.

When sharing a repository between Git and Game of Trees, a bare repository should be preferred to avoid interactions between Game of Trees and Git's work tree format. Running both version control systems on the same non-bare repository can sometimes result in unexpected or inconvenient behaviour of either version control system. For example, when Game of Trees creates new commits in the repository the Git index (staging area) of the repository's Git work tree will not change. This results in inconsistencies from Git's point of view.

4.4 The Game of Trees Work Tree

Game of Trees uses a custom work tree format which allows an arbitrary number of work trees to be checked out from a Git repository. Each work tree can commit new changes to the repository, and update itself to commits other than the one it was checked out from.

It is possible to scope a work tree to a sub-directory of the file and directory tree stored in commits. This is can be useful when a Git repository contains multiple projects or software modules.

A work tree is always tied to a branch, which means the "detached HEAD" state known to Git users cannot occur with Game of Trees.

4.5 Further reading

More details about the Git repository format are available in the *git-repository* manual page once Game of Trees has been installed, or online at: https://gameoftrees.org/git-repository.5.html

There is also no shortage of articles and books out there which describe the Git repository format in various degrees of detail.

For details about the Game of Trees work tree format, see the *got-worktree* manual page, or read it online at https://gameoftrees.org/got-worktree.5.html

Chapter 5

Installing Game of Trees

This tutorial assumes that the following commands can be run at the shell prompt:

```
got
tog
```

These commands should be part of any Game of Trees installation.

Game of Trees interacts with some external software components, which might need to be configured separately by the user:

- Unix terminals and shell command lines
- Secure Shell (Implementations other than OpenSSH might be incompatible)
- Text editors (ed, vi, emacs, etc.)

Other software which can be useful to have installed alongside Game of Trees:

- The unix *man* command for viewing manual pages
- Text pagers (more, less, etc.)
- Git, for some features not (yet) implemented by Game of Trees

5.1 Installation on OpenBSD

To install Game of Trees, run:

```
pkg_add got
```

See https://gameoftrees.org/install.html for details.

5.2 Installation on other Unix-like systems

Installable binary packages of Game of Trees are known to exist for the following systems. If your operating system is one of them then please search the database of installable packages.

- NetBSD/pkgsrc

- FreeBSD
- MacPorts
- Homebrew
- Alpine Linux
- NixOS
- Void Linux
- Arch Linux
- Debian
- Ubuntu
- Chimera Linux
- GNU Guix
- Slackware Linux (SlackBuild)
- openSUSE (Build Service)

On other Unix-like operating systems, Game of Trees can be built from source code. See https://gameoftrees.org/portable.html for details.

5.3 Microsoft Windows

On Windows, use the Windows Subsystem for Linux (WSL) with one of the Linux distributions above. For example, in Powershell, run:

```
wsl --install --distro openSUSE-16.0
```

Reboot, then start the WSL environment:

```
wsl --distro openSUSE-16.0
```

Now install Game of Trees packages from https://build.opensuse.org/package/show/devel:tools:scm/got by running:

```
wsl zypper addrepo \
  https://download.opensuse.org/repositories/devel:tools:scm/16.0/devel:tools:scm.repo
wsl zypper refresh
wsl zypper install got
```

You should now be able to run Game of Trees commands from a Windows Powershell prompt by prefixing the commands with *wsl*:

```
wsl got
wsl gotadmin
wsl tog
```

Note
The above instructions for Windows are untested. If you are running into problems please provide feedback.

Chapter 6

Version Control Workflows

6.1 Supported Workflows

In the abstract, version control can be represented by workflows (or "use cases"), each of which serves a specific task or goal as part of the life-cycle of a software development project.

Version control workflows supported by Game of Trees include:

- Adding one or more files to a software project under version control.
- Viewing changes made to files, relative to previous versions.
- Undoing specific changes or refining specific changes.
- Recording a new snapshot of the project after changes were made to files.
- Viewing a timeline of project snapshots recorded over time.
- Removing files while keeping a record of their previously existing versions.
- Managing parallel development of new software features by one or more developers.
- Synchronizing changes between machines used by one or more developers.
- Applying contributions received from external developers.
- Marking specific snapshots of the project as published versions of the software.
- Fixing specific problems in published versions of the software.

6.2 Modeling Workflows

The Game of Trees user interface models version control workflows on top of the Git repository format. User-visible Git repository concepts involved in modeling workflows were already discussed in the "Git Repository Format" section:

- The repository database (container of the version-controlled universe)
- Objects of types blob, tree, commit, and tag (files, directories, snapshots, releases)
- Object identifiers (SHA1 or SHA256 hashes)
- References (human-readable identifiers)

Knowledge of any other Git repository internals should not be required while using Game of Trees for version control.

Additional user-visible concepts, which will be covered later in this tutorial, are:

- Configuration files
- Work trees
- Branches
- Remote repositories

For long-term repository maintenance, especially on servers, a basic understanding of additional Git repository concepts will be useful:

- Loose object files
- Pack files

These concepts will likewise be covered later in this tutorial.

Chapter 7

First steps with Game of Trees

7.1 Setting up the environment

The behaviour of *got* is influenced by shell environment variables. The following two environment variables should be set for the purposes of this tutorial:

- *GOT_AUTHOR*: This variable sets the author name and email address recorded when a new snapshot is committed to a Git repository. Set it to either your own name and email address, or use a generic placeholder name, such as *Tom Smith*. Keep in mind to quote the value when passing it on the shell command line. For example, run:

  ```
  export GOT_AUTHOR='Tom Smith <tom.smith@example.com>'
  ```

  Note
  If the *GOT_AUTHOR* variable is not set, and no author information is set in the configuration file (covered later in this tutorial), then the *got commit* command will error out.

- *EDITOR*: This variable sets the program to run for editing commit log messages. Set this to an editor you are familiar and comfortable with. The default editor used by Game of Trees is *vi* (it used to be *ed*, but this caused complaints). For example, run:

  ```
  export EDITOR=emacs
  ```

7.2 Creating some files to work with

Create a new empty directory, and create some directories and text files inside of it. The following file and directory hierarchy is recommended for this tutorial:

```
.
|-- alpha
|-- beta
|-- epsilon
|   `-- zeta
`-- gamma
    `-- delta

2 directories, 4 files
```

Running the following commands will create this hierarchy of files and directories:

```
mkdir ~/new
cd ~/new
echo alpha > alpha
echo beta > beta
mkdir gamma
echo delta > gamma/delta
mkdir epsilon
echo zeta > epsilon/zeta
```

The *~/new* directory is temporary. It can be removed again once this entire tutorial section has been completed.

7.3 Creating a Git repository

Create a new directory which will host repositories and other data required throughout this tutorial:

```
mkdir ~/tutorial
```

Switch into this directory and create the first Git repository using the *got init* command:

```
cd ~/tutorial
got init firststeps.git
```

7.4 Importing files into a repository

Switch into the newly create repository and import the file and directory hierarchy which was created above:

```
cd firststeps.git
got import ~/new
```

Your editor should open and display a file which contains an empty line at the top, followed by a line which begins with # that you can ignore for now.

Type a log message anywhere in this file to document the initial import of the file hierarchy. (Any lines beginning with # will be ignored and will not become part of the recorded log message.) This first log message could be something as trivial as “initial import”, such that the editor window contains:

```
initial import
# /home/user/new to be imported to branch refs/heads/main
```

Now save the file and exit your editor. The *got* program should produce output similar to the following, showing the list of files which were added:

```
A  /home/user/new/epsilon/zeta
A  /home/user/new/gamma/delta
A  /home/user/new/alpha
A  /home/user/new/beta
Created branch refs/heads/main with commit 80902db6f7b62e715ff67617ca1d0b2fd11b4117
```

We have now recorded a new snapshot in the Git repository which contains the newly added files.

Some Git-repository concepts have already appeared during *got import*. We will go through them now:

- **Object-ID**: An object ID appears on the last line of output: *80902db6f7b62e715ff67617ca1d0b2fd11b4117* This hash is the intrinsic identifier of a new commit object which was created in the repository.

- **commit**: The commit object mentioned on the last line of output contains the log message which was entered, a timestamp, and the intrinsic identifier of a tree object via which the files, stored as blob objects, can be reached. You can view the commit object's content by passing the object identifier (or an abbreviated version of it) to the *got cat* command. For example, assuming the above identifier, a suitable command would be:

```
got cat 80902db
```

This should print something like:

```
tree df48b98c04c61ad621e55006e154cf76f50580be
numparents 0
author Tom Smith <tom.smith@example.com> 1787919674 +0000
committer Tom Smith <tom.smith@example.com> 1787919674 +0000
messagelen 16

initial import
```

- **tree**: The snapshot the commit object points to is composed of tree objects (directories) and blob objects (files). The *got tree* command can be used to obtain a view of files and directories stored in the snapshot. Run:

```
got tree -R
```

The output should be:

```
alpha
beta
epsilon/
epsilon/zeta
gamma/
gamma/delta
```

The intrinsic identifiers of files (blob objects) and directories (tree objects) are also part of the snapshot. They can be displayed by passing the *-i* option to the *got tree* command:

```
got tree -R -i
```

The output should be similar to:

```
4a58007052a65fbc2fc3f910f2855f45a4058e74 alpha
65b2df87f7df3aeedef04be96703e55ac19c2cfb beta
f009e7cde4d52767055b91ddc8f1d32367f476b6 epsilon/
fd08df0afa4d1d3faece37798d169e5a46d9d3fd epsilon/zeta
43d3931c967119e70d9d2c0ea8ecc6695275a54a gamma/
ab135eefea6f73b921c7fec469b5f0e9db86b910 gamma/delta
```

- **reference**: The *got import* log message comment, and the final line of output of *got import*, mention *refs/heads/main*, which looks like a filesystem path on Unix-like systems. In Git repository terms, *refs/heads/main* is a **reference**, a human-readable alias for the intrinsic identifier of the newly created commit. At the Git repository level, the names used beneath *refs/* are arbitrary. But when creating human-readable identifiers, both Game of Trees and Git follow some naming conventions. By convention, commits which record snaphots containing changes we made to files in our repository have reference names which begin with *refs/heads/*. To understand why, we will unpack the concept of "branches" below.
- **branch**: The *got import* log message comment, and the final line of output of *got import*, mention a branch being created. In the real world, branches grow on trees. But even though Git has a tree object type in its repository format, branches in Git repositories have more to do with commit objects than with tree objects. Our repository currently contains only a single commit. More commits can be added by recording more snapshots, resulting in a chain of commits called a **branch** (of the software project's development history). The top-most commit of this chain is referred to as a **branch head** or a **branch tip**. Whenever a new snapshot is recorded, a new head commit is added to a branch, and the branch's reference in *refs/heads/* will be updated such that it always keeps pointing at the latest head commit of the branch. That is, the branches which we modify locally live in the **reference namespace** *refs/heads/*. There are other special-purpose namespaces beneath the global *refs/* namespace. We will get to some of them later.

7.5 Checking files out of a repository

The *got import* command can be used to put a set of files under version control, which now serve as a base for further modifications. But files stored in the repository cannot be changed directly. We must create a **work tree** (or "working copy") of the files in order to make changes to them.

The *got checkout* command is used to create work trees. The simplest way to run this command is as follows:

```
cd ~/tutorial
got checkout firststeps.git
```

This should print output similar to the following:

```
A  /home/user/tutorial/firststeps/alpha
A  /home/user/tutorial/firststeps/beta
A  /home/user/tutorial/firststeps/epsilon/zeta
A  /home/user/tutorial/firststeps/gamma/delta
Checked out refs/heads/main: 53ed49f7cd92f7c3223838d856bb58cbb0e81e5d
```

A work tree of the *main* branch was created in the directory *~/tutorial/firststeps*.

7.6 Making and viewing changes to files

Edit files in the work tree with your editor, or with shell commands. For example:

```
cd ~/tutorial/firststeps
echo changed alpha > alpha
```

The current status of files in the work tree can be obtained by running:

```
got status
```

This should print a line indicating that the file *alpha* has been modified:

```
M  alpha
```

Changes to file content in the work tree can be viewed by running:

```
got diff
```

This should print changes in all modified files, using the "unified diff" command known from the *diff -u* utility on Unix-like systems.

```
diff /home/user/tutorial/firststeps
path + /home/user/tutorial/firststeps
commit - 53ed49f7cd92f7c3223838d856bb58cbb0e81e5d
blob - 4a58007052a65fbc2fc3f910f2855f45a4058e74
file + alpha
--- alpha
+++ alpha
@@ -1 +1 @@
-alpha
+changed alpha
```

The information conveyed here is as follows:

```
diff /home/user/tutorial/firststeps
path + /home/user/tutorial/firststeps
```

We are viewing differences between the base versions of files in the work tree at */home/user/tutorial/firststeps* and the files located at on-disk paths within this work tree. The base versions of files which the work tree refers to are stored as blobs in the repository.

```
commit - 53ed49f7cd92f7c3223838d856bb58cbb0e81e5d
blob - 4a58007052a65fbc2fc3f910f2855f45a4058e74
file + alpha
```

Specifically, for the file *alpha*, the differences we see are relative to the given blob object which was reached via the given commit object.

```
--- alpha
+++ alpha
```

Both the old (---) and new (*+*) versions of the file are called *alpha*.

```
@@ -1 +1 @@
```

This line is a "unified diff" chunk header. The @@ symbols are markers which allow the chunk header to be recognized as such when reading the unified diff. The negative number (here *-1*) indicates the number of lines present in the previous version of the chunk. The positive number (here *+1*) indicates the number of lines present in the new version of the chunk.

```
-alpha
+changed alpha
```

Following the chunk header are lines of file content related to the chunk. As indicated in the chunk header, there is one line which was present in the previous version of the file but has been removed in the new version of file:

```
-alpha
```

And there is one line which is instead present in the new version of the file:

```
+changed alpha
```

Note
"Diffs" are also known as "patches" because there is a well-known *patch* command (originally written by Larry Wall of Perl fame) which can read unified diffs and apply the differences to copies of the original version of files. Game of Trees has a built-in *got patch* command for this purpose, partly based on Larry Wall's implementation. We will learn more about it later.

7.7 Checking changes into a repository

To record a new snapshot which records the change we have made, run:

```
got commit
```

The editor should open again, showing an empty log message with some comments, similar to how *got import* behaved:

```
# changes to be committed on branch main:
#  M  /alpha
# detailed changes can be viewed in /tmp/got-Rmx4pOEFQy.diff
```

Write a log message describing the rationale of the changes you have made, (such as "change file alpha in accordance with tutorial instructions". Then save the file and exit the editor.

The *got commit* command should produce output similar to:

```
M  alpha
Created commit 47439e3bfa8445eb5aa0d939b9f342bae536bc40
```

The change has now been recorded in the repository.

7.8 Viewing changes in the repository

The snapshots recorded in repository history can be viewed with the *got log* command:

```
got log
```

This should print two entries, one for each commit present in our repository:

```
  ------------------------------------------------
  commit 47439e3bfa8445eb5aa0d939b9f342bae536bc40 (main)
  from: Tom Smith <tom.smith@example.com>
  date: Fri Aug 28 15:04:57 2026 UTC

   change file alpha in accordance with tutorial instructions

  ------------------------------------------------
  commit 53ed49f7cd92f7c3223838d856bb58cbb0e81e5d
  from: Tom Smith <tom.smith@example.com>
  date: Fri Aug 28 12:32:30 2026 UTC

   initial import
```

The head commit of the *main* branch is marked with the branch name in parenthesis next to the head commit's identifier.

By default, only log messages are displayed. It is also possible to include the list of files modified in each commit by running:

```
got log -d
```

The *-d* flag adds diff statistics to the output:

```
  ------------------------------------------------
  commit 47439e3bfa8445eb5aa0d939b9f342bae536bc40 (main)
  from: Tom Smith <tom.smith@example.com>
  date: Fri Aug 28 15:04:57 2026 UTC

   change file alpha in accordance with tutorial instructions

   M  alpha  |  1+  1-

  1 file changed, 1 insertion(+), 1 deletion(-)

  ------------------------------------------------
  commit 53ed49f7cd92f7c3223838d856bb58cbb0e81e5d
  from: Tom Smith <tom.smith@example.com>
  date: Fri Aug 28 12:32:30 2026 UTC

   initial import

   A  alpha         |  1+  0-
```

```
  A  beta          |  1+  0-
  A  epsilon/zeta  |  1+  0-
  A  gamma/delta   |  1+  0-

 4 files changed, 4 insertions(+), 0 deletions(-)
```

It is also possible to view changes to file content by passing the *-p* (patch) option to *got log*, either alone or together with *-d*. For example, run:

```
got log -p
```

The output should be similar to the following. Here, the unified diff headers indicate that we are viewing differences between two commit objects, rather than differences in a work tree.

```
-----------------------------------------------
commit 47439e3bfa8445eb5aa0d939b9f342bae536bc40 (main)
from: Tom Smith <tom.smith@example.com>
date: Fri Aug 28 15:04:57 2026 UTC

 change file alpha in accordance with tutorial instructions

diff 53ed49f7cd92f7c3223838d856bb58cbb0e81e5d 47439e3bfa8445eb5aa0d939b9f342bae536bc40
commit - 53ed49f7cd92f7c3223838d856bb58cbb0e81e5d
commit + 47439e3bfa8445eb5aa0d939b9f342bae536bc40
blob - 4a58007052a65fbc2fc3f910f2855f45a4058e74
blob + df6f19ffe953285ed6c4ff4f1568673a4b0ee0d6
--- alpha
+++ alpha
@@ -1 +1 @@
-alpha
+changed alpha

-----------------------------------------------
commit 53ed49f7cd92f7c3223838d856bb58cbb0e81e5d
from: Tom Smith <tom.smith@example.com>
date: Fri Aug 28 12:32:30 2026 UTC

 initial import

diff /dev/null 53ed49f7cd92f7c3223838d856bb58cbb0e81e5d
commit - /dev/null
commit + 53ed49f7cd92f7c3223838d856bb58cbb0e81e5d
blob - /dev/null
blob + 4a58007052a65fbc2fc3f910f2855f45a4058e74 (mode 644)
--- /dev/null
+++ alpha
@@ -0,0 +1 @@
+alpha
blob - /dev/null
blob + 65b2df87f7df3aeedef04be96703e55ac19c2cfb (mode 644)
--- /dev/null
+++ beta
@@ -0,0 +1 @@
+beta
blob - /dev/null
blob + fd08df0afa4d1d3faece37798d169e5a46d9d3fd (mode 644)
--- /dev/null
+++ epsilon/zeta
@@ -0,0 +1 @@
+zeta
blob - /dev/null
```

```
blob + ab135eefea6f73b921c7fec469b5f0e9db86b910 (mode 644)
--- /dev/null
+++ gamma/delta
@@ -0,0 +1 @@
+delta
```

When diffs become long, it will be useful to pipe the output of *got log* into a pager program. For example:

```
got log -d -p | less
```

This should display output similar to the above in a pager, allowing the user to scroll through.

Note
Unlike Git, Game of Trees does not invoke a pager automatically. One reason for this is that the file system view of *got* is restricted to the Git repository and/or the Work Tree via the *pledge* and *unveil* mechanisms of OpenBSD. Mixing this with the ability to run arbitrary programs would require relaxing these protection mechanisms in risky ways.

7.9 Committing changes from multiple work trees

Obtain a second work tree from the *firststeps.git* repository:

```
cd ~/tutorial
got checkout firststeps.git firststeps2
```

We can independently modify files in each work tree, and check our changes in.

First, commit a change to file *epsilon/zeta* from the newly created work tree:

```
cd firststeps2
echo changed zeta > epsilon/zeta
got commit
```

Now commit a change to file *gamma/delta* from the other work tree:

```
cd ../firststeps
echo changed delta > gamma/delta
got commit
```

Each work tree will now contain a different version of *epsilon/zeta* and *gamma/delta*, respectively. Run this to confirm:

```
cd ..
diff -u firststeps/gamma/delta firststeps2/gamma/delta
```

The above command should print something like:

```
--- firststeps/gamma/delta      Sun Aug 30 14:47:46 2026
+++ firststeps2/gamma/delta     Sun Aug 30 14:45:23 2026
@@ -1 +1 @@
-changed delta
+delta
```

Likewise, comparing epsilon/zeta between the two work trees yields a difference:

```
diff -u firststeps/epsilon/zeta firststeps2/epsilon/zeta
```

The above command should print something like:

```
--- firststeps/epsilon/zeta     Fri Aug 28 15:32:50 2026
+++ firststeps2/epsilon/zeta    Sun Aug 30 14:46:55 2026
@@ -1 +1 @@
-zeta
+changed zeta
```

This shows that *epsilon/zeta* was only changed in one work tree, while *gamma/delta* was only changed in the other work tree.

This might seem obvious, but there are some non-obvious consequences to consider:

In the latest commit found inside the repository, both files have been changed. We can use the *got cat* command to confirm this:

```
cd firststeps.git
got cat gamma/delta epsilon/zeta
```

The above should print:

```
changed delta
changed zeta
```

Both changes are now part of the branch both work trees where checked out from. Run this to confirm:

```
got log -l 2
```

The output should be similar to the following:

```
  ------------------------------------------------
  commit be218ab296caa9100355a42eef60f6c0a5708921 (main)
  from: Tom Smith <tom.smith@example.com>
  date: Sun Aug 30 12:48:40 2026 UTC

   change delta

  ------------------------------------------------
  commit d9034442c83401034bb5fb43738db1ea709e9997
  from: Tom Smith <tom.smith@example.com>
  date: Sun Aug 30 12:46:59 2026 UTC

   changing zeta
```

We can see that:

- In the repository's timeline, the change to *epsilon/zeta* occurred before the change to *gamma/delta*. From the repository's point of view, the second change was stacked upon the first change.
- Yet, despite this ordering of the timeline, we were able to create the second commit ("change delta") from an unrelated work tree which lacked the changes made in the first commit.
- While both work trees started out with a complete set of files copied from the same base commit, the work trees have now diverged. Neither work tree contains a complete copy of all the files anymore, as they are seen in the latest snapshot recorded in the repository.

We will learn some details about this an upcoming section called *The Game of Trees Work Tree*. For now, considering the consequences of the behaviour observed, let's conclude that committing changes is equivalent to a *write-only* operation on the repository. The files we changed were written to the repository by the *got commit* operation, but nothing was ever read back from the repository during the *got commit* operation. Reading changes from the repository is not a task of the *got commit* command. This task is left to another command, which is called *got update*.

7.10 Examining Work Tree Meta-Data

Switch to a work tree directory and run the *got info* command:

```
cd ../firststeps
got info
```

The *got info* command prints the work tree's meta-data. This meta-data is stored in a hidden *.got* directory inside the work tree:

```
work tree: /home/user/tutorial/firststeps
work tree base commit: 53ed49f7cd92f7c3223838d856bb58cbb0e81e5d
work tree path prefix: /
work tree branch reference: refs/heads/main
work tree UUID: b8cb4d1a-d022-4d72-8f2b-b1f990db35ec
work tree format version: 1
file index version: 3
repository: /home/user/tutorial/firststeps.git
```

For our purposes, the interesting parts are the following:

```
work tree base commit: 53ed49f7cd92f7c3223838d856bb58cbb0e81e5d
```

The work tree base commit indicates which commit the files in work tree where checked out from.

```
work tree branch reference: refs/heads/main
```

The work tree branch reference indicates which branch the work tree was checked out from.

```
repository: /home/user/tutorial/firststeps.git
```

The work tree records the absolute path of the repository it was checked out from. In case the repository is moved, the path can be adjusted by editing the file *.got/repository* in the work tree. Alternatively, a new work tree can be checked out from the moved repository.

7.11 Mixed-Commit Work Trees

As seen in the previous section, the work tree records the identifier of the commit from which all files in the work tree were initially obtained as the global **base commit**.

Additionally, for each file, separate base commit and **base blob** identifiers are recorded. This can be seen by passing a file path to *got info*. For example, run:

```
got info beta
```

This should display the same global work tree meta-data as before, following by meta-data specific to the file at path *beta*:

```
file: beta
mode: 644
timestamp: Fri Aug 28 15:32:50 2026 CEST
based on blob: 65b2df87f7df3aeedef04be96703e55ac19c2cfb
based on commit: 53ed49f7cd92f7c3223838d856bb58cbb0e81e5d
```

We can see that a base commit identifier is recorded per-file. This item of meta-data is distinct from the work tree's global base commit identifier. There is no requirement that all files in the work tree must be based on their versions as found in the single commit which was initially checked out. We can mix and match files from distinct commits in the work tree.

This can be useful in some workflows, e.g. to reset some files to an earlier version if that helps us with searching for the root cause of a problem in a code base.

Game of Trees makes use of this flexibility internally when *got commit* is run several times in succession in order to record multiple snapshots. Each subsequent commit operation only writes a subset of the changes from the work tree to the Git repository. Files modified in a given commit have their base commit identifier updated accordingly, while meta-data for unchanged files is left alone.

This works as long as all files being committed are "up-to-date". Before a file is committed to the repository, its base blob identifier is compared to the blob identifier recorded in the current branch head in the repository. If these identifiers differ then there are new changes for the file waiting to be fetched from the repository into the work tree, and the commit operation will fail.

7.12 Obtaining newer changes from a repository

Fetch new changes from the repository into the second work tree. Run:

```
cd ../firststeps2
got update
```

This should print something like:

```
U  gamma/delta
Updated to refs/heads/main: be218ab296caa9100355a42eef60f6c0a5708921
```

This work tree now accurately represents the state of the *main* branch's head commit again, after having diverged while working on changes for the file *epsilon/zeta*.

7.13 Handling out-of-date files in the work tree

As mentioned previously, committing to files which are based on out-of-date versions should not be possible. In this section we examine how Game of Trees behaves when we try.

Modify *gamma/delta* in both work trees, attempting to commit each time:

```
cd ../firststeps
echo new change for file delta > gamma/delta
got commit
```

The above commit from the first work tree should succeed as before.

Attempting the same from the second work tree should result in an error:

```
cd ../firststeps2
echo another change for file delta > gamma/delta
got commit
```

This commit operation should fail with the following error:

```
got: work tree must be updated before these changes can be committed
```

Run the *got update* command to fetch the new changes for *gamma/delta* into this work tree:

```
got update
```

The output should be similar to:

```
C  gamma/delta
Updated to refs/heads/main: bfd8d082c8806e3e1787648ef5c29fba99f7c45a
Files with new merge conflicts: 1
```

The new changes have been merged with the local (uncommitted) changes. Because the changes overlap (there is only one line in the file which was modified on both sides), a merge conflict has been recorded in the file, as per the diff3 algorithm mentioned earlier.

Examine the contents of *gamma/delta*. Run:

```
cat gamma/delta
```

The contents should be similar to the following:

```
<<<<<<< merged change: commit bfd8d082c8806e3e1787648ef5c29fba99f7c45a
new change for file delta
||||||| 3-way merge base: commit be218ab296caa9100355a42eef60f6c0a5708921
changed delta
=======
another change for file delta
>>>>>>>
```

The file now contains **conflict markers**, displaying all 3 versions of lines as seen by the diff3 algorithm. (For simplicity, this example uses a single line in all versions of the file. But conflicts across overlapping regions of multiple lines are more usual in practice, and will be surrounded by conflict markers just as well.)

The first version displayed contains the version which was newly fetched into the work tree from the repository:

```
new change for file delta
```

The second version displayed contains the version the other two versions were based on:

```
changed delta
```

The third version displayed contains the version found in the work tree:

```
another change for file delta
```

Trying to commit a file with conflict markers should fail. Before removing the conflict markers from the file, run:

```
got commit
```

The output should be:

```
C  gamma/delta
got: cannot commit file in conflicted status
```

While the version control system can detect overlapping modifications and flag them as merge conflicts, it does not know how conflicts can be resolved in a reasonable way. It's up to the user to modify conflicted sections in the file, ensuring that the merged result makes sense and matches their intentions.

Open *gamma/delta* in and editor and modify the content to your liking, removing the lines with conflict markers in the process, which are:

```
<<<<<<< merged change: commit bfd8d082c8806e3e1787648ef5c29fba99f7c45a
||||||| 3-way merge base: commit be218ab296caa9100355a42eef60f6c0a5708921
=======
>>>>>>>
```

Note
Unlike Git, Game of Trees does not require a specific command to be run in order to clear conflicted file status. Removing conflict markers from the file's content is sufficient.

Examine the result of your changes with *got diff* to verify that the changes you are about to commit match your intentions.

The commit operation should now succeed. Run:

```
got commit
```

The output should indicate a successful commit operation:

```
M  gamma/delta
Created commit df66b4cca406dfcd08281bdfb481a92139d65731
```

Chapter 8

Interacting with remote repositories

So far we have been working with a single repository. In this section we will examine ways to synchronize changes between repositories across a network connection.

Synchronization relies on the following Git repository format concepts:

- **Objects**: Objects of all types are copied between repositories by bundling them into a **pack file** which is sent across the network. The pack file format is also used for local on-disk storage. Any objects copied across the network from remote repositories will usually end up in a pack file stored on local disk.
- **References**: References are used to make objects in the downloaded pack file discoverable. By convention, the reference namespace *remotes/* is reserved for everything which was copied from remote repositories. References within this namespace essentially serve as a local cache of whatever the remote repository contained when it was last visited for synchronization.
- **Remote**: Each **remote** repository is referred to by a user-defined handle assigned via the local repository's configuration file. The default remote repository is usually called "origin". Local copies of the default remote repository's branches are reachable via references stored in the local repository's *remotes/origin/* reference namespace. When "origin" is the desired remote repository to use then its handle may be omitted on the command line. Otherwise, the handle of the remote repository to use must be specified.
- **Tags**: By convention, the reference namespace *refs/tags/* is shared between all repositories. Tags have unique names which are mirrored to all repositories. This implies that once a tag has been copied from the repository in which it was created to another repository, the tag should never be changed again. While there are ways to overwrite existing tags, it is usually better to make a new tag with a new name instead (for example, by increasing the release version number in the tag's name).

8.1 Simulating a remote Git server

For the purposes of this tutorial we will simulate remote repositories by connecting to localhost over SSH.

The Git repository server programs which Game of Trees will invoke across SSH connections are Git's *git-upload-pack* and *git-receive-pack* commands. Please ensure that these programs are installed on your system, and install Git from your operating system's package manager if they are missing.

Next, generate a temporary SSH key which can be removed again once we are done with this tutorial.

```
ssh-keygen -f ~/.ssh/id_got_tutorial
```

When *ssh-keygen* asks for a passphrase, you can leave the passphrase empty by pressing Enter twice.

Add the temporary SSH key to the end of your *.ssh/authorized_keys* file:

```
cat ~/.ssh/id_got_tutorial.pub >> ~/.ssh/authorized_keys
```

Connect once and accept the host key fingerprint if needed:

```
ssh -i ~/.ssh/id_got_totorial localhost
```

This SSH command should lead you to a new shell prompt on your own machine. If that worked, run:

```
exit
```

and SSH should print:

```
Connection to localhost closed.
```

To avoid the need for passing a temporary ssh key on the ssh command line every time, add the following lines at the end of *~/.ssh/config*:

```
Host localhost
    IdentityFile ~/.ssh/id_got_tutorial
```

These lines can be removed again once we are done with this tutorial.

The following command should now succeed as before, without having to pass the *-i* option on the command line:

```
ssh localhost
```

8.2 Configuring remote repositories

Create a second Git repository in your tutorial folder:

```
cd ~/tutorial
got init remoterepo.git
```

We leave this new repository empty for now, and configure Game of Trees to refer to it as a remote repository from the *firststeps.git* repository.

Create the file *~/tutorial/firststeps.git/got.conf* with the following contents:

```
remote "origin" {
       server localhost
       protocol ssh
       repository "~/tutorial/remoterepo.git"
}
```

This sets the default remote repository, called “origin” by convention, to *remoterepo.git* on the Git server *localhost*. (In the real world, the server name will rarely be *localhost*, of course. It will usually refer to some Git hosting site’s server instead.)

Note
More details about the *got.conf* configuration file are available in the *got.conf* manual page, or online at: https://gameoftrees.org/got.conf.5.html

---

8.3 Sending to remote repositories

With the remote repository configuration in place, we can now send all pending changes from *firststeps.git* to *remoterepo.git*:

```
cd firststeps.git
got send
```

A successful *got send* command should print something similar to:

```
Connecting to "origin" ssh://localhost/~/tutorial/remoterepo.git
6 commits colored; 27 objects found; 12 trees scanned
packing 1 reference; 27 objects; deltify: 100%; uploading pack:  2.2K 100%
Server has accepted refs/heads/main
```

Inspect commit history of the *firststeps.git* and keep it in mind for later reference:

```
got log -s
```

Inspect commit history of the *remoterepo.git* repository, which should now contain the same history as the *firststeps.git* repository:

```
cd ../remoterepo.git
got log -s
```

List the references which exist in *remoterepo.git*, using the *got ref* command:

```
got ref -l
```

The output should be similar to:

```
HEAD: refs/heads/main
refs/heads/main: df66b4cca406dfcd08281bdfb481a92139d65731
```

As expected, this repository now contains the *main* branch we have sent to it.

List the references which exist in *firststeps.git*:

```
cd ../firststeps.git
got ref -l
```

This should print output similar to:

```
HEAD: refs/heads/main
refs/got/worktree/base-53aea438-cb0e-4252-9ac9-88349e3fe7f9:  ↩
    df66b4cca406dfcd08281bdfb481a92139d65731
refs/got/worktree/base-b8cb4d1a-d022-4d72-8f2b-b1f990db35ec:  ↩
    bfd8d082c8806e3e1787648ef5c29fba99f7c45a
refs/heads/main: df66b4cca406dfcd08281bdfb481a92139d65731
refs/remotes/origin/main: df66b4cca406dfcd08281bdfb481a92139d65731
```

Compared to *remoterepo.git*, there are three additional references visible. Two references in the *refs/got/* namespace, and one reference in the *refs/remote/* namespace.

These references are in the *refs/got/* reference namespace, which is automatically managed by Game of Trees:

```
refs/got/worktree/base-53aea438-cb0e-4252-9ac9-88349e3fe7f9:  ↩
    df66b4cca406dfcd08281bdfb481a92139d65731
refs/got/worktree/base-b8cb4d1a-d022-4d72-8f2b-b1f990db35ec:  ↩
    bfd8d082c8806e3e1787648ef5c29fba99f7c45a
```

These references point at commits which the two work trees in our *~/tutorial* directory are using as their global base commit at present. We can ignore these references for now. They exist to prevent commits in active use by work trees from being garbage-collected by programs such as *git gc* or *gotadmin cleanup*.

For our current purposes, this other additional reference is far more interesting:

```
refs/remotes/origin/main: df66b4cca406dfcd08281bdfb481a92139d65731
```

This reference was created by the successful *got send* operation above and represents changes most recently sent to, or most recently fetched from, the *main* branch of the remote repository *remoterepo.git*, which we labeled "origin".

8.4 Fetching from remote repositories

Commit another change to the *firststeps.git* repository:

```
cd ..
cd firststeps
echo change file alpha again > alpha
got commit
```

The above *got commit* command should succeed with output similar to:

```
M  alpha
Created commit 3e3adcc09dfb44cb13f58f1de5d40430af6a7ce4
```

Create a work tree for *remoterepo.git* and commit a new change from there:

```
cd ..
got checkout remoterepo.git
cd remoterepo
echo changed file beta > beta
got commit
```

The above *got commit* command should succeed with output similar to:

```
M  beta
Created commit 345d47cea97c4d0c47629b0b831fabdd9b03138f
```

The remote repository *remoterepo.git* now has new changes which are missing in the *firststeps.git* repository, and vice versa.

In the *firststeps.git* repository, can use the *got fetch* command to fetch changes from the default "origin" remote:

```
cd ../firststeps.git
got fetch
```

Note
The *got fetch* and *got send* commands can be run either in a repository directory or in a work tree. When invoked in a work tree, these commands operate on the repository associated with the work tree. The work tree itself is not affected.

The above *got fetch* command should succeed with output similar to:

```
Connecting to "origin" ssh://localhost/~/tutorial/remoterepo.git
server: Enumerating objects: 3, done.
server: Counting objects: 100% (3/3), done.
server: Compressing objects: 100% (2/2), done.
server: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
 354B fetched; indexing 100%
Fetched ae085cb72e0455886f41c8525ca40208b3ace700.pack
Updated refs/remotes/origin/main: 345d47cea97c4d0c47629b0b831fabdd9b03138f
Created reference refs/remotes/origin/HEAD: refs/remotes/origin/main
```

The most important line of the above output is:

```
Updated refs/remotes/origin/main: 345d47cea97c4d0c47629b0b831fabdd9b03138f
```

The *firststeps.git* repository now contains a copy of the changes which were committed to the *main* branch of *remoterepo.git* above, visible under the reference *refs/remotes/origin/main*.

Inspect commit history of the *refs/remotes/origin/main* in the *firststeps.git* repository:

```
got log -c refs/remotes/origin/main -s
```

Here, we are asking *got log* to start traversing history at the commit resolved via the reference *refs/remotes/origin/main*.

As a side-note, the following command is equivalent to the above:

```
got log -c origin/main -s
```

To save us some typing effort on the command line, Game of Trees allows us to specify non-absolute reference names, i.e. names without the leading *refs/* component. The specified name will first be searched in the *refs/heads/* namespace, then in the *refs/tags/* namespace, and finally in the *refs/remotes/* namespace. This avoids having to type those reference prefixes every time.

The output of the above *got log* command should match the output of this command, which displays the history of the *refs/heads/main* branch in the *remoterepo.git* repository.

```
got log -r ../remoterepo.git -s
```

It is important to highlight that two distinct branch heads for *main* now exist in *firststeps.git*: One which was committed directly to *firststeps.git*, and one which was committed to *remoterepo.git* and then fetched into *firststeps.git*. Both branch heads represent distinct changes which have not been merged yet. We will perform a merge of the changes in the next section.

8.5 Merging changes fetched from remote repositories

Much like Git, Game of Trees provides two operations for merging changes. Both operations are explained below. We will use the first one in this section of the tutorial, and the second one in a later section.

Note
If the explanations below do not make a lot of sense, asking a knowledgable ally to draw some pictures might help.

Merge: The **merge** operation takes two branch heads (*main* and *origin/main* in our example), then merges files from both heads and their common ancestor commit using the diff3 algorithm, and creates a new snapshot which contains the merged files. The commit object created for this snapshot refers to both two branch heads via **two parent commits**. This approach leaves the intrinsic identifiers of both branch heads unchanged.

Rebase: The **rebase** operation builds a list of all commits between one of the two branch heads (the "rebased" one) and the common ancestor commit of both branches. The rebase operation then traverses this list of commits, and merges the changes

found in each commit on top of the other branch head (the "base" one) in sequence, using the diff3 algorithm. Each time, the common ancestor used for diff3 is the commit just below the head commit of the "rebased" branch. The commit object created for the resulting snapshot is the tip of a chain of commits, all of which have a **single parent commit**, which ultimately ends at the head commit of the "base" branch. While the "base" branch head gets to keep its intrinsic identifier, all commits of the "rebased" branch head are rewritten with their parent commit adjusted, which changes their intrinsic identifiers.

The question of whether to merge or rebase changes is largely philosophical. When contributing to a Git-repository-based software project in the real world, there will usually be guidance as to which strategy is preferred.

Nevertheless, the following rules of thumb will usually apply:

- If relevant project maintainers prefer to see all of their project's history as a flat linear history on a single main branch, always use **rebase** to make your own changes sit on top of the latest head commit of the main branch, before sending with *got send* or otherwise submitting your changes to the project.
- If you are not able to send changes to the remote repository yourself, use **rebase** and propose your changes for inclusion via appropriate means (e.g. by sending patches or pull requests).
- If you are able to send changes to the remote repository directly, and this remote repository is supposed to represent the main hub of the project, which everyone is synchronizing changes from before beginning new work, then using **merge** commits will be fine.

8.6 Creating merge commits

We can use the *got merge* command to merge outstanding changes reachable from the *refs/remotes/origin/main* branch head into a work tree checked out from *refs/heads/main*.

First, verify that your work tree of the *firststeps.git* repository is indeed on the *main* branch:

```
got branch
```

The output should be:

```
main
```

The work tree also needs to be a fully up-to-date single-commit work tree. Run the *got update* command to ensure this:

```
got update
```

The work tree status should be clean, without any uncommitted modifications present. Verify this with the *got status* command:

```
got status
```

If the *got status* command reports nothing then the work tree is clean. If there are any pending changes you want to keep, commit them now and the run *got update* again once more.

The work tree is now ready to merge changes from the remote repository. Run:

```
got merge origin/main
```

The output should be similar to:

```
G  beta
Merged refs/remotes/origin/main into refs/heads/main: 67 ↩
    fe73fbe97fe683ab162b05b1a6550209746569
```

Note
In this case, the *got merge* commands created a merge commit immediately, This is the default behaviour. You could pass the *-n* option to *got merge* if you would rather inspect or adjust the merged changes before the merge commit gets created, and run *got merge -c* the continue the interrupted merge operation once you are happy with the changes in the work tree. The merge operation will always be interrupted in case of merge conflicts, requiring manual conflict resolution followed by *got merge -c*.

This resulting merge commit can be inspected with the *got log* command:

```
got log -l 1
```

The above *got log* command should show a commit which lists two parent commits:

```
-----------------------------------------------
commit 67fe73fbe97fe683ab162b05b1a6550209746569 (main)
from: Tom Smith <tom.smith@example.com>
date: Thu Sep  3 08:18:26 2026 UTC
parent 1: 3e3adcc09dfb44cb13f58f1de5d40430af6a7ce4
parent 2: 345d47cea97c4d0c47629b0b831fabdd9b03138f

 merge refs/remotes/origin/main into refs/heads/main
```

The first parent commit is the base commit of the work tree in which the merge was performed.

The second parent commit is the branch head of the remote repository's *main* branch, as cached locally in *refs/remotes/origin/-main*.

By adding the *-p* option to the *got log* command we can examine the changes added to our local *main* branch during the merge:

```
got log -l 1 -p
```

The output should be similar to:

```
-----------------------------------------------
commit 67fe73fbe97fe683ab162b05b1a6550209746569 (main)
from: Tom Smith <tom.smith@example.com>
date: Thu Sep  3 08:18:26 2026 UTC
parent 1: 3e3adcc09dfb44cb13f58f1de5d40430af6a7ce4
parent 2: 345d47cea97c4d0c47629b0b831fabdd9b03138f

 merge refs/remotes/origin/main into refs/heads/main

diff 3e3adcc09dfb44cb13f58f1de5d40430af6a7ce4 67fe73fbe97fe683ab162b05b1a6550209746569
commit - 3e3adcc09dfb44cb13f58f1de5d40430af6a7ce4
commit + 67fe73fbe97fe683ab162b05b1a6550209746569
blob - 65b2df87f7df3aeedef04be96703e55ac19c2cfb
blob + 90f9e34b226a6ff78047f128051c7bb77c49a270
--- beta
+++ beta
@@ -1 +1 @@
-beta
+changed file beta
```

The changes shown in the patch below the commit's log message should correspond to the changes which were originally committed from the work tree of the *remoterepo.git* repository.

8.7 Sending merged changes to remote repositories

We can now send our changes to *remoterepo.git*, along with the merge commit we just created:

```
got send
```

There is no need to pass any arguments to *got send* in this case, The current work tree's branch will be sent by default, which is the branch we have merged new changes into. And we are sending to the default "origin" remote repository, so we do not need to specify a remote repository handle either.

The output should be similar to:

```
Connecting to "origin" ssh://localhost/~/tutorial/remoterepo.git
9 commits colored; 27 objects found; 15 trees scanned
packing 1 reference; 27 objects; deltify: 100%; uploading pack:  2.7K 100%
Server has accepted refs/heads/main
```

8.8 Updating work trees across merge commits

Update the work tree of *remoterepo.git* to to make the sent changes visible there:

```
cd ../remoterepo
got update
```

The above *got update* command might complain as follows:

```
got: work tree's head reference now points to a different branch; new head reference and/or ↩
     update -b required
```

Ideally, this would not happen and the update would succeed. The update fails because secondary parents of merge commits are currently skipped during the *got update* commit ancestry check for performance reasons. Until this behaviour is improved, the *-b* option can be used as a workaround, which explicitly tells Game of Trees that we are fine with updating to the head of the current *main* branch as seen in the repository:

```
got update -b main
```

The output should be similar to:

```
Switching work tree from 345d47cea97c4d0c47629b0b831fabdd9b03138f to refs/heads/main
U  alpha
Updated to refs/heads/main: 67fe73fbe97fe683ab162b05b1a6550209746569
```

The change to file *alpha* originally committed to *firststeps.git* is now visible in this work tree of *remoterepo.git*. Verify this by running:

```
got log -l1 -p alpha
```

The output should be similar to the following, and display the change you made to file *alpha* in *firststeps.git* earlier:

```
 ------------------------------------------------
 commit 3e3adcc09dfb44cb13f58f1de5d40430af6a7ce4
 from: Tom Smith <tom.smith@example.com>
 date: Wed Sep  2 15:08:33 2026 UTC

  changing file alpha
```

```
diff df66b4cca406dfcd08281bdfb481a92139d65731 3e3adcc09dfb44cb13f58f1de5d40430af6a7ce4
commit - df66b4cca406dfcd08281bdfb481a92139d65731
commit + 3e3adcc09dfb44cb13f58f1de5d40430af6a7ce4
blob - df6f19ffe953285ed6c4ff4f1568673a4b0ee0d6
blob + 25bb78dc2da9598b038b840a226f5651a83f3df5
--- alpha
+++ alpha
@@ -1 +1 @@
-changed alpha
+change file alpha again
```

8.9 Cloning a remote repository

Our current *remoterepo.git* started out as an empty repository which we sent changes to from our local repository *firststeps.git*. This is a common scenario when starting a new project and then publishing it on other Git servers. But the far more common scenario is that users want to clone a repository someone else has created and populated with interesting content.

The *got clone* command can be used to clone remote repositories to the local disk. We can clone our *remoterepo.git* as follows:

```
cd ~/tutorial
got clone ssh://localhost/~/tutorial/remoterepo.git localrepo.git
```

The output should be:

```
Connecting to ssh://localhost/~/tutorial/remoterepo.git
server: Enumerating objects: 35, done.
server: Counting objects: 100% (35/35), done.
server: Compressing objects: 100% (18/18), done.
server: Total 35 (delta 6), reused 0 (delta 0), pack-reused 0 (from 0)
 2.7K fetched; indexing 100%; resolving deltas 100%
Fetched 468d49acc1a6798081ad7e91111eb59ea74e3265.pack
Created cloned repository 'localrepo.git'
```

There is now an identical clone of *remoterepo.git* in *localrepo.git*. We can compare the set of references to see this:

```
got ref -l -r remoterepo.git
```

The list of references in *remoterepo.git* should be similar to:

```
HEAD: refs/heads/main
refs/got/worktree/base-5d9e61df-8855-485c-9423-74a18b696f2e: 67 ↩
    fe73fbe97fe683ab162b05b1a6550209746569
refs/heads/main: 67fe73fbe97fe683ab162b05b1a6550209746569
```

Obtain the list of references in *localrepo.git* for comparison:

```
got ref -l -r localrepo.git
```

This list should be similar to:

```
HEAD: refs/heads/main
refs/heads/main: 67fe73fbe97fe683ab162b05b1a6550209746569
refs/remotes/origin/HEAD: refs/remotes/origin/main
refs/remotes/origin/main: 67fe73fbe97fe683ab162b05b1a6550209746569
```

While the *refs/heads/main* branch head reference are identical on both sides, there are some differences:

- There is a *refs/got/worktree/* reference in *remoterepo.git* because we did check out a work tree from it earlier.
- There are references in *refs/remotes/* in the freshly cloned *localrepo.git* which do not exist in *remoterepo.git* as such. As before, these references provide a cached view of the remote repository as seen during the *got clone* operation.

Additionally, *got clone* has created a *got.conf* configuration file in *localrepo.git* which defines the default "origin" remote repository handle for us, so we do not need to add it manually. Use the *cat* command to display this configuration file:

```
cat localrepo.git/got.conf
```

The output should be:

```
remote "origin" {
        server localhost
        protocol ssh
        repository "~/tutorial/remoterepo.git"
        fetch {
                branch { "main" }
        }
}
```

This looks similar to the *got.conf* file we created manually earlier. The only addition is a *fetch* configuration block, which instructs *got fetch* to always fetch the default *main* branch from the remote repository, regardless of which branch is currently checked out in the work tree which *got fetch* is invoked in.

Chapter 9

Managing files in the work tree

In this section we cover commands which manipulate files in the work tree in ways that go beyond just changing the content of files.

9.1 Adding additional files to version control

By default, Game of Trees ignores additional files which appear in the work tree. Files will be tracked under version control only if we explicitly ask for them to be tracked.

Create a new file in a work tree of *localrepo.git*:

```
cd ~/tutorial
got checkout localrepo.git
cd localrepo
echo this is a new file called psi > psi
```

Examine the current work tree status with the *got status* command:

```
got status
```

The output should be:

```
?  psi
```

In the above output, the *?* status marker refers to files which are "unversioned", i.e. not tracked under version control. While Game of Trees notices such files, it will ignore them. In fact, such files are assumed to belong to the user and not the version control system. Game of Trees will never touch such files without an explicit request, even if such files could be affected by operations performed on this work tree. For instance, assuming some remote repository already tracks a file *psi* and has changes for it which we fetch, the changes destined for this file would be ignored if we were trying to merge them into this work tree. If we do not want the file *psi* to be ignored, we must add it to version control and commit it.

Unless changes to already tracked files are present in your work tree, trying to commit from the work tree at this point should fail:

```
got commit
```

The output should be:

```
got: no changes to commit
```

Adding new files to version control is done with the *got add* command:

```
got add psi
```

The output of the *got add* command should be:

```
A  psi
```

The *A* status marker means the file is now in "added" state, ready to become part of the next snapshot we create in the repository. While the file is marked for addition, its content is not yet stored in the repository. Any further changes made to the file while in "added" state will become part of the next snapshot created with *got commit*. Removing the file from disk at this point would still cause its content to be lost.

Commit the addition of file *psi* by running:

```
got commit
```

The editor window which opens should contain a log message template which mentions file *psi* being added:

```
# changes to be committed on branch main:
#  A  /psi
# detailed changes can be viewed in /tmp/got-pESrCsKKgr.diff
```

Enter a suitable log message, save, and exit the editor.

The commit should succeed with output similar to the following:

```
A  psi
Created commit de2057b74aea606fe978ab4dff3eef80a3d651d9
```

The file *psi* is now part of the latest snapshot stored in the repository. Listing files and directories in the snapshot should now display the file:

```
got tree
```

The output should be:

```
alpha
beta
epsilon/
gamma/
psi
```

9.2 Hiding unversioned files from status output

If your work tree ends up contains a lot of unversioned files (e.g. because software compilation produces a lot of derived build artifacts), the amount of lines reported the *?* status marker can drown out useful status information about files under version control.

Game of Trees supports 'ignore" lists to hide such files from the output of *got status*. Ignore lists are stored in a file with the special filename *.gitignore* (and, for compatibility with the OpenBSD CVS repository, *.cvsignore*). A *.gitignore* file can appear in any directory of the work tree and applies to unversioned files located within this directory, and can optionally also apply to unversioned files anywhere below this directory.

The *.gitignore* file provides patterns which match unversioned file names that should be ignored. One pattern can be defined per line.

In your work tree of *localrepo.git*, add another unversioned file:

```
cd ~/tutorial/localrepo
echo This file should be ignored > ignore-me
```

Running *got status* at this point should result in the file being displayed as unversioned:

```
got status
```

The output should be:

```
?  ignore-me
```

Create a *.gitignore* file which matches all file names ending with "-me":

```
echo '*-me' > .gitignore
```

Run *got status* again:

```
got status
```

The output should now be:

```
?  .gitignore
```

Add the file *.gitignore* to version control and commit it:

```
got add .gitignore
got commit
```

Ignored files will be hidden from the output of *got status* unless the *-I* option of *got status* is used. Try:

```
got status -I
```

With *-I*, the expected output is:

```
?  ignore-me
```

9.3 Adding ignored files to version control

Attempt to add the *ignore-me* file to version control:

```
got add ignore-me
```

There should be no output, and the output of *got status* should remain empty as well.

Ignored files will not be added to version control by *got add*. unless the *-I* option is used to exceptionally add a file which matches an ignore pattern:

```
got add -I ignore-me
```

Now we should see:

```
A  ignore-me
```

Commit the addition of the *ignore-me* file. We will remove it again in the next section:

```
got commit
```

The output should be:

```
A  ignore-me
Created commit 926dc418635f0541c53d820b954a7ed063072946
```

9.4 Removing files from version control

In the previous section, we exceptionally added an ignored file to version control. This was probably a mistake, so let's remove it from version control again.

Removing files from version control is done with the *got remove* command. Run:

```
got remove ignore-me
```

The output should be:

```
D  ignore-me
```

The *D* status marker applies to files scheduled for deletion in the next commit. When a versioned file is removed with *got remove* it is also removed from disk.

We can now commit the deletion of ignore-me as usual:

```
got commit
```

The output of *got commit* should be similar to:

```
D  ignore-me
Created commit 7c1424929a9eaddcee8745ebc827651a766c450e
```

Since *ignore-me* was removed from disk, it is no longer an unversioned and ignored file. Run:

```
got status -I
```

The output should be empty.

9.5 Generating patch files

When contributing to Git repository you do not have write access to, the repository owner might ask you to send them a patch file.

Game of Trees can generated patch files in the "unified diff" format with the *got diff* command.

First, let's make another change to a file in the work tree:

```
echo add a line to file alpha >> alpha
```

View the change you have made to the work tree:

```
got diff
```

The output should be similar to:

```
diff /home/stsp/tutorial/localrepo
path + /home/stsp/tutorial/localrepo
commit - e7d64e8b65608cf6e3577527bbc967651d0641f4
blob - 25bb78dc2da9598b038b840a226f5651a83f3df5
file + alpha
--- alpha
+++ alpha
@@ -1 +1,2 @@
 change file alpha again
 +add a line to file alpha
```

Commit this change to the repository:

```
got commit
```

The output of the *got commit* command should be similar to:

```
M  alpha
Created commit 65ad026b2001e9d26a0e82fec8d4709bc5da826e
```

This commit should now be the head commit of the *main* branch.

So far, we have run *got diff* without any arguments to inspect uncommitted changes in the work tree. We can also use *got diff* to view changes that are already present in the repository.

For example, to see the changes in the head commit of the *main* branch, pass one *-c* option to *got diff* as follows:

```
got diff -c main
```

The output should be similar to the local change above, but with diff headers indicating that the patch is showing the difference between two commits: The *main* branch head, and the first parent commit of the *main* branch head:

```
diff 7c1424929a9eaddcee8745ebc827651a766c450e refs/heads/main
commit - 7c1424929a9eaddcee8745ebc827651a766c450e
commit + 65ad026b2001e9d26a0e82fec8d4709bc5da826e
blob - 25bb78dc2da9598b038b840a226f5651a83f3df5
blob + 7c942f4e44e84971edf730376487105606d0c7ae
--- alpha
+++ alpha
@@ -1 +1,2 @@
 change file alpha again
 +add a line to file alpha
```

Passing one *-c* option always results in a comparison with the specified commit's firt parent commit.

By passing two *-c* options, we can view differences between two arbitrary commits. For example, let's view a patch of all the changes we have made so far since cloning the *localrepo.git* repository:

```
got diff -c origin/main -c main
```

The output should be similar to the following:

```
diff refs/remotes/origin/main refs/heads/main
commit - 67fe73fbe97fe683ab162b05b1a6550209746569
commit + 65ad026b2001e9d26a0e82fec8d4709bc5da826e
blob - 25bb78dc2da9598b038b840a226f5651a83f3df5
blob + 7c942f4e44e84971edf730376487105606d0c7ae
--- alpha
+++ alpha
@@ -1 +1,2 @@
 change file alpha again
+add a line to file alpha
blob - /dev/null
blob + 44652d86bdd1c1b039bee6eedd4285ffa050adae (mode 644)
--- /dev/null
+++ .gitignore
@@ -0,0 +1 @@
+*-me
blob - /dev/null
blob + a3e542480f5fe385bbe7cd7afad1e0c2377999d6 (mode 644)
--- /dev/null
+++ psi
@@ -0,0 +1 @@
+this is a new file called psi
```

To obtain a patch file which describres all these changes, we can simply save the above output to a file. Usually, patch files carry the file extensions *.patch* or *.diff*, but this is an arbitrary choice. Run:

```
got diff -c origin/main -c main > /tmp/localrepo.diff
```

We can now send the file *localrepo.diff* to someone who has write access to another repository, and they will be able to apply our changes there and commit them.

9.6 Applying a patch file to a work tree

Let's apply our patch file to a work tree of the *firststeps.git* repository.

We could use the standard *patch* command for this task. However, the standard *patch* command is unaware of the Game of Trees work tree format. When the *got patch* command is used instead, file additions and removals described by the patch file are automatically recorded in the work tree. Additionally, if the object IDs mentioned in the patch file headers can be found in the local repository, *got patch* is able to merge files using the diff3 algorithm. The search-replace strategy known from the standard *patch* command is only used as a fallback. The benefit of this behaviour is that we get merge conflict markers in files instead of *.rej* (reject) files listing changes which could not be applied.

Switch to your work tree of *firststeps.git* and apply the patch there:

```
cd ../firststeps
got patch /tmp/localrepo.diff
```

The output of the *got patch* command should be:

```
G  alpha
A  .gitignore
A  psi
```

Status code *G* indicates that the file *alpha* was merged using the diff3 algorithm. Files patched using the search-replace strategy would appear with status code *M* instead.

The files *.gitignore* and *psi* were newly added by the patch, and are now already marked for addition with the next commit.

Commit the changes with:

```
got commit
```

The output should be:

```
A  .gitignore
A  psi
M  alpha
Created commit 6ce60cf9093008741619abc657026be7eb5b3d7e
```

This commit modifies all three files at once. We thus have differences in commit history between *localrepo.git* and *firststeps.git*, even though the changes contained within them are the same.

Chapter 10

Undoing changes

Sometimes changes turn out to be undesirable, and they need to be undone. Game of Trees provides two commands which can undo changes, depending on whether the changes to be undone are sitting in a work tree or have already been committed to the repository.

10.1 Undoing uncommitted changes in the work tree

Changes in the work tree can be undone with the *got revert* command. This command will make all versioned files appear as they did when the work tree was last checked out or updated.

This command needs to be used very carefully because **there is no way to bring discarded changes back after *got revert*!**

Schedule a few changes for commit in the work tree, but do not actually commit them. Modify a file, add a new file, and remove another file:

```
cd ~/tutorial/localrepo
echo modified alpha yet again > alpha
echo this is a new file > new
got add new
got remove beta
got status
```

The output of the above *got status* command should be:

```
M  alpha
D  beta
A  new
```

We can run *got revert* on one file at a time. Try running:

```
got revert alpha
```

The output should be:

```
R  alpha
```

And now alpha should no longer appear in the output of *got status*. Run:

```
got status
```

The output should be:

```
D  beta
A  new
```

We can also revert multiple files at once by passing them on the command line. Run:

```
got revert *
```

The shell expands the wild-card argument * to a list of files (*alpha*, *new*, and *psi*) and directories (*epsilon*, *gamma*). Because of this, directories appear on the argument list, and our *got revert* command should fail with an error:

```
got: reverting directories requires -R option
```

Let's modify a file in inside a directory as well. Run:

```
echo testing reverting of changes with zeta > epsilon/zeta
```

Run *got status* to verify that zeta is now shown as modified.

Next, we run *got revert* again, this time in recursive (*-R*) mode:

```
got revert -R *
```

The output of *got revert* should be:

```
R  epsilon/zeta
R  new
```

This looks fine at first glance, doesn't it?

Obtain the current status of files in the work tree with *got status*:

```
got status
```

```
D  beta
?  new
```

The newly added file has become an unversioned file again, just as it was before we ran *got add*.

However, we can now clearly see that the file *beta* has not been reverted. It is still scheduled for deletion. Since deleted files are not present on disk, the shell wild-card argument * did not add *beta* to our *got revert* command line.

Reverting all changes within the current working directory can be done more reliably by passing a path to the current working directory itself. Run:

```
got revert -R .
```

Here we are passing a dot (.) instead of an asterisk (*), and now beta will also be reverted since it appears in the directory listing of the current working directory.

The output of *got revert* should be:

```
R  beta
```

Obtain the current status of files in the work tree with *got status*:

```
got status
```

Since *got revert* will not touch unversioned files, the file *new* should still be present, and the output of *got status* will be:

```
?  new
```

10.2 Selectively undoing uncommitted changes

In the previous section we reverted entire files. But sometimes, there is a bad change sitting between several good changes in a file, and undoing just the one individual bad change (or more) would make us happy.

The *got revert* command has an interactive mode which allows us to select specific changes in files to undo. To try this feature we will need a file which contains more than one line of text.

Generate a file with more than one line of text, and commit it:

```
for i in `seq 20`; do
      echo this is line $i of a new file >> manylines
done
got add manylines
got commit
```

Let's modify a couple of lines in this file:

```
sed -i -e 's/line 2 /the second line /' -e 's/line 7/the seventh line/' \
    -e 's/line 16/the sixteenth line/' manylines
```

View the changes made to the *manylines* file by running:

```
got diff
```

The output should be similar to:

```
diff /home/user/tutorial/localrepo
path + /home/user/tutorial/localrepo
commit - bf373e66d43a75c230d8ecca48b91d5cad996661
blob - 35fe4858eb7b100a913a7fb0fcf9e1bd5dafad6b
file + manylines
--- manylines
+++ manylines
@@ -1,10 +1,10 @@
 this is line 1 of a new file
-this is line 2 of a new file
+this is the second line of a new file
 this is line 3 of a new file
 this is line 4 of a new file
 this is line 5 of a new file
 this is line 6 of a new file
-this is line 7 of a new file
+this is the seventh line of a new file
 this is line 8 of a new file
 this is line 9 of a new file
 this is line 10 of a new file
@@ -13,7 +13,7 @@ this is line 12 of a new file
 this is line 13 of a new file
 this is line 14 of a new file
 this is line 15 of a new file
-this is line 16 of a new file
+this is the sixteenth line of a new file
 this is line 17 of a new file
 this is line 18 of a new file
 this is line 19 of a new file
```

Let's assume we wanted to undo the change to line 7 only. Run:

```
got revert -p manylines
```

The output should be:

```
---------------------------------------------
@@ -1,5 +1,5 @@
 this is line 1 of a new file
-this is line 2 of a new file
+this is the second line of a new file
 this is line 3 of a new file
 this is line 4 of a new file
 this is line 5 of a new file
---------------------------------------------
M  manylines (change 1 of 3)
revert this change? [y/n/q]
```

Here, the *got revert* command displays only the first change on line 2, and then asks whether we would like to revert this change.

Respond by typing:

```
n
```

This will cause *got revert* to keep the change intact, and move on to the next change:

```
---------------------------------------------
@@ -4,7 +4,7 @@ this is line 3 of a new file
 this is line 4 of a new file
 this is line 5 of a new file
 this is line 6 of a new file
-this is line 7 of a new file
+this is the seventh line of a new file
 this is line 8 of a new file
 this is line 9 of a new file
 this is line 10 of a new file
---------------------------------------------
M  manylines (change 2 of 3)
revert this change? [y/n/q]
```

This time, respond with:

```
y
```

This change will be reverted.

Next, the third and final change should be displayed:

```
---------------------------------------------
@@ -13,7 +13,7 @@ this is line 12 of a new file
 this is line 13 of a new file
 this is line 14 of a new file
 this is line 15 of a new file
-this is line 16 of a new file
+this is the sixteenth line of a new file
 this is line 17 of a new file
 this is line 18 of a new file
 this is line 19 of a new file
---------------------------------------------
M  manylines (change 3 of 3)
revert this change? [y/n/q]
```

We have already reverted the one change we wanted to revert in this file, so we can stop the process for this file by typing:

```
q
```

This will keep the current change, and any potential later changes, intact.

Now examine the changes with *got diff* again:

```
got diff
```

The output should be similar to:

```
diff /home/user/tutorial/localrepo
path + /home/user/tutorial/localrepo
commit - bf373e66d43a75c230d8ecca48b91d5cad996661
blob - 35fe4858eb7b100a913a7fb0fcf9e1bd5dafad6b
file + manylines
--- manylines
+++ manylines
@@ -1,5 +1,5 @@
 this is line 1 of a new file
-this is line 2 of a new file
+this is the second line of a new file
 this is line 3 of a new file
 this is line 4 of a new file
 this is line 5 of a new file
@@ -13,7 +13,7 @@ this is line 12 of a new file
 this is line 13 of a new file
 this is line 14 of a new file
 this is line 15 of a new file
-this is line 16 of a new file
+this is the sixteenth line of a new file
 this is line 17 of a new file
 this is line 18 of a new file
 this is line 19 of a new file
```

Commit these changes to the *manylines* file by running:

```
got commit
```

10.3 Undoing already committed changes

Bad changes which have already been committed cannot easily be undone directly. Considering that such changes might have been synchronized to other repositories already, attempting to eradicate them completely in all known, or even unknown, clones of the repository is either a difficult or an impossible proposition.

But instead of undoing such changes, we can commit a change which undoes the bad effect of the change. Sometimes, a follow-up fix which gets propagated to all repositories is appropriate. But in some situations, making the same change in reverse will be a more desirable quick remedy.

The *got backout* command makes it easy to commit the inverse of a bad change. It runs in a work tree and takes either a commit object identifier or a reference name as argument. In both case, a single commit will be undone and presented as a committable change in the work tree. This approach is also known as a “reverse merge” of the bad change.

Let’s try undoing the *main* branch head commit in the work tree of the *localrepo.git* repository:

```
cd ~/tutorial/localrepo
got backout main
```

We just made a commit from this work tree in the previous section, and the work tree will likely be in a mixed-commit state as a result. In this, *got backout* will raise an error:

```
got: work tree contains files from multiple base commits; the entire work tree must be  ↩
    updated first
```

In order to provide a clean and accurate single-commit representation of the latest repository snapshot for *got backout* to work with, we need to run:

```
got update
```

This will likely not bring any actual changes into the work tree, but simply tweak base commit meta data for all files, and then print:

```
Already up-to-date
```

Still, this was worth doing, because if a work tree happened to be in a state where it was entirely out of sync with the latest repository snapshot, we would likely run into rather strange merge conflicts while trying to reverse-merge a change into this work tree.

With a fully up-to-date work tree, running *got backout* again should succeed:

```
got backout main
```

The output should be similar to:

```
G  manylines
Backed out commit b0064a134696f629f94453bdb61720ef2ef6b7f8
```

Examine the changes which have been merged into the work tree:

```
got diff
```

```
diff /home/user/tutorial/localrepo
path + /home/user/tutorial/localrepo
commit - b0064a134696f629f94453bdb61720ef2ef6b7f8
blob - 9eff53599c682cff07f30fd3377116fa55a472e8
file + manylines
--- manylines
+++ manylines
@@ -1,5 +1,5 @@
 this is line 1 of a new file
-this is the second line of a new file
+this is line 2 of a new file
 this is line 3 of a new file
 this is line 4 of a new file
 this is line 5 of a new file
@@ -13,7 +13,7 @@ this is line 12 of a new file
 this is line 13 of a new file
 this is line 14 of a new file
 this is line 15 of a new file
-this is the sixteenth line of a new file
+this is line 16 of a new file
 this is line 17 of a new file
 this is line 18 of a new file
 this is line 19 of a new file
```

These changes undo the effects of our latest change which was committed to the *main* branch in the previous section.

Commit the result by running:

```
got commit
```

When the log message template opens in the editor, the log message of the backed-out commit should conveniently be present already, ready for modification or quoting when describing the reasons for and context of undoing this change. Save and exit the editor, allowing the commit to proceed.

The output of *got commit* should be similar to:

```
M  manylines
Created commit 556314028545e6294c928040ef17455056cfe865
```

Note
While we only backed out a single change in this example, it is possible to back out multiple changes in sequence before running *got commit*. When doing so, backing out all relevant commits in reverse order is a good idea since this will avoid unnecessary merge conflicts. The log messages of all backed out commits will be appended to the log message buffer when *got commit* runs, provided the changes in relevant files have not been reverted with *got revert* beforehand.

Chapter 11

Committing changes selectively

So far, throughout this tutorial, we have been committing entire files with all changes they contain at once. This is the default behaviour of Game of Trees when creating new commits.

Git users might instead be used to the “staging area” of the Git work tree. Every change committed with Git has to pass through this staging area.

Staging is supported by Game of Trees as well, but use of the staging feature is not required. It is possible to stage files in their entirety, and to selectively stage changes found within files.

11.1 Staging entire files

When several files were changed in the work tree, we might want to commit some of those files separately from other files.

Modify two files and add one new file in your work tree of *localrepo.git*:

```
cd ~/tutorial/localrepo
echo testing commits with file alpha > alpha
echo testing commits with file psi > psi
echo this is a new file > new
got add new
got status
```

The above *got status* command should print:

```
M  alpha
A  new
M  psi
```

Commit just the changes to file psi by running:

```
got commit psi
```

The output should be similar to:

```
M  psi
Created commit dd534e4160fac70063bc01126b4af1b87388fabd
```

Only *psi* was committed, and the other files should still be in modified and added state as before. To confirm this, run:

```
got status
```

The above *got status* command should print:

```
M  alpha
A  new
```

This shows that we can ask Game of Trees to commit changes to specific files by passing only the desired files to *got commit* on the command line. However, we can also stage entire files while leaving other files unstaged, preparing a list of files for *got commit* to pick up.

Stage the modifications in file *alpha* by running:

```
got stage alpha
```

The output of *got stage* should be:

```
 M alpha
```

Only *alpha* was staged, and the file *psi* files should still be in added state as before. To confirm this, run:

```
got status
```

When a file is staged, its status code is shown in to the second column rather than first column. The above *got status* command should print:

```
 M alpha
A  new
```

Staged file content becomes the new base for local changes in the work tree.

View the local modifications in file alpha by running:

```
got diff alpha
```

The output of the above *got diff* command should be empty. To view staged changes, we can pass the *-s* option to *got diff*. Try:

```
got diff -s
```

This shows all staged changes, and the output should display the staged changes for file *alpha*, similar to:

```
diff -s /home/user/tutorial/localrepo
path + /home/user/tutorial/localrepo (staged changes)
commit - 556314028545e6294c928040ef17455056cfe865
blob - 7c942f4e44e84971edf730376487105606d0c7ae
blob + 43c4172a101a91887bf54f29ab0fdec310c09c7a
--- alpha
+++ alpha
@@ -1,2 +1 @@
-change file alpha again
-add a line to file alpha
+testing commits with file alpha
```

Modify the file *alpha* again by running:

```
echo modifying alpha which has been staged > alpha
```

View the local modifications in file alpha by running:

```
got diff alpha
```

This time we should see changes in the work tree, relative to the staged version of file *alpha*, similar to:

```
diff /home/user/tutorial/localrepo
path + /home/user/tutorial/localrepo
commit - 556314028545e6294c928040ef17455056cfe865
blob - 43c4172a101a91887bf54f29ab0fdec310c09c7a (staged)
file + alpha
--- alpha
+++ alpha
@@ -1 +1 @@
-testing commits with file alpha
+modifying alpha which has been staged
```

Commit the staged changes by running:

```
got commit
```

Your log message editor buffer should be preloaded with comments that display staged files only:

```
# changes to be committed on branch main:
#  M  /alpha
# detailed changes can be viewed in /tmp/got-AN2RybZ7ef.diff
```

The output of *got commit* should be similar to:

```
M  alpha
Created commit 3bb0cb474042f0b52f83cc1663a6fa918d5c58d3
```

The file *new* which was not staged was not made part of this commit and should still appear as locally added in the work tree. To confirm this, run:

```
got status
```

The output should be:

```
M  alpha
A  new
```

Commit any remaining changes to alpha by running:

```
got commit alpha
```

Revert the addition of file *new* for now, by running:

```
got revert new
```

11.2 Staging individual changes in files

When several changes appear in a single file, there might be more than one semantic change contained within the file. For example, a subset of the changes might be about fixing a bug, while another change fixes an unrelated typo in a comment.

By staging specific files or changes within files for commit with the *got stage -p* command, we can prepare a specific set of changes which *got commit* will pick up, while leaving all other file modifications in the work tree.

Modify two lines in the *manylines* file created earlier:

```
sed -i -e 's/line 3/the third line/' \
    -e 's/line 10/the tenth line/' \
    manylines
```

Examine the local changes in the work tree. Run:

```
got diff
```

The output of *got diff* should be similar to:

```
diff /home/user/tutorial/localrepo
path + /home/user/tutorial/localrepo
commit - 998ea5798df257c0abfeea3d38855f48560d4de2
blob - 9eff53599c682cff07f30fd3377116fa55a472e8
file + manylines
--- manylines
+++ manylines
@@ -1,13 +1,13 @@
 this is line 1 of a new file
 this is the second line of a new file
-this is line 3 of a new file
+this is the third line of a new file
 this is line 4 of a new file
 this is line 5 of a new file
 this is line 6 of a new file
 this is line 7 of a new file
 this is line 8 of a new file
 this is line 9 of a new file
-this is line 10 of a new file
+this is the tenth line of a new file
 this is line 11 of a new file
 this is line 12 of a new file
 this is line 13 of a new file
```

Similar to *got revert*, the *got stage* command supports a *-p* (patch) option which provides an interactive menu to stage parts of a patch. Run:

```
got stage -p
```

The first section of output should be:

```
 ----------------------------------------------
 @@ -1,6 +1,6 @@
  this is line 1 of a new file
  this is the second line of a new file
 -this is line 3 of a new file
 +this is the third line of a new file
  this is line 4 of a new file
  this is line 5 of a new file
  this is line 6 of a new file
 ----------------------------------------------
 M  manylines (change 1 of 2)
 stage this change? [y/n/q]
```

Avoid staging this changes by responding with:

```
n
```

The next section of output should be:

```
----------------------------------------------
@@ -7,7 +7,7 @@ this is line 6 of a new file
 this is line 7 of a new file
 this is line 8 of a new file
 this is line 9 of a new file
-this is line 10 of a new file
+this is the tenth line of a new file
 this is line 11 of a new file
 this is line 12 of a new file
 this is line 13 of a new file
----------------------------------------------
M  manylines (change 2 of 2)
stage this change? [y/n/q]
```

Do stage this changes by responding with:

```
y
```

The *got status* command should now report *M* (modified) status in both columns for file *manylines*. Run:

```
got status
```

The output should be similar to;

```
MM manylines
?  new
```

The double *M* indicates that we have both staged and unstaged modifications in file *manylines*.

Now examine staged changes with:

```
got diff -s
```

The output should be similar to:

```
diff -s /home/user/tutorial/localrepo
path + /home/user/tutorial/localrepo (staged changes)
commit - 998ea5798df257c0abfeea3d38855f48560d4de2
blob - 9eff53599c682cff07f30fd3377116fa55a472e8
blob + 8c94cbd62b0508c8013b25ea15b960d895755158
--- manylines
+++ manylines
@@ -7,7 +7,7 @@ this is line 6 of a new file
 this is line 7 of a new file
 this is line 8 of a new file
 this is line 9 of a new file
-this is line 10 of a new file
+this is the tenth line of a new file
 this is line 11 of a new file
 this is line 12 of a new file
 this is line 13 of a new file
```

Examine the local changes, which are now shown relative to the staged changes, with:

```
got diff
```

The output should be similar to:

```
diff /home/user/tutorial/localrepo
path + /home/user/tutorial/localrepo
commit - 998ea5798df257c0abfeea3d38855f48560d4de2
blob - 8c94cbd62b0508c8013b25ea15b960d895755158 (staged)
file + manylines
--- manylines
+++ manylines
@@ -1,6 +1,6 @@
 this is line 1 of a new file
 this is the second line of a new file
-this is line 3 of a new file
+this is the third line of a new file
 this is line 4 of a new file
 this is line 5 of a new file
 this is line 6 of a new file
```

Commit the staged changes by running:

```
got commit
```

Keep the local modification in *manylines* intact for the next section.

11.3 Unstaging staged changes

Staged changes can be merged back into the work tree with *got unstage*. This is a 3-way merge operation, which means that unstaging changes may conflict if local modifications exist on top of staged changes. If the staged file was not modified after staging then the merge result is virtually guaranteed to be free of conflicts.

The file *manylines* should already contain local modifications:

```
got status manylines
```

The output should be:

```
M  manylines
```

Stage changes for the file *manylines* by running:

```
got stage manylines
```

The output should be:

```
M  manylines
```

Add another local change on top of the staged changes:

```
sed -i -e 's/line 4/the fourth line/' manylines
```

Verify the local changes by running:

```
got diff
```

The output should be similar to:

```
diff /home/user/tutorial/localrepo
path + /home/user/tutorial/localrepo
commit - bc0a7019f2fe21ef588a67f5183ca6cf6cea2e35
blob - 9c8c5ced84c7ef0e53e7074145c46eea2304a20f (staged)
file + manylines
--- manylines
+++ manylines
@@ -1,7 +1,7 @@
 this is line 1 of a new file
 this is the second line of a new file
 this is the third line of a new file
-this is line 4 of a new file
+this is the fourth line of a new file
 this is line 5 of a new file
 this is line 6 of a new file
 this is line 7 of a new file
```

The file *manylines* now has both staged and local modifications:

```
got status manylines
```

The output should be:

```
MM manylines
```

Unstage the staged changes by running:

```
got unstage
```

You will likely get a merge conflict, in which case the output looks like this:

```
C  manylines
Files with new merge conflicts: 1
```

The *manylines* file is now in conflicted status:

```
got status manylines
```

The output should be:

```
C  manylines
```

If you did not get a merge conflict, don't worry about it. You can skip the conflict resolution steps below.

View the differences stored in the *manylines* file:

```
got diff manylines
```

The output should look similar to:

```
diff /home/user/tutorial/localrepo
path + /home/user/tutorial/localrepo
commit - bc0a7019f2fe21ef588a67f5183ca6cf6cea2e35
blob - 8c94cbd62b0508c8013b25ea15b960d895755158
file + manylines
--- manylines
+++ manylines
@@ -1,7 +1,14 @@
```

```
 this is line 1 of a new file
 this is the second line of a new file
-this is line 3 of a new file
+<<<<<<< merged change: commit bc0a7019f2fe21ef588a67f5183ca6cf6cea2e35
+this is the third line of a new file
 this is line 4 of a new file
+||||||| 3-way merge base: commit bc0a7019f2fe21ef588a67f5183ca6cf6cea2e35
+this is line 3 of a new file
+=======
+this is the third line of a new file
+this is the fourth line of a new file
+>>>>>>>
 this is line 5 of a new file
 this is line 6 of a new file
 this is line 7 of a new file
```

Edit the file and resolve the conflict in whichever way you prefer.

For example, we could accept both changes to resolve the conflict, and run:

```
got diff
```

The output would then be similar to the following:

```
diff /home/user/tutorial/localrepo
path + /home/user/tutorial/localrepo
commit - bc0a7019f2fe21ef588a67f5183ca6cf6cea2e35
blob - 8c94cbd62b0508c8013b25ea15b960d895755158
file + manylines
--- manylines
+++ manylines
@@ -1,7 +1,7 @@
 this is line 1 of a new file
 this is the second line of a new file
-this is line 3 of a new file
-this is line 4 of a new file
+this is the third line of a new file
+this is the fourth line of a new file
 this is line 5 of a new file
 this is line 6 of a new file
 this is line 7 of a new file
```

Commit the changes by running:

```
got commit
```

Note
Like the *got stage* command, the *got unstage* command accepts the *-p* option for unstaging patch chunks individually.

Chapter 12

Working with branches

Proficient users of version control systems tend to pay attention to the quality of the commit history of a project. This means they will spend effort writing log messages which clearly communicate intent, and will aim for splitting changes up in separately digestible chunks, allowing themselves and others to follow the evolution of the project over time.

Keeping each semantic change to a project separate from other semantic changes is not always easy. All version control systems provide ways of separating unrelated changes, and even though there will not always be a 100% satisfactory answer to any given situation or workflow, understanding how version control systems attempt to achieve separation of changes is key to becoming an advanced user of these systems, beyond just treating version control as a means of storing backups of older versions.

We have already seen one way to make separate changes with Game of Trees, namely the use of multiple work trees. Managing separate changes in separate work trees would be a viable and simple approach, but has downsides.

One is that we cannot send changes to another machine before the changes are committed, be it to share the changes with someone else, or to have a backup on a remote machine. When our work-in-progress changes depend on the survival and availability of the storage medium on which they were initially stored then it is not difficult to imagine how our work could be lost or destroyed.

Another downside is that the commit history we can produce from changes sitting in a work tree is set in stone. We can commit separate files and even stage individual changes to files while recording changes as incremental steps. But the order of these steps cannot be adjusted after the fact, and any mistakes we make will require follow-up changes to be corrected. These extra commits become a permanent part of the project's history. While they are really not a big deal, it would be nice to keep such follow-up corrections at a minimum to ensure that readers of the history do not have to connect too many distracting dots to learn what has changed and why.

So while there is no restriction on how many work trees we may use to our benefit, there is another tool at our disposal in order to keep changes separate, and making use of this in addition to work trees has additional benefits.

So far, we have been committing all our changes to the *main* branch of the repository. As a reminder, the reference *refs/head/main* always refers to the identifier of a specific commit, a commit which is considered the current head of the *main* branch. The name *main* is just a convention, and it might be named differently elsewhere. It refers to the version of project history which is considered official and canonical, and which everyone is expected to base further changes on top of.

By using more than one such branch, i.e. more than just one reference inside the *refs/heads/* reference namespace, we can achieve all of the following:

- Keep changes scoped to a specific task or problem we are working on, while making commits to the repository.
- Keep backups of our changes on another machine reachable via a network connection, without impacting the *main* history of the project.
- Collaborate with a sub-group of project developers on a very specific problem, across a local network or the internet.
- Temporary stash changes when suddenly having to switch context to another task, and get these changes back from our stash at a later time.
- Maintain a version of our project which only receives selected changes, for example changes which address specific bugs only.

- Committing changes in a “messy” way without ever having to worry about impacting the “clean” *main* history of the project.
- Creating a revised and “clean” versions of the history of a “messy” branch, before integrating the changes into the *main* branch.

12.1 Listing branch references

Game of Trees provides several commands which manage references in the Git repository. The most general of these if *got ref*, which operates on reference in a Git repository and can provide a full view of all references. The *got ref* command will operate on references without restrictions but also without any safe-guards.

Other commands operate on references in the context of specific operations and will enforce conventions which make sense in a specific context. These commands usually restrict themselves to one or more reference namespaces. For example, the *got branch* command uses references in the *refs/heads* and *refs/remotes/* namespace only.

In its simplest form, we can use the *got branch* command to list the current branch checked out in a work tree. Try it in a work tree of the *localrepo.git* repository;

```
cd ~/tutorial/localrepo
got branch
```

If you have followed the tutorial, the output should be:

```
main
```

The command can also be used to list all known branch heads, local or remote:

```
got branch -l
```

The output should be similar to:

```
* main: 0b4ccf44d2e710939c5eb54db13b9e6a685aa613
  origin/HEAD: refs/remotes/origin/main
  origin/main: 67fe73fbe97fe683ab162b05b1a6550209746569
```

Above, *main* refers to the local branch *refs/heads/main*, which is marked with an asterisk * because *main* is the current branch of our work tree, and this work tree is fully up-to-date. In mixed-commit work trees, a tilde ~ will be displayed next to the current branch instead.

The *origin/main* branch refers to *refs/remotes/origin/main*, as cloned from the default remote “origin” repository *remoterepo.git*. We also see the remote’s HEAD value being cached, which is useful in case the default branch name is changed remotely.

The output of *got branch -l* is a simplified view of the output shown by the equivalent *got ref* command. Try:

```
got ref -l
```

The output should be similar to:

```
HEAD: refs/heads/main
refs/got/worktree/base-ce324e95-729d-45f4-aa00-c8cf7100bf2e: 0 ↩
    b4ccf44d2e710939c5eb54db13b9e6a685aa613
refs/heads/main: 0b4ccf44d2e710939c5eb54db13b9e6a685aa613
refs/remotes/origin/HEAD: refs/remotes/origin/main
refs/remotes/origin/main: 67fe73fbe97fe683ab162b05b1a6550209746569
```

12.2 Creating a new branch

A new branch can be created by passing the desired name to the *got branch* command. Try:

```
got branch main
```

The output should be:

```
got: specified branch already exists
```

The name we pass is looked up in the *refs/heads/* namespace. We cannot accidentally change or overwrite an existing branch while using *got branch*. Whereas the more low-level *got ref* command could accidentally overwrite an existing reference if we are not careful.

Create a new branch called *counting*:

```
got branch counting
```

The output should be:

```
Switching work tree from refs/heads/main to refs/heads/counting
```

By default, new branches are based on the head commit of the current branch checked out in the work tree. If *got branch* does not run in a work tree, the new branch will be based on the head commit of the branch resolved via the repository's HEAD reference.

The new branch should now appear in the branch listing:

```
got branch -l
```

```
* counting: 0b4ccf44d2e710939c5eb54db13b9e6a685aa613
  main: 0b4ccf44d2e710939c5eb54db13b9e6a685aa613
  origin/HEAD: refs/remotes/origin/main
  origin/main: 67fe73fbe97fe683ab162b05b1a6550209746569
```

Currently, both branch heads are still equivalent. Let's commit some changes to our new branch. For example, we could create 3 new files as follows:

```
echo 1 > one
echo 2 > two
echo 3 > three
got add one two three
got commit
```

The output of the above *got commit* command should be similar to:

```
A  one
A  three
A  two
Created commit 1261b1cc8dfd522cbc52fce8cbbaae9a2cd57fa2
```

The head of the *counting* branch has moved forward, but *main* is unchanged. We can change *main* independently. While we could check out one dedicated work tree per branch, for now let's switch Our current work back to the *main* branch ;

```
got update -b main
```

The output of the above *got update* command should be:

```
Switching work tree from refs/heads/counting to refs/heads/main
D  one
D  three
D  two
Updated to refs/heads/main: 0b4ccf44d2e710939c5eb54db13b9e6a685aa613
```

The new files added to the *counting* branch were removed again. Change a file on the *main* branch:

```
echo testing branches with file psi > psi
got commit
```

Now both branches have diverged. We can inspect their independent histories with *got log* We limit it to traversing just 2 commits via the *-l* option:

```
got log -l 2
```

This should show the head commit of the work tree's current branch, *main*, followed by its parent commit. The output should be similar to:

```
------------------------------------------------
commit 254f3b1d4f6579ab4399306cd99ed89053062f2d (main)
from: Tom Smith <tom.smith@example.com>
date: Mon Sep  7 20:13:55 2026 UTC

 changing psi

------------------------------------------------
commit 0b4ccf44d2e710939c5eb54db13b9e6a685aa613
from: Tom Smith <tom.smith@example.com>
date: Sun Sep  6 12:58:33 2026 UTC

 changing lines 3 and 4
```

Next, ask *got log* to start traversing history at the commit which can be resolved via the *counting* branch reference:

```
got log -l 2 -c counting
```

This time we should see the branch head commit of the *counting* branch, based on the same parent commit as *main*, similar to:

```
------------------------------------------------
commit 1261b1cc8dfd522cbc52fce8cbbaae9a2cd57fa2 (counting)
from: Tom Smith <tom.smith@example.com>
date: Mon Sep  7 20:08:44 2026 UTC

 add some numbers

------------------------------------------------
commit 0b4ccf44d2e710939c5eb54db13b9e6a685aa613
from: Tom Smith <tom.smith@example.com>
date: Sun Sep  6 12:58:33 2026 UTC

 changing lines 3 and 4
```

Both changes are sitting in parallel to one another. This situation is quite similar to an earlier situation we saw in this tutorial, where a remote repository contained another change we had just fetched into the local repository but had not merged yet. Only, this time, both references involved are in the same reference namespace, namely *refs/heads/*.

We could use *got merge* again to reconcile the branches as we did earlier, but let's try out the rebase strategy instead this time.

12.3 Rebasing a branch

Rebasing the *counting* branch on top of *main* requires a work tree of the *main* branch, which we already have. The work tree must be fully up-to-date and contain no local modifications, run:

```
got update
got status
```

The output of the above *got status* command should be empty.

When rebasing, the command which brings changes from a rebased branch back into the branch it forked off from is called *got integrate* rather than *got merge*. Running this command at this point should raise an error:

```
got integrate counting
```

There should be an error indicating that *counting* needs to be rebased first:

```
got: specified branch must be rebased first
```

Let's try to rebase the *counting* branch now:

```
got rebase counting
```

The output should be similar to:

```
A  one
A  three
A  two
1261b1cc8dfd -> af1f25f1032b: add some numbers
Switching work tree to refs/heads/counting
```

The changes in the rebased commit did not overlap with changes we had sitting on the main branch. Thus, there was no merge conflict. If there were merge conflicts, we would have to resolve them and continue the rebase operation with *got rebase -c*, in the same was as was done earlier with *got merge -c*.

Running *got log* at this point should reveal that the branch head of *counting* now has a different parent commit, namely the head commit of *main*:

```
got log -l 2
```

The output should be similar to:

```
-----------------------------------------------
commit af1f25f1032b5bf79df4f0d22ceb887e0e4adba6 (counting)
from: Tom Smith <tom.smith@example.com>
date: Mon Sep  7 20:32:51 2026 UTC

 add some numbers

-----------------------------------------------
commit 254f3b1d4f6579ab4399306cd99ed89053062f2d (main)
from: Tom Smith <tom.smith@example.com>
date: Mon Sep  7 20:13:55 2026 UTC

 changing psi
```

Add some more numbers to the *counting* branch:

```
echo 4 > four
echo 5 > give
echo 6 > six
got add four five six
got commit
```

Examine the history of *counting*, walking up to 3 commits this time:

```
got log -l 3
```

The output should be similar to:

```
------------------------------------------------
commit a3682741cef4807ce0e85e561ad81812e9abe254 (counting)
from: Tom Smith <tom.smith@example.com>
date: Mon Sep  7 20:36:41 2026 UTC

 adding more numbers

------------------------------------------------
commit af1f25f1032b5bf79df4f0d22ceb887e0e4adba6
from: Tom Smith <tom.smith@example.com>
date: Mon Sep  7 20:32:51 2026 UTC

 add some numbers

------------------------------------------------
commit 254f3b1d4f6579ab4399306cd99ed89053062f2d (main)
from: Tom Smith <tom.smith@example.com>
date: Mon Sep  7 20:13:55 2026 UTC

 changing psi
```

As long as the changes on the *counting* branch are sitting on top of the head of the *main* branch, we can integrate our changes into the *main* branch.

First, switch the work tree back to the main branch;

```
got update -b main
```

The output should be similar to;

```
Switching work tree from refs/heads/counting to refs/heads/main
D  five
D  four
D  one
D  six
D  three
D  two
Updated to refs/heads/main: 254f3b1d4f6579ab4399306cd99ed89053062f2d
```

Integrate the *counting* branch by running:

```
got integrate counting
```

The output should be:

```
A  five
A  four
A  one
A  six
A  three
A  two
Integrated refs/heads/counting into refs/heads/main
```

View the history of the most recent three commits again:

```
got log -l 3
```

The output should be similar to the following. The branch heads are now equivalent again:

```
 ------------------------------------------------
 commit a3682741cef4807ce0e85e561ad81812e9abe254 (counting, main)
 from: Tom Smith <tom.smith@example.com>
 date: Mon Sep  7 20:36:41 2026 UTC

  adding more numbers

 ------------------------------------------------
 commit af1f25f1032b5bf79df4f0d22ceb887e0e4adba6
 from: Tom Smith <tom.smith@example.com>
 date: Mon Sep  7 20:32:51 2026 UTC

  add some numbers

 ------------------------------------------------
 commit 254f3b1d4f6579ab4399306cd99ed89053062f2d
 from: Tom Smith <tom.smith@example.com>
 date: Mon Sep  7 20:13:55 2026 UTC

  changing psi
```

This cycle can be repeated over and over, and with as many branches as we would like, to develop specific changes in isolation.

For now, we can delete the *counting* branch again. Run:

```
got branch -d counting
```

The output should be similar to:

```
Deleted refs/heads/counting: a3682741cef4807ce0e85e561ad81812e9abe254
```

Note
An important thing to keep in mind is that commit identifiers of rebased changes will change. It is rarely appropriate to rebase the *main* branch of a project, especially when this branch is synchronized to different machines. Only rebase branches which contain work-in-progress changes being prepared for eventual integration.

12.4 Backups of rebased branches

Game of Trees keeps backups of all rebased branches, allowing us to view the branch as it appeared before rebasing, and potentially restore the branch to an older state.

To see a list of backed-up rebased branch, run:

```
got rebase -l
```

The output should be similar to:

```
---------------------------------------------
commit 1261b1cc8dfd522cbc52fce8cbbaae9a2cd57fa2 (formerly counting)
from: Tom Smith <tom.smith@example.com>
date: Mon Sep  7 20:08:44 2026 UTC

 add some numbers

has become commit af1f25f1032b5bf79df4f0d22ceb887e0e4adba6
 2026-09-07 tom.smith  add some numbers
history forked at 0b4ccf44d2e710939c5eb54db13b9e6a685aa613
 2026-09-06 tom.smith  changing lines 3 and 4
```

This gives us all information needed to view this change as it appeared before being rebased. Given the above commit identifiers, a suitable command to see the changes would be:

```
got diff -c 1261b1cc
```

We could even recreate the *counting* branch in its earlier state if we wanted to, by running:

```
got branch -c 1261b1cc counting
got log -l 2
```

The above *got log* command should display the *counting* branch back in its original state:

```
---------------------------------------------
commit 1261b1cc8dfd522cbc52fce8cbbaae9a2cd57fa2 (counting)
from: Tom Smith <tom.smith@example.com>
date: Mon Sep  7 20:08:44 2026 UTC

 add some numbers

---------------------------------------------
commit 0b4ccf44d2e710939c5eb54db13b9e6a685aa613
from: Tom Smith <tom.smith@example.com>
date: Sun Sep  6 12:58:33 2026 UTC

 changing lines 3 and 4
```

This can be useful when a rebase didn't work out as expected, or was performed in error.

When backups of rebased branches are no longer required, they can be removed by running:

```
got rebase -X
```

This will remove references in the *refs/got/backup/rebase/* namespace, where the old heads of rebased branches are preserved.

12.5 Editing branch history

Editing branch history can be useful in order to transform a "messy" commit history in a revised and more appealing commit history. This is usually done only for changes which are still in draft state and have not yet been merged into the *main* branch. More generally, if anyone is already building other changes on top of a published or privately synchronized part commit of history, editing this part of history is usually unwise.

In our running example, the *localrepo.git* repository was last synchronized with the "origin" repository *remoterepo.git* several commits ago, and all changes we have added on top is only visible in our own repository. We can thus freely edit this history before sending our contributions to the *remoterepo.git* repository.

In a work tree which contains the *main* branch of *localrepo.git*, Examine the commit history of *localrepo.git*, up to and including the cached *remotes/origin/main* remote branch head. We can enable the summary mode of *got log* via the *-s* option to shorten the output a bit. Run:

```
got log -s -x origin/main
```

The output should be similar to:

```
2026-09-07 main    adding more numbers
2026-09-07 af1f25f add some numbers
2026-09-07 254f3b1 changing psi
2026-09-06 0b4ccf4 changing lines 3 and 4
2026-09-05 bc0a701 modify manylines file via a staged change
2026-09-05 998ea57 changes to manylines file
2026-09-05 3bb0cb4 modifying file alpha via staged changes
2026-09-05 dd534e4 changing psi
2026-09-04 5563140 backout previous: tweak 2 lines
2026-09-04 b0064a1 tweak 2 lines
2026-09-04 bf373e6 add a file with many lines
2026-09-03 65ad026 adding a line to file alpha
2026-09-03 7c14249 remove ignore-me file again
2026-09-03 926dc41 add ignored file
2026-09-03 70732df add ignore list
2026-09-03 e7d64e8 add psi
2026-09-03 67fe73f merge refs/remotes/origin/main into refs/heads/main
```

In this example, there are 16 commits which could in theory all be edited before sending them out.

For example, let's try to edit the file *psi* across our local history to make it much prettier. First, we need to ensure that our work tree is clean. Run:

```
got status
```

If the output is not empty, commit or revert outstanding changes.

Starting with a clean work tree, we keep our work on the *main* branch but back-date our work tree's base commit to the commit which is the parent of the bottom-most commit being edited. In our case, we can run:

```
got update -c origin/main
```

Note
It is important to use the *-c* (commit) option here, rather than the *-b* (branch) option which would switch our work tree's current branch to *origin/main*. Our work tree must remain on the *main* branch which we wish to edit.

The output should be similar to:

```
D  .gitignore
U  alpha
D  five
D  four
D  manylines
D  one
D  psi
D  six
D  three
D  two
Updated to refs/heads/main: 67fe73fbe97fe683ab162b05b1a6550209746569
```

Our work tree is now based on the commit which equals the branch head of *origin/main*, and we can edit commits between *origin/main* up to, and including, *main* by running:

```
got histedit
```

The *got histedit* command opens an editor window which displays a "histedit script", showing the part commit history we can edit in chronological order (unlike *got log* which, displays commits in reverse chronological order). We can edit the histedit script in our editor to let Game of Trees know how history should be changed.

The editor window should look similar to the following:

```
# Editing the history of branch 'main' starting at
# commit e7d64e8b65608cf6e3577527bbc967651d0641f4
# Commits will be processed in order from top to bottom of this file.
# Available histedit commands:
#   pick (p): use commit
#   edit (e): use commit but stop for amending
#   fold (f): combine with next commit that will be used
#   drop (d): remove commit from history
#   mesg (m): open editor to edit the log message
pick e7d64e8b65608cf6e3577527bbc967651d0641f4 add psi
pick 70732df8a3d93309918723814e27ae01194e06e2 add ignore list
pick 926dc418635f0541c53d820b954a7ed063072946 add ignored file
pick 7c1424929a9eaddcee8745ebc827651a766c450e remove ignore-me file again
pick 65ad026b2001e9d26a0e82fec8d4709bc5da826e adding a line to file alpha
pick bf373e66d43a75c230d8ecca48b91d5cad996661 add a file with many lines
pick b0064a134696f629f94453bdb61720ef2ef6b7f8 tweak 2 lines
pick 556314028545e6294c928040ef17455056cfe865 backout previous: tweak 2 lines
pick dd534e4160fac70063bc01126b4af1b87388fabd changing psi
pick 3bb0cb474042f0b52f83cc1663a6fa918d5c58d3 modifying file alpha via staged ch
pick 998ea5798df257c0abfeea3d38855f48560d4de2 changes to manylines file
pick bc0a7019f2fe21ef588a67f5183ca6cf6cea2e35 modify manylines file via a staged
pick 0b4ccf44d2e710939c5eb54db13b9e6a685aa613 changing lines 3 and 4
pick 254f3b1d4f6579ab4399306cd99ed89053062f2d changing psi
pick af1f25f1032b5bf79df4f0d22ceb887e0e4adba6 add some numbers
pick a3682741cef4807ce0e85e561ad81812e9abe254 adding more numbers
```

By default, the histedit script picks all commits as-is, meaning it creates the same history as the original, except that timestamps and perhaps the author meta-data stored in commits can be adjusted (e.g. if the *GOT_AUTHOR* environment variable was changed).

Let's do something more interesting: Locate all changes which modify the file *psi*, and mark them for editing by replacing *pick* with *edit* (or just *e*) at the beginning of the line.

If you are unsure which commits edited the file *psi*, run the following in another terminal window:

```
cd ~/tutorial/localrepo.git
got log -s psi
```

And the output of *got log* should list all relevant commits:

```
2026-09-07 254f3b1 changing psi
2026-09-05 dd534e4 changing psi
2026-09-03 e7d64e8 add psi
```

In the example below, the corresponding 3 commits are marked for editing:

```
# Editing the history of branch 'main' starting at
# commit e7d64e8b65608cf6e3577527bbc967651d0641f4
# Commits will be processed in order from top to bottom of this file.
# Available histedit commands:
```

```
#   pick (p): use commit
#   edit (e): use commit but stop for amending
#   fold (f): combine with next commit that will be used
#   drop (d): remove commit from history
#   mesg (m): open editor to edit the log message
e e7d64e8b65608cf6e3577527bbc967651d0641f4 add psi
pick 70732df8a3d93309918723814e27ae01194e06e2 add ignore list
pick 926dc418635f0541c53d820b954a7ed063072946 add ignored file
pick 7c1424929a9eaddcee8745ebc827651a766c450e remove ignore-me file again
pick 65ad026b2001e9d26a0e82fec8d4709bc5da826e adding a line to file alpha
pick bf373e66d43a75c230d8ecca48b91d5cad996661 add a file with many lines
pick b0064a134696f629f94453bdb61720ef2ef6b7f8 tweak 2 lines
pick 556314028545e6294c928040ef17455056cfe865 backout previous: tweak 2 lines
e dd534e4160fac70063bc01126b4af1b87388fabd changing psi
pick 3bb0cb474042f0b52f83cc1663a6fa918d5c58d3 modifying file alpha via staged ch
pick 998ea5798df257c0abfeea3d38855f48560d4de2 changes to manylines file
pick bc0a7019f2fe21ef588a67f5183ca6cf6cea2e35 modify manylines file via a staged
pick 0b4ccf44d2e710939c5eb54db13b9e6a685aa613 changing lines 3 and 4
e 254f3b1d4f6579ab4399306cd99ed89053062f2d changing psi
pick af1f25f1032b5bf79df4f0d22ceb887e0e4adba6 add some numbers
pick a3682741cef4807ce0e85e561ad81812e9abe254 adding more numbers
```

Save the script, and exit the editor. The *got histedit* command will stop at the first commit marked or editing and allow you to make arbitrary edits in the work tree:

```
A  psi
Stopping histedit for amending commit e7d64e8b65608cf6e3577527bbc967651d0641f4
```

The work tree now contains an interrupted *histedit* operation, which allows for arbitrary changes to be made part of the commit being edited, including the addition and removal of files, and so on.

Run:

```
got status
```

to see the status of files in the work tree. The output should be:

```
A  psi
Work tree is editing the history of refs/heads/main
```

The change which has been merged into the work tree matches the original change which introduced file *psi*. For example, when running

```
got diff
```

the output should be similar to:

```
diff /home/user/tutorial/localrepo
path + /home/user/tutorial/localrepo
commit - 67fe73fbe97fe683ab162b05b1a6550209746569
blob - /dev/null
file + psi (mode 644)
--- /dev/null
+++ psi
@@ -0,0 +1 @@
+this is a new file called psi
```

We can finally begin our quest to make *psi* a prettier file:

```
echo 'this is a pretty file called psi' > psi
```

Examine the changes once more with:

```
got diff
```

If you are happy with the result, ask *got histedit* to continue editing history:

```
got histedit -c
```

A text editor will open, allowing you to adjust the log message of the modified commit as needed. Edit the log message, save, and exit the text editor.

This time, *got histedit* should produce output similar to:

```
e7d64e8b6560 -> 16de732ca6ef: add pretty psi
A  .gitignore
70732df8a3d9 -> 59183b7640a3: add ignore list
A  ignore-me
926dc418635f -> 680a875d6007: add ignored file
D  ignore-me
7c1424929a9e -> 23205c28d34c: remove ignore-me file again
G  alpha
65ad026b2001 -> b9e07e3d605f: adding a line to file alpha
A  manylines
bf373e66d43a -> b1210ac4f08a: add a file with many lines
G  manylines
b0064a134696 -> 729aebec2489: tweak 2 lines
G  manylines
556314028545 -> 6b1e5727da8b: backout previous: tweak 2 lines
C  psi
Files with new merge conflicts: 1
dd534e4160fa -> merge conflict: changing psi
got: conflicts must be resolved before histedit can continue
```

This time, *got histedit* stops because of a merge conflict in file *psi*. It would have stopped anyway because we marked all commits which change *psi* for editing. The merge conflict is a result of changing the contents of *psi* in ways that conflict when the diff3 algorithm tries to merge changes from later commits into this file.

Open the file *psi* in a text editor, which should look something like this:

```
<<<<<<<
this is a pretty file called psi
||||||| 3-way merge base: commit 556314028545e6294c928040ef17455056cfe865
this is a new file called psi
=======
testing commits with file psi
>>>>>>> merged change: commit dd534e4160fac70063bc01126b4af1b87388fabd
```

Resolve the conflict by making the content of *psi* very pretty, for example:

```
this is a very pretty file called psi
```

Then save and exit the editor, and ask *got histedit* to continue once more. Edit the log message of the edited commit, and save and exit the text editor again.

Again, another merge conflict in *psi* might occur, similar to:

```
dd534e4160fa -> 86e3c31663ae: changing pretty psi
G  alpha
3bb0cb474042 -> 3655bb9010fb: modifying file alpha via staged changes
G  manylines
998ea5798df2 -> 208f99c46c41: changes to manylines file
```

```
G  manylines
bc0a7019f2fe -> d49a6ca915b2: modify manylines file via a staged change
G  manylines
0b4ccf44d2e7 -> b4ebace009e6: changing lines 3 and 4
C  psi
Files with new merge conflicts: 1
254f3b1d4f65 -> merge conflict: changing psi
got: conflicts must be resolved before histedit can continue
```

Resolve this conflict to your liking, and continue editing history again:

```
got histedit -c
```

Eventually, we will have passed all commits which touched file *psi*, and the histedit operation will conclude:

```
254f3b1d4f65 -> 1f6e6c84bf3b: changing psi
A  one
A  three
A  two
af1f25f1032b -> 3b0764397aee: add some numbers
A  five
A  four
A  six
a3682741cef4 -> f72839cf531c: adding more numbers
Switching work tree to refs/heads/main
```

The modified changes to file *psi* are now visible in commit history. Run:

```
got log -p psi
```

The output should only show the changes from the edited history of *psi*, similar to:

```
 ------------------------------------------------
 commit 1f6e6c84bf3b23efaacb4aa7446a31837d7badb5
 from: Tom Smith <tom.smith@example.com>
 date: Tue Sep  8 11:58:45 2026 UTC

  changing psi

 diff b4ebace009e63a9efa03e211ad66d583bc57f226 1f6e6c84bf3b23efaacb4aa7446a31837d7badb5
 commit - b4ebace009e63a9efa03e211ad66d583bc57f226
 commit + 1f6e6c84bf3b23efaacb4aa7446a31837d7badb5
 blob - ec14374bc372763e961f900a66797bd4294c929b
 blob + 08a7f4949d8e18d3969fe336a79a5650fa92d0e2
 --- psi
 +++ psi
 @@ -1 +1 @@
 -this is a very pretty file called psi
 +testing branches with a very pretty file called psi

 ------------------------------------------------
 commit 86e3c31663aef76e382fdb859a29b8721b5042c7
 from: Tom Smith <tom.smith@example.com>
 date: Tue Sep  8 11:56:44 2026 UTC

  changing pretty psi

 diff 6b1e5727da8bb34d4c92dc851665934b32e3ec93 86e3c31663aef76e382fdb859a29b8721b5042c7
 commit - 6b1e5727da8bb34d4c92dc851665934b32e3ec93
 commit + 86e3c31663aef76e382fdb859a29b8721b5042c7
```

```
blob - c1125bfe1eb29b9b4ff2814b92e26a0debfa8707
blob + ec14374bc372763e961f900a66797bd4294c929b
--- psi
+++ psi
@@ -1 +1 @@
-this is a pretty file called psi
+this is a very pretty file called psi

-----------------------------------------------
commit 16de732ca6efb9c19d2907df660922dbc7d603f0
from: Tom Smith <tom.smith@example.com>
date: Tue Sep  8 11:51:05 2026 UTC

 add pretty psi

diff 67fe73fbe97fe683ab162b05b1a6550209746569 16de732ca6efb9c19d2907df660922dbc7d603f0
commit - 67fe73fbe97fe683ab162b05b1a6550209746569
commit + 16de732ca6efb9c19d2907df660922dbc7d603f0
blob - /dev/null
blob + c1125bfe1eb29b9b4ff2814b92e26a0debfa8707
--- /dev/null
+++ psi
@@ -0,0 +1 @@
+this is a pretty file called psi
```

As seen in comments at the top of the histedit script, we are not limited to just editing file contents. We can also fold two or more commits into one commit with the *fold* command, drop commits from history with the *drop* command, and only edit log messages in history with the *mesg* command.

The *got histedit* command provides some shortcuts: If all you want to do is fold commits ("squashing" in Git's vernacular) then *got histedit -f* is the equivalent of replacing all *pick* commands except for the final *pick*, with *fold* commands. The same shortcut works for *drop* via *got histedit -d*, and for *mesg* via *got histedit -m*.

If you ever find yourself in a bad spot while editing history and change your mind, an interrupted *histedit* operation can be aborted with *got histedit -a*.

12.6 Backups of edited branches

The previous history of file *psi* is not really lost yet. Just like in the case of *got rebase*, the *got histedit* command keeps backups of the previous states of history. To see such backups, run:

```
got histedit -l
```

The output should be similar to:

```
-----------------------------------------------
commit a3682741cef4807ce0e85e561ad81812e9abe254 (formerly main)
from: Tom Smith <tom.smith@example.com>
date: Mon Sep  7 20:36:41 2026 UTC

 adding more numbers

has become commit f72839cf531cc79fd14a45fd18b1051bb8e93fb4 (main)
 2026-09-08 tom.smith  adding more numbers
history forked at 67fe73fbe97fe683ab162b05b1a6550209746569 (origin/main)
 2026-09-03 tom.smith  merge refs/remotes/origin/main into refs/heads/main
```

Using this information, we can view the history of file *psi* from before we started editing it:

```
got log -p -c a368274 psi
```

The output should display the previous version of the *psi* file's history, similar to:

```
------------------------------------------------
commit 254f3b1d4f6579ab4399306cd99ed89053062f2d
from: Tom Smith <tom.smith@example.com>
date: Mon Sep  7 20:13:55 2026 UTC

 changing psi

diff 0b4ccf44d2e710939c5eb54db13b9e6a685aa613 254f3b1d4f6579ab4399306cd99ed89053062f2d
commit - 0b4ccf44d2e710939c5eb54db13b9e6a685aa613
commit + 254f3b1d4f6579ab4399306cd99ed89053062f2d
blob - f741ca352ea516d8bdc2c339a87cbd8548fe279d
blob + 85bd6244f8779361ee84f00b31970c26bb9f641c
--- psi
+++ psi
@@ -1 +1 @@
-testing commits with file psi
+testing branches with file psi

------------------------------------------------
commit dd534e4160fac70063bc01126b4af1b87388fabd
from: Tom Smith <tom.smith@example.com>
date: Sat Sep  5 12:02:31 2026 UTC

 changing psi

diff 556314028545e6294c928040ef17455056cfe865 dd534e4160fac70063bc01126b4af1b87388fabd
commit - 556314028545e6294c928040ef17455056cfe865
commit + dd534e4160fac70063bc01126b4af1b87388fabd
blob - a3e542480f5fe385bbe7cd7afad1e0c2377999d6
blob + f741ca352ea516d8bdc2c339a87cbd8548fe279d
--- psi
+++ psi
@@ -1 +1 @@
-this is a new file called psi
+testing commits with file psi

------------------------------------------------
commit e7d64e8b65608cf6e3577527bbc967651d0641f4
from: Tom Smith <tom.smith@example.com>
date: Thu Sep  3 13:11:42 2026 UTC

 add psi

diff 67fe73fbe97fe683ab162b05b1a6550209746569 e7d64e8b65608cf6e3577527bbc967651d0641f4
commit - 67fe73fbe97fe683ab162b05b1a6550209746569
commit + e7d64e8b65608cf6e3577527bbc967651d0641f4
blob - /dev/null
blob + a3e542480f5fe385bbe7cd7afad1e0c2377999d6
--- /dev/null
+++ psi
@@ -0,0 +1 @@
+this is a new file called psi
```

As with *rebase*, backups can be removed with the *-X* option if they are no longer required:

```
got histedit -X
```

The output should be similar to:

```
Deleted refs/got/backup/histedit/main/f72839cf531cc79fd14a45fd18b1051bb8e93fb4:  ↩
    a3682741cef4807ce0e85e561ad81812e9abe254
```

Chapter 13

Managing stable software releases

With *got branch* and some additional commands, Game of Trees can be used to maintain a version of a software project which is almost frozen in time, but will still receive critical bug fixes.

Game of Trees uses *tag* objects to mark released versions of software, and enforces a convention where references which point at tag objects are stored in the *refs/tags* namespace.

Each tag points at a specific commit which represents the released state of the project. This commit might be on the *main* branch, or it might be on another branch such as a *stable* branch, or a branch named after the software's version number such as *1.x*, or *3.5.x*, or *v5*, or similar.

Names of tags are user-defined and no specific convention is enforced. Users should decide upon a suitable naming convention and stick to it. For example, if the branch which contains work-in-progress release state was called *1.x* then it would make sense to tag releases with names such as *1.0*, *1.1*, *1.2*, etc.

Each tag object contains a tag message field which can be used to store a user-defined message about the release.

In Game of Trees, tags are intentionally difficult to delete once created. Only the low-level *got ref* command can remove tags by removing the corresponding reference in the *refs/tags/* namespace.

References in the *refs/tags/* namespace objects are always automatically fetched by *got fetch*, under the assumption that tags are of global interest to every clone of the repository, including every object which is reachable via tags.

It is expected that unique tag objects will not collide under the same name across clones of the repository. Existing tags are not overwritten by default when a name collision occurs.

The *got send* command does not send tags by default, in order to prevent accidental propagation of tags which were created by mistake.

13.1 Creating a release branch

Creating a release branch is no different than creating any other branch. For example, we can create a *1.x.* branch in our work tree of *localrepo.git* as follows:

```
cd ~/tutorial/localrepo
got up -b main
got branch 1.x
```

The output should be:

```
Switching work tree from refs/heads/main to refs/heads/1.x
```

13.2 Creating tags

Tag objects are created with the *got tag* command. Let's say we are happy enough with the current state of our *1.x* release branch, and want to make our first *1.0* release from it.

We could create a tag in our work tree where the release branch is checked out by running:

```
got tag 1.0
```

We can also create one directly in the repository if we'd like to:

```
cd ~/tutorial/localrepo.git
got tag -c 1.x 1.0
```

In either case, a text editor will open and allow us to enter a tag message. The editor window should look similar to this:

```
# tagging commit f72839cf531cc79fd14a45fd18b1051bb8e93fb4 as 1.0
```

Enter a simple tag message such as "initial release 1.0", then save and exit the text editor. The *got tag* command should print something similar to:

```
Created tag 838c1a59ac659c4aa7883cf585dc10800cbdcec5
```

We can list all tags and their tag messages by running:

```
got tag -l
```

The output should look very similar to the output of *got log*:

```
------------------------------------------------
tag 1.0 031d598adf26beee9e512dce9cb62c6d00d791e1
from: Tom Smith <tom.smith@example.com>
date: Tue Sep  8 15:10:17 2026 UTC
object: commit f72839cf531cc79fd14a45fd18b1051bb8e93fb4

 initial release 1.0
```

And like with *got log*, there is also a summary mode:

```
got tag -l -s
```

The output should look similar to:

```
2026-09-08 commit:f72839cf53 1.0: initial release 1.0
```

13.3 Merging changes into the release branch

Some projects prefer to fix problems in the oldest supported release first, then merge the fix towards newer releases and eventually into the *main* branch. Other projects work the other way around, fixing problems in the *main* branch first. For our purposes, we will use the second approach. The steps to follow are indeed similar enough in either case that once you have seen one of the two approaches, you should be able to easily handle either of them.

Assuming we are fixing problems in the *main* branch first, switch your work tree of *localrepo.git* to the *main* branch, and commit three separate changes. Here, we use the *-m* option of *got commit* to pass a log message via the command line, which saves us from having to use a text editor to write log messages:

```
got update -b main
echo one fix > one
got commit -m 'fix one'
echo two fixes  > two
got commit -m 'fix two'
echo three fixes  > three
got commit -m 'fix three'
```

There are now three separate changes on the *main* branch. Imagine that only the second fix is critical enough to become part of the next release. In this case, we would merge just this particular change into the *1.x* release branch.

13.4 Addressing individual changes

To merge a specific change, we will need to select a specific commit on the command line.

There are several ways to "address" a change we committed. We could pass a literal or abbreviated commit object ID, or we could use the Game of Trees command line keyword syntax to address the change relative to a branch head.

Let's try out the keyword feature with *got log* first, before merging anything:

```
got log -l 1 -c main
```

This should print the third fix, i.e. the head commit of *main*, which is not the fix we want:

```
------------------------------------------------
commit 0d56f0b4fa519fe36248e29e0f8bc17c3b476ab4 (main)
from: Tom Smith <tom.smith@example.com>
date: Tue Sep  8 15:16:59 2026 UTC

 fix three
```

We can append a colon and a negative number after the branch name in order to have our commit selection step further down in history. Try:

```
got log -l 1 -c main:-2
```

```
------------------------------------------------
commit 0d0349edf13d7ccac77558a74bbb0d78073ac7c0
from: Tom Smith <tom.smith@example.com>
date: Tue Sep  8 15:16:34 2026 UTC

 fix one
```

This shows the first fix, which is still not the one we want. We want the second. The index of the branch head commit itself is zero, which means the head commit's parent commit can be addressed with index -1:

```
got log -l 1 -c main:-1
```

```
------------------------------------------------
commit 064380e87f3af6a5cfcade1f58183d91ff976dab
from: Tom Smith <tom.smith@example.com>
date: Tue Sep  8 15:16:46 2026 UTC

 fix two
```

There it is, this is the fix we want to merge.

13.5 Merging individual changes

In order to merge an individual change, we can use the *got cherrypick* command. The metaphor being that we can use this command to pick only the best cherries from the tree, or rather from the pile of changes we have collected in our repository.

Switch your work tree to the *1.x* branch and merge the second fix using the syntax discussed in the previous section:

```
got update -b 1.x
got cherrypick main:-1
```

The output of the *got cherrypick* command should be:

```
G  two
Merged commit 064380e87f3af6a5cfcade1f58183d91ff976dab
```

Given the above commit identifier, the following command would be equivalent to the one we just used:

```
got cherrypick 064380e87f3af6a5cfcade1f58183d91ff976dab
```

And of course, the commit identifier could be abbreviated, provided the abbreviation is unique.

As previously seen with *got backout*, the change is now sitting in our work tree as a local change. Run:

```
got diff
```

The output should be similar to:

```
diff /home/user/tutorial/localrepo
path + /home/user/tutorial/localrepo
commit - f72839cf531cc79fd14a45fd18b1051bb8e93fb4
blob - e69de29bb2d1d6434b8b29ae775ad8c2e48c5391
file + two
--- two
+++ two
@@ -0,0 +1 @@
+two fixes
```

At this point, we can make any required adjustments needed to apply the change in the context of our release branch, and test our changes. We could even run *got cherrypick* again to merge additional changes, and use commands which modify work tree state such as *got add* or *got remove*, or undo parts of the change with *got revert -p*. Staging changes and creating multiple commits would also be possible. But it is usually good to keep things simple, and merge and commit just one change at a time.

When we commit the change, the log message editor will be prefilled with the log message of the original change we have cherrypicked into the release branch. Run:

```
got commit
```

The text editor buffer should look similar to;

```
# log message of cherrypicked commit 064380e87f3af6a5cfcade1f58183d91ff976dab:
fix two

# changes to be committed on branch 1.x:
#  M  /two
# detailed changes can be viewed in /tmp/got-BZPuJo1mJs.diff
```

The log message buffer must be changed in some way for *got commit* to succeed. For example, by removing all the comments, or by adding additional text mentioning that the change is being backported to the 1.x branch. Save the file and exit the editor. Then we should see the usual output from *got commit*:

```
M  two
Created commit d249d48f4bc9a8b2bc2e1a1edfb795984bba87f7
```

Our *1.x* release branch now contains one additional change, while the other two changes on the *main* branch are not present here. Run:

```
got log -l2 -s
```

The output should be similar to:

```
2026-09-08 1.x     backporting fix two to the 1.x branch
2026-09-08 tags/1.0 adding more numbers
```

We could keep picking more changes from the *main* branch in this way, and eventually tag the next release version *1.1*.

Chapter 14

Browsing version control history

As a project grows over time, both in terms of lines of text, and in terms of commit history, it becomes easier to get lost. This section presents two tools which can support you when trying to make sense of a project's history.

14.1 Finding the origin of lines in a file

So far, we have been using the *got log* and *got diff* commands to view changes, but those tools are limited in scope and usually assume that we already know which commit we want to look at.

The *got blame* command can be used to annotate the lines of text in a particular file with commit IDs which last changed the line.

Try it out on the *manylines* file in the *localrepo.git* repository.

```
cd ~/tutorial/localrepo
got blame manylines
```

The output should be similar to:

```
01) b1210ac4 2026-09-08 tom.smit this is line 1 of a new file
02) 208f99c4 2026-09-08 tom.smit this is the second line of a new file
03) b4ebace0 2026-09-08 tom.smit this is the third line of a new file
04) b4ebace0 2026-09-08 tom.smit this is the fourth line of a new file
05) b1210ac4 2026-09-08 tom.smit this is line 5 of a new file
06) b1210ac4 2026-09-08 tom.smit this is line 6 of a new file
07) b1210ac4 2026-09-08 tom.smit this is line 7 of a new file
08) b1210ac4 2026-09-08 tom.smit this is line 8 of a new file
09) b1210ac4 2026-09-08 tom.smit this is line 9 of a new file
10) d49a6ca9 2026-09-08 tom.smit this is the tenth line of a new file
11) b1210ac4 2026-09-08 tom.smit this is line 11 of a new file
12) b1210ac4 2026-09-08 tom.smit this is line 12 of a new file
13) b1210ac4 2026-09-08 tom.smit this is line 13 of a new file
14) b1210ac4 2026-09-08 tom.smit this is line 14 of a new file
15) b1210ac4 2026-09-08 tom.smit this is line 15 of a new file
16) 208f99c4 2026-09-08 tom.smit this is the sixteenth line of a new file
17) b1210ac4 2026-09-08 tom.smit this is line 17 of a new file
18) b1210ac4 2026-09-08 tom.smit this is line 18 of a new file
19) b1210ac4 2026-09-08 tom.smit this is line 19 of a new file
20) b1210ac4 2026-09-08 tom.smit this is line 20 of a new file
```

Each line is annotated with its line number, commit ID, commit date stamp, and author. We can see that all lines in this file can be blamed on Tom.

We can now pass commit identifiers to *got log* or *got diff* as we are already used to in order to find out more about the change which introduced a particular line. With the above example output in mind, the following *got log* command will give us the log message of the commit which introduced line 17:

```
got log -l 1 -c 208f99c4
```

The output should match a usual *got log* change log entry:

```
-----------------------------------------------
commit 208f99c46c41683dfccb9b926ab6291b537b22ae
from: Tom Smith <tom.smith@example.com>
date: Tue Sep  8 11:56:44 2026 UTC

 changes to manylines file
```

14.2 Browsing repository history interactively

Game of Trees provides an interactive Git repository browser for terminals, called *tog*, which is similar but not equivalent to the third-party *tig* tool as known from the Git tooling ecosystem.

The *tog* command should be installed by default with Game of Trees. When started in a Game of Trees work tree, *tog* will show local changes sitting in the work tree as well as data in the repository.

Start tog in your work tree of *localrepo.git*:

```
cd ~/tutorial/localrepo
tog
```

With the work tree still switched to the *1.x* branch, the default view of *tog* should look something like this:

```
commit d249d48f4bc9a8b2bc2e1a1edfb795984bba87f7 [1/25] 1.x
2026-09-08 tom.smith ~[1.x] backporting fix two to the 1.x branch
2026-09-08 tom.smith  [tags/1.0] adding more numbers
2026-09-08 tom.smith  add some numbers
2026-09-08 tom.smith  changing psi
2026-09-08 tom.smith  changing lines 3 and 4
2026-09-08 tom.smith  modify manylines file via a staged change
2026-09-08 tom.smith  changes to manylines file
2026-09-08 tom.smith  modifying file alpha via staged changes
2026-09-08 tom.smith  changing pretty psi
2026-09-08 tom.smith  backout previous: tweak 2 lines
2026-09-08 tom.smith  tweak 2 lines
2026-09-08 tom.smith  add a file with many lines
2026-09-08 tom.smith  adding a line to file alpha
2026-09-08 tom.smith  remove ignore-me file again
2026-09-08 tom.smith  add ignored file
2026-09-08 tom.smith  add ignore list
2026-09-08 tom.smith  add pretty psi
2026-09-03 tom.smith  [origin/main] merge refs/remotes/origin/main into refs/hea
2026-09-02 tom.smith  changing file alpha
```

The default view of *tog* is very similar to *got log*. Use the arrow keys or vi-style navigation to select a particular commit.

Pressing Enter on a commit will open a new view which is equivalent to *got diff*. Depending on the horizontal size of your terminal, *tog* might display a split-screen showing both the *log* and *diff* view side-by-side, or it might display a full-screen *diff* view. In either case, press *q* to exit the *diff* view again.

Pressing *R* opens the list of references found in the repository. A *log* view which displays the corresponding branch can be opened by pressing Enter on a particular reference.

In a *log* view, pick any commit and press *T*. This opens a *tree* view which displays the snapshot recorded by this particular commit. Directories can be entered by pressing Enter, and a view equivalent to *got blame* can be opened by passing Enter on a file. For a realistic example, try the blame view of the *manylines* file, where pressing Enter on a particular line will open a *diff* view for the corresponding commit.

In any view, pressing F1 or H will display the available key bindings.

A manual page for *tog* should already be installed, and can also be viewed online at https://gameoftrees.org/tog.1.html

If you still have time left, try taking *tog* for a spin on real world repositories, guided by the built-in help feature and the manual page.