SAML Python Toolkit

Python package passing | PyPI Downloads 4.3k/month | coverage 94% | pypi v2.12.0 | python 2.7

`Python 2 was deprecated on January 1, 2020. We recommend to migrate your project to Python 3 and use python3-saml`

Add SAML support to your Python software using this library. Forget those complicated libraries and use the open source library.

This version supports Python2. There is a separate version that supports Python3: python3-saml.

Warning Version 2.7.0 sets strict mode active by default

Update `python-saml` to `2.5.0`, this version includes security improvements for preventing XEE and Xpath Injections.

Update `python-saml` to `2.4.0`, this version includes a fix for the CVE-2017-11427 vulnerability.

This version also changes how the calculate fingerprint method works, and will expect as input a formatted X.509 certificate

Update `python-saml` to `2.2.3`, this version replaces some etree.tostring calls, that were introduced recently, by the sanitized call provided by `defusedxml`

Update `python-saml` to `2.2.0`, this version includes a security patch that contains extra validations that will prevent signature wrapping attacks. CVE-2016-1000252

`python-saml` < `v2.2.0` is vulnerable and allows signature wrapping!

Security Guidelines If you believe you have discovered a security vulnerability in this toolkit, please report it by mail to the maintainer: sixto.martin.garcia+security@gmail.com

Why add SAML support to my software?

SAML is an XML-based standard for web browser single sign-on and is defined by the OASIS Security Services Technical Committee. The standard has been around since 2002, but lately it is becoming popular due its advantages:

- **Usability** - One-click access from portals or intranets, deep linking, password elimination and automatically renewing sessions make life easier for the user.

- **Security** - Based on strong digital signatures for authentication and integrity, SAML is a secure single sign-on protocol that the largest and most security conscious enterprises in the world rely on.
- **Speed** - SAML is fast. One browser redirect is all it takes to securely sign a user into an application.
- **Phishing Prevention** - If you don't have a password for an app, you can't be tricked into entering it on a fake login page.
- **IT Friendly** - SAML simplifies life for IT because it centralizes authentication, provides greater visibility and makes directory integration easier.
- **Opportunity** - B2B cloud vendor should support SAML to facilitate the integration of their product.

General Description

SAML Python toolkit lets you turn your Python application into a SP (Service Provider) that can be connected to an IdP (Identity Provider).

Supports:

- SSO and SLO (SP-Initiated and IdP-Initiated).
- Assertion and nameId encryption.
- Assertion signatures.
- Message signatures: `AuthNRequest`, `LogoutRequest`, `LogoutResponses`.
- Enable an Assertion Consumer Service endpoint.
- Enable a Single Logout Service endpoint.
- Publish the SP metadata (which can be signed).

Key features:

- **saml2int** - Implements the SAML 2.0 Web Browser SSO Profile.
- **Session-less** - Forget those common conflicts between the SP and the final app, the toolkit delegate session in the final app.
- **Easy to use** - Programmer will be allowed to code high-level and low-level programming, 2 easy to use APIs are available.
- **Tested** - Thoroughly tested.
- **Popular** - Developers use it. Add easy support to your Django/Flask/Bottle/Pyramid web projects.

Installation

Dependencies

- python 2.7
- lxml Python bindings for the libxml2 and libxslt libraries.
- dm.xmlsec.binding Cython/lxml based binding for the XML security library (depends on python-dev libxml2-dev libxmlsec1-dev)
- isodate An ISO 8601 date/time/duration parser and formater
- defusedxml XML bomb protection for Python stdlib modules

Review the setup.py file to know the version of the library that python-saml is using

OSX Dependencies

- python 2.7
- libxmlsec1

```
# using brew
$ brew install libxmlsec1
```

Code

Option 1. Download from Github The toolkit is hosted on Github. You can download it from:

- Lastest release: https://github.com/SAML-Toolkits/python-saml/releases/latest
- Master repo: https://github.com/SAML-Toolkits/python-saml/tree/master

Copy the core of the library `(src/onelogin/saml2 folder)` and merge the setup.py inside the Python application. (Each application has its structure so take your time to locate the Python SAML toolkit in the best place).

Option 2. Download from pypi The toolkit is hosted in pypi, you can find the `python-saml` package at https://pypi.python.org/pypi/python-saml

You can install it executing:

```
$ pip install python-saml
```

If you want to know how a project can handle python packages review this guide and review this sampleproject

NOTE To avoid `libxml2` library version incompatibilities between `xmlsec` and `lxml` it is recommended that `lxml` is not installed from binary.

This can be ensured by executing:

```
$ pip install --force-reinstall --no-binary lxml lxml
```

Security Warning

In production, the **strict** parameter MUST be set as **"true"**. Otherwise your environment is not secure and will be exposed to attacks.

In production also we highly recommend to register on the settings the IdP certificate instead of using the fingerprint method. The fingerprint, is a hash, so at the end is open to a collision attack that can end on a signature validation bypass. Other SAML toolkits deprecated that mechanism, we maintain it for compatibility and also to be used on test environment.

Avoiding Open Redirect attacks

Some implementations uses the RelayState parameter as a way to control the flow when SSO and SLO succeeded. So basically the user is redirected to the value of the RelayState.

If you are using Signature Validation on the HTTP-Redirect binding, you will have the RelayState value integrity covered, otherwise, and on HTTP-POST binding, you can't trust the RelayState so before executing the validation, you need to verify that its value belong a trusted and expected URL.

Read more about Open Redirect CWE-601.

Avoiding Replay attacks

A replay attack is basically try to reuse an intercepted valid SAML Message in order to impersonate a SAML action (SSO or SLO).

SAML Messages have a limited timelife (NotBefore, NotOnOrAfter) that make harder this kind of attacks, but they are still possible.

In order to avoid them, the SP can keep a list of SAML Messages or Assertion IDs alredy valdidated and processed. Those values only need to be stored the amount of time of the SAML Message life time, so we don't need to store all processed message/assertion Ids, but the most recent ones.

The OneLogin_Saml2_Auth class contains the get_last_request_id, get_last_message_id and get_last_assertion_id methods to retrieve the IDs

Checking that the ID of the current Message/Assertion does not exists in the lis of the ones already processed will prevent replay attacks.

Getting Started

Knowing the toolkit

The SAML Toolkit contains different folders (`cert`, `lib`, `demo-django`, `demo-flask`, `demo-bottle` and `tests`) and some files.

Let's start describing them:

src This folder contains the heart of the toolkit, **onelogin/saml2** folder contains the new version of the classes and methods that are described in a later section.

demo-django This folder contains a Django project that will be used as demo to show how to add SAML support to the Django Framework. **demo** is the main folder of the Django project (with its `settings.py`, `views.py`, `urls.py`), **templates** is the Django templates of the project and **saml** is a folder that contains the 'certs' folder that could be used to store the X.509

public and private key, and the SAML toolkit settings (`settings.json` and `advanced_settings.json`).

*** Notice about certs ***

SAML requires a x.509 cert to sign and encrypt elements like `NameID`, `Message`, `Assertion`, `Metadata`.

If our environment requires sign or encrypt support, the certs folder may contain the X.509 cert and the private key that the SP will use:

- sp.crt The public cert of the SP
- sp.key The private key of the SP

Or also we can provide those data in the setting file at the 'x509cert' and the privateKey' JSON parameters of the `sp` element.

Sometimes we could need a signature on the metadata published by the SP, in this case we could use the x.509 cert previously mentioned or use a new x.509 cert: `metadata.crt` and `metadata.key`.

Use `sp_new.crt` if you are in a key rollover process and you want to publish that X.509 certificate on Service Provider metadata.

If you want to create self-signed certs, you can do it at the https://www.samltool.com/self_signed_certs.php service, or using the command:

```
openssl req -new -x509 -days 3652 -nodes -out sp.crt -keyout saml.key
```

demo-bottle This folder contains a Bottle project that will be used as demo to show how to add SAML support to the Bottle Framework. `index.py` contains all the logic of the demo project, **templates** is the Bottle templates of the project and **saml** is a folder that contains the 'certs' folder that could be used to store the X.509 public and private key, and the SAML toolkit settings (`settings.json` and `advanced_settings.json`).

demo-flask This folder contains a Flask project that will be used as demo to show how to add SAML support to the Flask Framework. `index.py` is the main Flask file that has all the code, this file uses the templates stored at the 'templates' folder. In the 'saml' folder we found the 'certs' folder to store the X.509 public and private key, and the SAML toolkit settings (`settings.json` and `advanced_settings.json`).

demo_pyramid This folder contains a Pyramid project that will be used as demo to show how to add SAML support to the Pyramid Web Framework. `\_\_init__.py` is the main file that configures the app and its routes, `views.py` is where all the logic and SAML handling takes place, and the templates are stored in the **templates** folder. The **saml** folder is the same as in the other two demos.

setup.py Setup script is the centre of all activity in building, distributing, and installing modules. Read more at https://pythonhosted.org/an_example_pypi_project/setuptools.html

tests Contains the unit test of the toolkit.

In order to execute the test you need to load the `virtualenv` with the toolkit installed on it and execute:

```
pip install -e ".[test]"
```

that will install dependences that the test requires.

and later execute:

```
python setup.py test
```

The previous line will run the tests for the whole toolkit. You can also run the tests for a specific module. To do so for the auth module you would have to execute this:

```
python setup.py test --test-suite tests.src.OneLogin.saml2_tests.auth_test.OneLogin_Saml2_Au
```

With the `--test-suite` parameter you can specify the module to test. You'll find all the module available and their class names at `tests/src/OneLogin/saml2_tests/`

How it Works

Settings First of all we need to configure the toolkit. The SP's info, the IdP's info, and in some cases, configure advanced security issues like signatures and encryption.

There are two ways to provide the settings information:

- Use a `settings.json` file that we should locate in any folder, but indicates its path with the `custom_base_path` parameter.
- Use a JSON object with the setting data and provide it directly to the constructor of the class (if your toolkit integation requires certs, remember to provide the `custom_base_path` as part of the settings or as a parameter in the constructor.

In the `demo-django`, `demo-flask`, `demo-pyramid` and `demo-bottle` folders you will find a `saml` folder, inside there is a `certs` folder and a `settings.json` and a `advanced_settings.json` files. Those files contain the settings for the SAML toolkit. Copy them in your project and set the correct values.

This is the `settings.json` file:

```
{
    // If strict is True, then the Python Toolkit will reject unsigned
    // or unencrypted messages if it expects them to be signed or encrypted.
    // Also it will reject the messages if the SAML standard is not strictly
    // followed. Destination, NameId, Conditions ... are validated too.
    "strict": true,

    // Enable debug mode (outputs errors).
    "debug": true,

    // Service Provider Data that we are deploying.
    "sp": {
        // Identifier of the SP entity  (must be a URI)
        "entityId": "https://<sp_domain>/metadata/",
        // Specifies info about where and how the <AuthnResponse> message MUST be
        // returned to the requester, in this case our SP.
        "assertionConsumerService": {
            // URL Location where the <Response> from the IdP will be returned
            "url": "https://<sp_domain>/?acs",
            // SAML protocol binding to be used when returning the <Response>
            // message. SAML Toolkit supports this endpoint for the
            // HTTP-POST binding only.
            "binding": "urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
        },
        // If you need to specify requested attributes, set a
        // attributeConsumingService. nameFormat, attributeValue and
        // friendlyName can be omitted
        "attributeConsumingService": {
                "serviceName": "SP test",
                "serviceDescription": "Test Service",
                "requestedAttributes": [
                    {
                        "name": "",
                        "isRequired": false,
                        "nameFormat": "",
                        "friendlyName": "",
                        "attributeValue": []
                    }
                ]
        },
        // Specifies info about where and how the <Logout Request/Response> message MUST be
        "singleLogoutService": {
            // URL Location where the <LogoutRequest> from the IdP will be sent (IdP-initia
            "url": "https://<sp_domain>/?sls",
            // URL Location where the <LogoutResponse> from the IdP will sent (SP-initiated
            // OPTIONAL: only specify if different from url parameter
            //"responseUrl": "https://<sp_domain>/?sls",
            // SAML protocol binding to be used when returning the <Response>
            // message. SAML Toolkit supports the HTTP-Redirect binding
            // only for this endpoint.
```

```
            "binding": "urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect"
        },
        // Specifies the constraints on the name identifier to be used to
        // represent the requested subject.
        // Take a look on src/onelogin/saml2/constants.py to see the NameIdFormat that are s
        "NameIDFormat": "urn:oasis:names:tc:SAML:1.1:nameid-format:unspecified",
        // Usually X.509 cert and privateKey of the SP are provided by files placed at
        // the certs folder. But we can also provide them with the following parameters
        "x509cert": "",
        "privateKey": ""

        /*
         * Key rollover
         * If you plan to update the SP X.509 cert and privateKey
         * you can define here the new X.509 cert and it will be
         * published on the SP metadata so Identity Providers can
         * read them and get ready for rollover.
         */
        // 'x509certNew': '',
    },

    // Identity Provider Data that we want connected with our SP.
    "idp": {
        // Identifier of the IdP entity  (must be a URI)
        "entityId": "https://app.onelogin.com/saml/metadata/<onelogin_connector_id>",
        // SSO endpoint info of the IdP. (Authentication Request protocol)
        "singleSignOnService": {
            // URL Target of the IdP where the Authentication Request Message
            // will be sent.
            "url": "https://app.onelogin.com/trust/saml2/http-post/sso/<onelogin_connector_i
            // SAML protocol binding to be used when returning the <Response>
            // message. SAML Toolkit supports the HTTP-Redirect binding
            // only for this endpoint.
            "binding": "urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect"
        },
        // SLO endpoint info of the IdP.
        "singleLogoutService": {
            // URL Location where the <LogoutRequest> from the IdP will be sent (IdP-initiat
            "url": "https://app.onelogin.com/trust/saml2/http-redirect/slo/<onelogin_connect
            // URL Location where the <LogoutResponse> from the IdP will sent (SP-initiated
            // OPTIONAL: only specify if different from url parameter
            "responseUrl": "https://app.onelogin.com/trust/saml2/http-redirect/slo_return/<o
            // SAML protocol binding to be used when returning the <Response>
            // message. SAML Toolkit supports the HTTP-Redirect binding
            // only for this endpoint.
            "binding": "urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect"
        },
        // Public X.509 certificate of the IdP
        "x509cert": "<onelogin_connector_cert>"
        /*
         *  Instead of using the whole X.509 cert you can use a fingerprint in order to
         *  validate a SAMLResponse (but you still need the X.509 cert to validate LogoutRe
         *  But take in mind that the fingerprint, is a hash, so at the end is open to a co
         *  that why we don't recommend it use for production environments.
         *
         *  (openssl x509 -noout -fingerprint -in "idp.crt" to generate it,
         *  or add for example the -sha256 , -sha384 or -sha512 parameter)
         *
         *  If a fingerprint is provided, then the certFingerprintAlgorithm is required in 
         *  let the toolkit know which algorithm was used. Possible values: sha1, sha256, s
         *  'sha1' is the default value.
         *
         *  Notice that if you want to validate any SAML Message sent by the HTTP-Redirect 
         *  will need to provide the whole X.509 cert.
     *
         */
        // 'certFingerprint': '',
        // 'certFingerprintAlgorithm': 'sha1',

        /* In some scenarios the IdP uses different certificates for
         * signing/encryption, or is under key rollover phase and
         * more than one certificate is published on IdP metadata.
         * In order to handle that the toolkit offers that parameter.
         * (when used, 'x509cert' and 'certFingerprint' values are
         * ignored).
         */
        // 'x509certMulti': {
        //      'signing': [
        //          '<cert1-string>'
        //      ],
        //      'encryption': [
        //          '<cert2-string>'
        //      ]
        // }
    }
}
```

In addition to the required settings data (idp, sp), extra settings can be defined in `advanced_settings.json`:

```
{
    // Security settings
    "security": {
```

```
/** signatures and encryptions offered **/

// Indicates that the nameID of the <samlp:logoutRequest> sent by this SP
// will be encrypted.
"nameIdEncrypted": false,

// Indicates whether the <samlp:AuthnRequest> messages sent by this SP
// will be signed.  [Metadata of the SP will offer this info]
"authnRequestsSigned": false,

// Indicates whether the <samlp:logoutRequest> messages sent by this SP
// will be signed.
"logoutRequestSigned": false,

// Indicates whether the <samlp:logoutResponse> messages sent by this SP
// will be signed.
"logoutResponseSigned": false,

/* Sign the Metadata
 false || true (use sp certs) || {
                                    "keyFileName": "metadata.key",
                                    "certFileName": "metadata.crt"
                                 }
*/
"signMetadata": false,

/** signatures and encryptions required **/

// Indicates a requirement for the <samlp:Response>, <samlp:LogoutRequest>
// and <samlp:LogoutResponse> elements received by this SP to be signed.
"wantMessagesSigned": false,

// Indicates a requirement for the <saml:Assertion> elements received by
// this SP to be signed. [Metadata of the SP will offer this info]
"wantAssertionsSigned": false,

// Indicates a requirement for the <saml:Assertion>
// elements received by this SP to be encrypted.
"wantAssertionsEncrypted": false,

// Indicates a requirement for the NameID element on the SAMLResponse
// received by this SP to be present.
"wantNameId": true,

// Indicates a requirement for the NameID received by
// this SP to be encrypted.
"wantNameIdEncrypted": false,

// Indicates a requirement for the AttributeStatement element
"wantAttributeStatement": true,

// Rejects SAML responses with a InResponseTo attribute when request_id
// not provided in the process_response method that later call the
// response is_valid method with that parameter.
"rejectUnsolicitedResponsesWithInResponseTo": false,

// Authentication context.
// Set to false and no AuthContext will be sent in the AuthNRequest,
// Set true or don't present this parameter and you will get an AuthContext 'exact'
// Set an array with the possible auth context values: array ('urn:oasis:names:tc:S
"requestedAuthnContext": true,
// Allows the authn comparison parameter to be set, defaults to 'exact' if the sett
"requestedAuthnContextComparison": "exact",

// Set to true to check that the AuthnContext(s) received match(es) the requested.
"failOnAuthnContextMismatch": false,

// In some environment you will need to set how long the published metadata of the S
// is possible to not set the 2 following parameters (or set to null) and default va
// Provide the desired Timestamp, for example 2015-06-26T20:00:00Z
"metadataValidUntil": null,
// Provide the desired duration, for example PT518400S (6 days)
"metadataCacheDuration": null,

// If enabled, URLs with single-label-domains will
// be allowed and not rejected by the settings validator (Enable it under Docker/Kub
"allowSingleLabelDomains": false,

// Algorithm that the toolkit will use on signing process. Options:
//    'http://www.w3.org/2000/09/xmldsig#rsa-sha1'
//    'http://www.w3.org/2000/09/xmldsig#dsa-sha1'
//    'http://www.w3.org/2001/04/xmldsig-more#rsa-sha256'
//    'http://www.w3.org/2001/04/xmldsig-more#rsa-sha384'
//    'http://www.w3.org/2001/04/xmldsig-more#rsa-sha512'
"signatureAlgorithm": "http://www.w3.org/2001/04/xmldsig-more#rsa-sha256",

// Algorithm that the toolkit will use on digest process. Options:
//    'http://www.w3.org/2000/09/xmldsig#sha1'
//    'http://www.w3.org/2001/04/xmlenc#sha256'
//    'http://www.w3.org/2001/04/xmldsig-more#sha384'
//    'http://www.w3.org/2001/04/xmlenc#sha512'
```

```
        "digestAlgorithm": "http://www.w3.org/2001/04/xmlenc#sha256",

        // If the toolkit receive a message signed with a
        // deprecated algoritm (defined at the constant class)
        // will raise an error and reject the message
        "rejectDeprecatedAlgorithm": true
    },

    // Contact information template, it is recommended to supply
    // technical and support contacts.
    "contactPerson": {
        "technical": {
            "givenName": "technical_name",
            "emailAddress": "technical@example.com"
        },
        "support": {
            "givenName": "support_name",
            "emailAddress": "support@example.com"
        }
    },

    // Organization information template, the info in en_US lang is
    // recommended, add more if required.
    "organization": {
        "en-US": {
            "name": "sp_test",
            "displayname": "SP test",
            "url": "http://sp.example.com"
        }
    }
}
```

In the `security` section, you can set the way that the SP will handle the messages and assertions. Contact the admin of the IdP and ask them what the IdP expects, and decide what validations will handle the SP and what requirements the SP will have and communicate them to the IdP's admin too.

Once we know what kind of data could be configured, let's talk about the way settings are handled within the toolkit.

The settings files described (`settings.json` and `advanced_settings.json`) are loaded by the toolkit if not other dict with settings info is provided in the constructors of the toolkit. Let's see some examples.

```
# Initializes toolkit with settings.json & advanced_settings.json files.
auth = OneLogin_Saml2_Auth(req)
# or
settings = OneLogin_Saml2_Settings()
```

```
# Initializes toolkit with settings.json & advanced_settings.json files from a custom base p
custom_folder = '/var/www/django-project'
auth = OneLogin_Saml2_Auth(req, custom_base_path=custom_folder)
# or
settings = OneLogin_Saml2_Settings(custom_base_path=custom_folder)

# Initializes toolkit with the dict provided.
auth = OneLogin_Saml2_Auth(req, settings_data)
# or
settings = OneLogin_Saml2_Settings(settings_data)
```

You can declare the `settings_data` in the file that constains the constructor execution or locate them in any file and load the file in order to get the dict available as we see in the following example:

```
filename = "/var/www/django-project/custom_settings.json" # The custom_settings.json contai
json_data_file = open(filename, 'r')                      # settings_data dict.
settings_data = json.load(json_data_file)
json_data_file.close()

auth = OneLogin_Saml2_Auth(req, settings_data)
```

Metadata Based Configuration The method above requires a little extra work to manually specify attributes about the IdP. (And your SP application)

There's an easier method – use a metadata exchange. Metadata is just an XML file that defines the capabilities of both the IdP and the SP application. It also contains the X.509 public key certificates which add to the trusted relationship. The IdP administrator can also configure custom settings for an SP based on the metadata.

Using `parse_remote` IdP metadata can be obtained and added to the settings withouth further ado.

But take in mind that the OneLogin_Saml2_IdPMetadataParser class does not validate in any way the URL that is introduced in order to be parsed.

Usually the same administrator that handles the Service Provider also sets the URL to the IdP, which should be a trusted resource.

But there are other scenarios, like a SAAS app where the administrator of the app delegates this functionality to other users. In this case, extra precaution should be taken in order to validate such URL inputs and avoid attacks like SSRF.

```
idp_data = OneLogin_Saml2_IdPMetadataParser.parse_remote('https://example.com/auth/saml2/idp
```

If the Metadata contains several entities, the relevant `EntityDescriptor` can be specified when retrieving the settings from the `IdpMetadataParser` by its

EntityId value:

```
idp_data = OneLogin_Saml2_IdPMetadataParser.parse_remote(https://example.com/metadatas, enti
```

How load the library In order to use the toolkit library you need to import the file that contains the class that you will need on the top of your python file.

```
from onelogin.saml2.auth import OneLogin_Saml2_Auth
from onelogin.saml2.settings import OneLogin_Saml2_Settings
from onelogin.saml2.utils import OneLogin_Saml2_Utils
```

The Request Building an `OneLogin_Saml2_Auth object` requires a `request` parameter.

```
auth = OneLogin_Saml2_Auth(req)
```

This parameter has the following scheme:

```
req = {
    "https": ""
    "http_host": "",
    "script_name": "",
    "server_port": "",
    "get_data": "",
    "post_data": ""
}
```

Each Python framework built its own `request` object, you may map its data to match what the SAML toolkit expects. Let‘s see some examples:

```
def prepare_from_django_request(request):
    return {
        'http_host': request.META['HTTP_HOST'],
        'script_name': request.META['PATH_INFO'],
        'server_port': request.META['SERVER_PORT'],
        'get_data': request.GET.copy(),
        'post_data': request.POST.copy()
    }

def prepare_from_flask_request(request):
    url_data = urlparse(request.url)
    return {
        'http_host': request.host,
        'server_port': url_data.port,
        'script_name': request.path,
        'get_data': request.args.copy(),
        'post_data': request.form.copy()
    }
```

The `https` dictionary entry should be set to `on` for https requests and `off` for http

Initiate SSO In order to send an AuthNRequest to the IdP:

```
from onelogin.saml2.auth import OneLogin_Saml2_Auth

req = prepare_request_for_toolkit(request)
auth = OneLogin_Saml2_Auth(req)   # Constructor of the SP, loads settings.json
                                  # and advanced_settings.json

auth.login()      # Method that builds and sends the AuthNRequest
```

The `AuthNRequest` will be sent signed or unsigned based on the security info of the `advanced_settings.json` (`authnRequestsSigned`).

The IdP will then return the SAML Response to the user's client. The client is then forwarded to the **Assertion Consumer Service (ACS)** of the SP with this information.

We can set a `return_to` url parameter to the login function and that will be converted as a `RelayState` parameter:

```
target_url = 'https://example.com'
auth.login(return_to=target_url)
```

The login method can recieve 4 more optional parameters:

- `force_authn` When `true` the `AuthNReuqest` will set the `ForceAuthn='true'`
- `is_passive` When `true` the `AuthNReuqest` will set the `Ispassive='true'`
- `set_nameid_policy` When `true` the `AuthNReuqest` will set a `nameIdPolicy` element.
- `name_id_value_req` Indicates to the IdP the `Subject` that should be authenticated

If a match on the future SAMLResponse ID and the AuthNRequest ID to be sent is required, that AuthNRequest ID must to be extracted and stored for future validation, we can get that ID by

```
auth.get_last_request_id()
```

The SP Endpoints Related to the SP there are 3 important endpoints: The metadata view, the ACS view and the SLS view. The toolkit provides examples of those views in the demos, but lets see an example.

*** SP Metadata ***

This code will provide the XML metadata file of our SP, based on the info that we provided in the settings files.

```
req = prepare_request_for_toolkit(request)
auth = OneLogin_Saml2_Auth(req)
saml_settings = auth.get_settings()
metadata = saml_settings.get_sp_metadata()
errors = saml_settings.validate_metadata(metadata)
if len(errors) == 0:
    print metadata
else:
    print "Error found on Metadata: %s" % (', '.join(errors))
```

The `get_sp_metadata` will return the metadata signed or not based on the security info of the `advanced_settings.json` (`signMetadata`).

Before the XML metadata is exposed, a check takes place to ensure that the info to be provided is valid.

Instead of using the `Auth` object, you can directly use

```
saml_settings = OneLogin_Saml2_Settings(settings=None, custom_base_path=None, sp_validation_
```

to get the settings object and with the `sp_validation_only=True` parameter we will avoid the IdP settings validation.

*** Assertion Consumer Service (ACS) ***

This code handles the SAML response that the IdP forwards to the SP through the user's client.

```
req = prepare_request_for_toolkit(request)
auth = OneLogin_Saml2_Auth(req)
auth.process_response()
errors = auth.get_errors()
if not errors:
    if auth.is_authenticated():
        request.session['samlUserdata'] = auth.get_attributes()
        if 'RelayState' in req['post_data'] and
          OneLogin_Saml2_Utils.get_self_url(req) != req['post_data']['RelayState']:
            # To avoid 'Open Redirect' attacks, before execute the redirection confirm
            # the value of the req['post_data']['RelayState'] is a trusted URL.
            auth.redirect_to(req['post_data']['RelayState'])
        else:
            for attr_name in request.session['samlUserdata'].keys():
                print '%s ==> %s' % (attr_name, '|| '.join(request.session['samlUserdata'][a
    else:
      print 'Not authenticated'
else:
    print "Error when processing SAML Response: %s" % (', '.join(errors))
```

The SAML response is processed and then checked that there are no errors. It also verifies that the user is authenticated and stored the userdata in session.

At that point there are 2 possible alternatives:

- If no `RelayState` is provided, we could show the user data in this view or however we wanted.
- If `RelayState` is provided, a rediretion take place.

Notice that we saved the user data in the session before the redirection to have the user data available at the `RelayState` view.

In order to retrieve attributes we use:

```
attributes = auth.get_attributes();
```

With this method we get a dict with all the user data provided by the IdP in the Assertion of the SAML Response.

If we execute print attributes we could get:

```
{
    "cn": ["Jhon"],
    "sn": ["Doe"],
    "mail": ["Doe"],
    "groups": ["users", "members"]
}
```

Each attribute name can be used as a key to obtain the value. Every attribute is a list of values. A single-valued attribute is a listy of a single element.

The following code is equivalent:

```
attributes = auth.get_attributes();
print attributes['cn']

print auth.get_attribute('cn')
```

Before trying to get an attribute, check that the user is authenticated. If the user isn't authenticated, an empty dict will be returned. For example, if we call to `auth.get_attributes` before a `auth.process_response`, the `auth.get_attributes` will return an empty dict.

*** Single Logout Service (SLS) ***

This code handles the Logout Request and the Logout Responses.

```
delete_session_callback = lambda: request.session.flush()
url = auth.process_slo(delete_session_cb=delete_session_callback)
errors = auth.get_errors()
if len(errors) == 0:
    if url is not None:
        # To avoid 'Open Redirect' attacks, before execute the redirection confirm
        # the value of the url is a trusted URL.
        return redirect(url)
    else:
        print "Sucessfully Logged out"
else:
    print "Error when processing SLO: %s" % (', '.join(errors))
```

If the SLS endpoints receives a Logout Response, the response is validated and the session could be closed, using the callback.

```
# Part of the process_slo method
logout_response = OneLogin_Saml2_Logout_Response(self.__settings, self.__request_data['get_d
if not logout_response.is_valid(self.__request_data, request_id):
    self.__errors.append('invalid_logout_response')
elif logout_response.get_status() != OneLogin_Saml2_Constants.STATUS_SUCCESS:
    self.__errors.append('logout_not_success')
elif not keep_local_session:
    OneLogin_Saml2_Utils.delete_local_session(delete_session_cb)
```

If the SLS endpoints receives an Logout Request, the request is validated, the session is closed and a Logout Response is sent to the SLS endpoint of the IdP.

```
# Part of the process_slo method
request = OneLogin_Saml2_Utils.decode_base64_and_inflate(self.__request_data['get_data']['SA
if not OneLogin_Saml2_Logout_Request.is_valid(self.__settings, request, self.__request_data)
    self.__errors.append('invalid_logout_request')
else:
    if not keep_local_session:
        OneLogin_Saml2_Utils.delete_local_session(delete_session_cb)

    in_response_to = request.id
    response_builder = OneLogin_Saml2_Logout_Response(self.__settings)
    response_builder.build(in_response_to)
    logout_response = response_builder.get_response()

    parameters = {'SAMLResponse': logout_response}
    if 'RelayState' in self.__request_data['get_data']:
        parameters['RelayState'] = self.__request_data['get_data']['RelayState']

    security = self.__settings.get_security_data()
    if 'logoutResponseSigned' in security and security['logoutResponseSigned']:
        parameters['SigAlg'] = OneLogin_Saml2_Constants.RSA_SHA256
        parameters['Signature'] = self.build_response_signature(logout_response, parameters.

    return self.redirect_to(self.get_slo_url(), parameters)
```

If we don't want that `process_slo` to destroy the session, pass a `true` parameter to the `process_slo` method

```
keepLocalSession = true
auth.process_slo(keep_local_session=keepLocalSession);
```

Initiate SLO In order to send a Logout Request to the IdP:

```
from onelogin.saml2.auth import OneLogin_Saml2_Auth

req = prepare_request_for_toolkit(request)
auth = OneLogin_Saml2_Auth(req)    # Constructor of the SP, loads settings.json
                                   # and advanced_settings.json

auth.logout()       # Method that builds and sends the LogoutRequest
```

The Logout Request will be sent signed or unsigned based on the security info of the `advanced_settings.json` (`logoutRequestSigned`).

The IdP will return the Logout Response through the user's client to the Single Logout Service (SLS) of the SP.

We can set a `return_to` url parameter to the logout function and that will be converted as a `RelayState` parameter:

```
target_url = 'https://example.com'
auth.logout(return_to=target_url)
```

Also there are another 5 optional parameters that can be set:

- `name_id`. That will be used to build the LogoutRequest. If not `name_id` parameter is set and the auth object processed a SAML Response with a NameId, then this NameId will be used.
- `session_index`. SessionIndex that identifies the session of the user.
- `nq`. IDP Name Qualifier
- `name_id_format`. The NameID Format that will be set in the LogoutRequest
- `spnq`: The `NameID SP NameQualifier` will be set in the `LogoutRequest`.

If no name_id is provided, the LogoutRequest will contain a NameID with the entity Format. If name_id is provided and no name_id_format is provided, the NameIDFormat of the settings will be used.

If a match on the LogoutResponse ID and the LogoutRequest ID to be sent is required, that LogoutRequest ID must to be extracted and stored for future validation, we can get that ID by:

```
auth.get_last_request_id()
```

Example of a view that initiates the SSO request and handles the response (is the acs target) We can code a unique file that initiates the SSO process, handle the response, get the attributes, initiate the slo and processes the logout response.

Note: Review the demos, in a later section we explain the demo use case further in detail.

```
req = prepare_request_for_toolkit(request)  # Process the request and build the request dic
                                            # the toolkit expects

auth = OneLogin_Saml2_Auth(req)             # Initialize the SP SAML instance

if 'sso' in request.args:                   # SSO action (SP-SSO initited).  Will send an A
    return redirect(auth.login())
elif 'sso2' in request.args:                       # Another SSO init action
    return_to = '%sattrs/' % request.host_url      # but set a custom RelayState URL
    return redirect(auth.login(return_to))
elif 'slo' in request.args:                     # SLO action. Will sent a Logout Request to
    nameid = request.session['samlNameId']
    nameid_format = request.session['samlNameIdFormat']
    nameid_nq = request.session['samlNameIdNameQualifier']
    nameid_spnq = request.session['samlNameIdSPNameQualifier']
    session_index = request.session['samlSessionIndex']
    return redirect(auth.logout(None, nameid, session_index, nameid_nq, nameid_format, namei
elif 'acs' in request.args:                 # Assertion Consumer Service
    auth.process_response()                     # Process the Response of the IdP
    errors = auth.get_errors()              # This method receives an array with the errors
    if len(errors) == 0:                    # that could took place during the process
        if not auth.is_authenticated():         # This check if the response was ok and the
            msg = "Not authenticated"           # data retrieved or not (user authenticated
        else:
            request.session['samlUserdata'] = auth.get_attributes()     # Retrieves user da
            request.session['samlNameId'] = auth.get_nameid()
            request.session['samlNameIdFormat'] = auth.get_nameid_format()
            request.session['samlNameIdNameQualifier'] = auth.get_nameid_nq()
            request.session['samlNameIdSPNameQualifier'] = auth.get_nameid_spnq()
            request.session['samlSessionIndex'] = auth.get_session_index()
            self_url = OneLogin_Saml2_Utils.get_self_url(req)
            if 'RelayState' in request.form and self_url != request.form['RelayState']:
                # To avoid 'Open Redirect' attacks, before execute the redirection confirm
                # the value of the request.form['RelayState'] is a trusted URL.
                return redirect(request.form['RelayState'])   # Redirect if there is a rela
            else:                           # If there is user data we save that to print i
                msg = ''
                for attr_name in request.session['samlUserdata'].keys():
                    msg += '%s ==> %s' % (attr_name, '|| '.join(request.session['samlUserdat
elif 'sls' in request.args:                                         # Single Logout Ser
    delete_session_callback = lambda: session.clear()           # Obtain session clear call
    url = auth.process_slo(delete_session_cb=delete_session_callback)   # Process the Logou
    errors = auth.get_errors()              #  Retrieves possible validation errors
    if len(errors) == 0:
        if url is not None:
            # To avoid 'Open Redirect' attacks, before execute the redirection confirm
            # the value of the url is a trusted URL.
            return redirect(url)
        else:
            msg = "Sucessfully logged out"

if len(errors) == 0:
  print msg
else:
  print ', '.join(errors)
```

SP Key rollover

If you plan to update the SP X.509 cert and privateKey you can define the new X.509 cert as `settings['sp']['x509certNew']` and it will be published on the SP metadata so Identity Providers can read them and get ready for rollover.

IdP with multiple certificates

In some scenarios the IdP uses different certificates for signing/encryption, or is under key rollover phase and more than one certificate is published on IdP metadata.

In order to handle that the toolkit offers the `settings['idp']['x509certMulti']` parameter.

When that parameter is used, `x509cert` and `certFingerprint` values will be ignored by the toolkit.

The `x509certMulti` is an array with 2 keys: - `signing`. An array of certs that will be used to validate IdP signature - `encryption` An array with one unique cert that will be used to encrypt data to be sent to the IdP

Replay attacks

In order to avoid replay attacks, you can store the ID of the SAML messages already processed, to avoid processing them twice. Since the Messages expires and will be invalidated due that fact, you don't need to store those IDs longer than the time frame that you currently accepting.

Get the ID of the last processed message/assertion with the `get_last_message_id/get_last_assertion_id method` of the `Auth` object.

Main classes and methods

Described below are the main classes and methods that can be invoked from the SAML2 library.

OneLogin_Saml2_Auth - auth.py Main class of SAML Python Toolkit

- `__init__` Initializes the SP SAML instance.
- ***login*** Initiates the SSO process.
- ***logout*** Initiates the SLO process.
- ***process_response*** Process the SAML Response sent by the IdP.
- ***process_slo*** Process the SAML Logout Response / Logout Request sent by the IdP.
- ***redirect_to*** Redirects the user to the url past by parameter or to the url that we defined in our SSO Request.
- ***is_authenticated*** Checks if the user is authenticated or not.
- ***get_attributes*** Returns the set of SAML attributes.
- ***get_attribute*** Returns the requested SAML attribute.
- ***get_nameid*** Returns the `nameID`.
- ***get_session_index*** Gets the `SessionIndex` from the `AuthnStatement`.
- ***get_session_expiration*** Gets the `SessionNotOnOrAfter` from the `AuthnStatement`.
- ***get_errors*** Returns a list with code errors if something went wrong.
- ***get_last_error_reason*** Returns the reason of the last error
- ***get_sso_url*** Gets the SSO url.
- ***get_slo_url*** Gets the SLO url.
- ***get_last_request_id*** The `ID` of the last Request SAML message generated (`AuthNRequest`, `LogoutRequest`).
- ***get_last_authn_contexts*** Returns the list of authentication contexts sent in the last SAML Response.
- ***build_request_signature*** Builds the Signature of the SAML Request.
- ***build_response_signature*** Builds the Signature of the SAML Response.
- ***get_settings*** Returns the settings info.
- ***set_strict*** Set the strict mode active/disable.
- ***get_last_request_xml*** Returns the most recently-constructed/processed XML SAML request (AuthNRequest, LogoutRequest)
- ***get_last_response_xml*** Returns the most recently-constructed/processed XML SAML response (`SAMLResponse`, `LogoutResponse`). If the SAML-Response had an encrypted assertion, decrypts it.
- ***get_last_message_id*** The `ID` of the last Response SAML message processed.
- ***get_last_assertion_id*** The `ID` of the last assertion processed.
- ***get_last_assertion_not_on_or_after*** The `NotOnOrAfter` value of the valid SubjectConfirmationData node (if any) of the last assertion processed (is only calculated with strict = true)

OneLogin_Saml2_Auth - authn_request.py SAML 2 Authentication Request class

- `__init__` This class handles an AuthNRequest. It builds an `AuthNRequest` object.

- ***get_request*** Returns unsigned AuthnRequest.
- ***get_id*** Returns the AuthNRequest ID.
- ***get_xml*** Returns the XML that will be sent as part of the request.

OneLogin_Saml2_Response - response.py SAML 2 Authentication Response class

- `__init__` Constructs the SAML Response object.
- ***is_valid*** Determines if the SAML Response is valid. Includes checking of the signature by a certificate.
- ***check_status*** Check if the status of the response is success or not
- ***get_audiences*** Gets the audiences
- ***get_issuers*** Gets the issuers (from message and from assertion)
- ***get_nameid_data*** Gets the `NameID` Data provided by the SAML Response from the IdP (returns a dict)
- ***get_nameid*** Gets the NameID provided by the SAML Response from the IdP (returns a string)
- ***get_session_not_on_or_after*** Gets the SessionNotOnOrAfter from the AuthnStatement
- ***get_session_index*** Gets the `SessionIndex` from the `AuthnStatement`
- ***get_attributes*** Gets the `Attributes` from the `AttributeStatement` element.
- ***validate_num_assertions*** Verifies that the document only contains a single Assertion (encrypted or not)
- ***validate_timestamps*** Verifies that the document is valid according to `Conditions` element
- ***get_error*** After execute a validation process, if fails this method returns the cause
- ***get_xml_document*** Returns the SAML Response document (If contains an encrypted assertion, decrypts it).
- ***get_id*** the ID of the response
- ***get_assertion_id*** the `ID` of the assertion in the response
- ***get_assertion_not_on_or_after*** the `NotOnOrAfter` value of the valid SubjectConfirmationData if any

OneLogin_Saml2_LogoutRequest - logout_request.py SAML 2 Logout Request class

- `__init__` Constructs the Logout Request object.
- ***get_request*** Returns the Logout Request defated, base64-encoded.
- ***get_id*** Returns the ID of the Logout Request. (If you have the object you can access to the id attribute)
- ***get_nameid_data*** Gets the NameID Data of the the Logout Request (returns a dict).
- ***get_nameid*** Gets the NameID of the Logout Request Message (returns a string).

- ***get_issuer*** Gets the Issuer of the Logout Request Message.
- ***get_session_indexes*** Gets the `SessionIndexes` from the Logout Request.
- ***is_valid*** Checks if the Logout Request recieved is valid.
- ***get_error*** After execute a validation process, if fails this method returns the cause.
- ***get_xml*** Returns the XML that will be sent as part of the request or that was received at the SP

OneLogin_Saml2_LogoutResponse - logout_response.py SAML 2 Logout Response class

- `__init__` Constructs a Logout Response object.
- ***get_issuer*** Gets the Issuer of the Logout Response Message
- ***get_status*** Gets the Status of the Logout Response.
- ***is_valid*** Determines if the SAML LogoutResponse is valid
- ***build*** Creates a Logout Response object.
- ***get_response*** Returns a Logout Response object.
- ***get_error*** After execute a validation process, if fails this method returns the cause.
- ***get_xml*** Returns the XML that will be sent as part of the response or that was received at the SP

OneLogin_Saml2_Settings - settings.py Configuration of the SAML Python Toolkit

- `__init__` Initializes the settings: Sets the paths of the different folders and Loads settings info from settings file or array/object provided.
- ***check_settings*** Checks the settings info.
- ***check_idp_settings*** Checks the IdP settings info.
- ***check_sp_settings*** Checks the SP settings info.
- ***get_errors*** Returns an array with the errors, the array is empty when the settings is ok.
- ***get_sp_metadata*** Gets the SP metadata. The XML representation.
- ***validate_metadata*** Validates an XML SP Metadata.
- ***get_base_path*** Returns base path.
- ***get_cert_path*** Returns cert path.
- ***get_lib_path*** Returns lib path.
- ***get_ext_lib_path*** Returns external lib path.
- ***get_schemas_path*** Returns schema path.
- ***check_sp_certs*** Checks if the X.509 certs of the SP exists and are valid.
- ***get_sp_key*** Returns the X.509 private key of the SP.
- ***get_sp_cert*** Returns the X.509 public cert of the SP.
- ***get_sp_cert_new*** Returns the future X.509 public cert of the SP.
- ***get_idp_cert*** Returns the X.509 public cert of the IdP.
- ***get_sp_data*** Gets the SP data.

- ***get_idp_data*** Gets the IdP data.
- ***get_security_data*** Gets security data.
- ***get_contacts*** Gets contacts data.
- ***get_organization*** Gets organization data.
- ***format_idp_cert*** Formats the IdP cert.
- ***format_idp_cert_multi*** Formats all registered IdP certs.
- ***format_sp_cert*** Formats the SP cert.
- ***format_sp_cert_new*** Formats the SP cert new.
- ***format_sp_key*** Formats the private key.
- ***set_strict*** Activates or deactivates the strict mode.
- ***is_strict*** Returns if the `strict` mode is active.
- ***is_debug_active*** Returns if the debug is active.

OneLogin_Saml2_Metadata - metadata.py A class that contains functionality related to the metadata of the SP

- ***builder*** Generates the metadata of the SP based on the settings.
- ***sign_metadata*** Signs the metadata with the key/cert provided.
- ***add_x509_key_descriptors*** Adds the X.509 descriptors (sign/encriptation) to the metadata

OneLogin_Saml2_Utils - utils.py Auxiliary class that contains several methods

- ***decode_base64_and_inflate*** Base64 decodes and then inflates according to RFC1951.
- ***deflate_and_base64_encode*** Deflates and the base64 encodes a string.
- ***validate_xml*** Validates a xml against a schema.
- ***format_cert*** Returns a X.509 cert (adding header & footer if required).
- ***format_private_key*** Returns a private key (adding header & footer if required).
- ***redirect*** Executes a redirection to the provided url (or return the target url).
- ***get_self_url_host*** Returns the protocol + the current host + the port (if different than common ports).
- ***get_self_host*** Returns the current host.
- ***is_https*** Checks if https or http.
- ***get_self_url_no_query*** Returns the URL of the current host + current view.
- ***get_self_routed_url_no_query*** Returns the routed URL of the current host + current view.
- ***get_self_url*** Returns the URL of the current host + current view + query.
- ***generate_unique_id*** Generates an unique string (used for example as ID for assertions).

- ***parse_time_to_SAML*** Converts a UNIX timestamp to SAML2 timestamp on the form yyyy-mm-ddThh:mm:ss(.s+)?Z.
- ***parse_SAML_to_time*** Converts a SAML2 timestamp on the form yyyy-mm-ddThh:mm:ss(.s+)?Z to a UNIX timestamp.
- ***now*** Returns unix timestamp of actual time.
- ***parse_duration*** Interprets a ISO8601 duration value relative to a given timestamp.
- ***get_expire_time*** Compares 2 dates and returns the earliest.
- ***query*** Extracts nodes that match the query from the Element.
- ***delete_local_session*** Deletes the local session.
- ***calculate_x509_fingerprint*** Calculates the fingerprint of a X.509 cert.
- ***format_finger_print*** Formates a fingerprint.
- ***generate_name_id*** Generates a nameID.
- ***get_status*** Gets Status from a Response.
- ***decrypt_element*** Decrypts an encrypted element.
- ***write_temp_file*** Writes some content into a temporary file and returns it.
- ***add_sign*** Adds signature key and senders certificate to an element (Message or Assertion).
- ***validate_sign*** Validates a signature (Message or Assertion).
- ***validate_binary_sign*** Validates signed bynary data (Used to validate GET Signature).
- ***def get_encoded_parameter*** Return an url encoded get parameter value
- ***extract_raw_query_parameter***

OneLogin_Saml2_IdPMetadataParser - idp_metadata_parser.py
A class that contains methods to obtain and parse metadata from IdP

- ***get_metadata*** Get the metadata XML from the provided URL
- ***parse_remote*** Get the metadata XML from the provided URL and parse it, returning a dict with extracted data
- ***parse*** Parse the Identity Provider metadata and returns a dict with extracted data
- ***merge_settings*** Will update the settings with the provided new settings data extracted from the IdP metadata

For more info, look at the source code. Each method is documented and details about what does and how to use it are provided. Make sure to also check the doc folder where HTML documentation about the classes and methods is provided.

Demos included in the toolkit

The toolkit includes 4 demos to teach how use the toolkit (Django, Flask, Pyramid and Bootle projects), take a look on them. Demos require that SP and IdP are well configured before test it, so edit the settings files.

Notice that each python framework has it own way to handle routes/urls and process request, so focus on how it deployed. New demos using other python frameworks are welcome as a contribution.

Getting Started

We said that this toolkit includes a demos, lets see how fast is deploy some of them.

*** Virtualenv ***

The use of a virtualenv is highly recommended.

Virtualenv helps isolating the python enviroment used to run the toolkit. You can find more details and an installation guide in the official documentation.

Once you have your virtualenv ready and loaded, then you can install the toolkit on it in development mode executing this:

```
python setup.py develop
```

Using this method of deployment the toolkit files will be linked instead of copied, so if you make changes on them you won't need to reinstall the toolkit.

If you want install it in a normal mode, execute:

```
python setup.py install
```

Demo Flask

You'll need a virtualenv with the toolkit installed on it.

To run the demo you need to install the requirements first. Load your virtualenv and execute:

```
pip install -r demo-flask/requirements.txt
```

This will install flask and its dependences. Once it has finished, you have to complete the configuration of the toolkit. You'll find it at `demo-flask/settings.json`

Now, with the virtualenv loaded, you can run the demo like this:

```
cd demo-flask
python index.py
```

You'll have the demo running at `http://localhost:8000`

Content The flask project contains:

- ***index.py*** Is the main flask file, where or the SAML handle take place.

- ***templates***. Is the folder where flask stores the templates of the project. It was implemented a base.html template that is extended by `index.html` and `attrs.html`, the templates of our simple demo that shows messages, user attributes when available and login and logout links.
- ***saml*** Is a folder that contains the 'certs' folder that could be used to store the X.509 public and private key, and the saml toolkit settings (`settings.json` and `advanced_settings.json`).

SP setup The SAML Python Toolkit allows you to provide the settings info in 2 ways: Settings files or define a setting dict. In the `demo-flask`, it uses the first method.

In the index.py file we define the `app.config['SAML_PATH']`, that will target to the `saml` folder. We require it in order to load the settings files.

First we need to edit the `saml/settings.json`, configure the SP part and review the metadata of the IdP and complete the IdP info. Later edit the saml/advanced_settings.json files and configure the how the toolkit will work. Check the settings section of this document if you have any doubt.

IdP setup Once the SP is configured, the metadata of the SP is published at the `/metadata` url. Based on that info, configure the IdP.

How it works

1. First time you access to the main view `http://localhost:8000`, you can select to login and return to the same view or login and be redirected to `/?attrs` (attrs view).
2. When you click:

   2.1 in the first link, we access to `/?sso` (index view). An `AuthNRequest` is sent to the IdP, we authenticate at the IdP and then a Response is sent through the user's client to the SP, specifically the Assertion Consumer Service view: /?acs. Notice that a RelayState parameter is set to the url that initiated the process, the index view.

   2.2 in the second link we access to `/?attrs` (attrs view), we will expetience have the same process described at 2.1 with the diference that as `RelayState` is set the attrs url.
3. The `SAMLResponse` is processed in the ACS `/?acs`, if the Response is not valid, the process stops here and a message is shown. Otherwise we are redirected to the `RelayState` view. a) / or b) `/?attrs`
4. We are logged in the app and the user attributes are showed. At this point, we can test the single log out functionality.

The single log out funcionality could be tested by 2 ways.

```
5.1 SLO Initiated by SP. Click on the ``logout`` link at the SP, after that a Logout Request
```

```
5.2 SLO Initiated by IdP. In this case, the action takes place on the IdP side, the logout p
```

Notice that all the SAML Requests and Responses are handled at a unique view (index) and how GET paramters are used to know the action that must be done.

Demo Django

You'll need a virtualenv with the toolkit installed on it.

To run the demo you need to install the requirements first. Load your virtualenv and execute:

```
pip install -r demo-django/requirements.txt
```

This will install django and its dependences. Once it has finished, you have to complete the configuration of the toolkit.

Later, with the virtualenv loaded, you can run the demo like this:

```
cd demo-django
python manage.py runserver 0.0.0.0:8000
```

You'll have the demo running at `http://localhost:8000`.

Note that many of the configuration files expect HTTPS. This is not required by the demo, as replacing these SP URLs with HTTP will work just fine. HTTPS is however highly encouraged, and left as an exercise for the reader for their specific needs.

If you want to integrate a production django application, take a look on this SAMLServiceProviderBackend that uses our toolkit to add SAML support: https://github.com/KristianOellegaard/django-saml-service-provider

Content The django project contains:

- ***manage.py***. A file that is automatically created in each Django project. Is a thin wrapper around `django-admin.py` that takes care of putting the project's package on `sys.path` and sets the `DJANGO_SETTINGS_MODULE` environment variable.
- ***saml*** Is a folder that contains the `certs` folder that could be used to store the X.509 public and private key, and the saml toolkit settings (`settings.json` and `advanced_settings.json`).
- ***demo*** Is the main folder of the django project, that contains the typical files:
  - ***settings.py*** Contains the default parameters of a django project except the SAML_FOLDER parameter, that may contain the path where is located the 'saml' folder.

  - ***urls.py*** A file that define url routes. In the demo we defined `/` that is related to the index view, `/attrs` that is related with the attrs view and `/metadata`, related to th metadata view.
  - ***views.py*** This file contains the views of the django project and some aux methods.
  - ***wsgi.py*** A file that let as deploy django using WSGI, the Python standard for web servers and applications.

- ***templates***. Is the folder where django stores the templates of the project. It was implemented a base.html template that is extended by `index.html` and `attrs.html`, the templates of our simple demo that shows messages, user attributes when available and login and logout links.

SP setup The SAML Python Toolkit allows you to provide the settings info in 2 ways: Settings files or define a setting dict. In the `demo-django`, it uses the first method.

After set the `SAML_FOLDER` in the `demo/settings.py`, the settings of the python toolkit will be loaded on the django web.

First we need to edit the `saml/settings.json`, configure the SP part and review the metadata of the IdP and complete the IdP info. Later edit the saml/advanced_settings.json files and configure the how the toolkit will work. Check the settings section of this document if you have any doubt.

IdP setup Once the SP is configured, the metadata of the SP is published at the `/metadata` url. Based on that info, configure the IdP.

How it works This demo works very similar to the `flask-demo` (We did it intentionally).

Demo Pyramid

Unlike the other two projects, you don't need a pre-existing virtualenv to get up and running here, since Pyramid comes from the buildout school of thought.

To run the demo you need to install Pyramid, the requirements, etc.:

```
cd demo_pyramid
python -m venv env
env/bin/pip install --upgrade pip setuptools
env/bin/pip install -e ".[testing]"
```

Next, edit the settings in `demo_pyramid/saml/settings.json`. (Pyramid runs on port 6543 by default.)

Now you can run the demo like this:

```
env/bin/pserve development.ini
```

If that worked, the demo is now running at `http://localhost:6543`.

Content The Pyramid project contains:

- ***___init___.py*** is the main Pyramid file that configures the app and its routes.
- ***views.py*** is where all the SAML handling takes place.
- ***templates*** is the folder where Pyramid stores the templates of the project. It was implemented a `layout.jinja2` template that is extended by `index.jinja2` and `attrs.jinja2`, the templates of our simple demo that shows messages, user attributes when available and login and logout links.
- ***saml*** is a folder that contains the 'certs' folder that could be used to store the X.509 public and private key, and the saml toolkit settings (`settings.json` and `advanced_settings.json`).

SP setup The Python Toolkit allows you to provide the settings info in 2 ways: Settings files or define a setting dict. In `demo_pyramid` the first method is used.

In the views.py file we define the `SAML_PATH`, which will target the `saml` folder. We require it in order to load the settings files.

First we need to edit the `saml/settings.json`, configure the SP part and review the metadata of the IdP and complete the IdP info. Later edit the `saml/advanced_settings.json` files and configure the how the toolkit will work. Check the settings section of this document if you have any doubt.

IdP setup Once the SP is configured, the metadata of the SP is published at the `/metadata/` url. Based on that info, configure the IdP.

How it works

1. First time you access to the main view `http://localhost:6543`, you can select to login and return to the same view or login and be redirected to `/?attrs` (attrs view).
2. When you click:

   2.1 in the first link, we access to `/?sso` (index view). An `AuthNRequest` is sent to the IdP, we authenticate at the IdP and then a Response is sent through the user's client to the SP, specifically the Assertion Consumer Service view: `/?acs`. Notice that a RelayState parameter is set to the url that initiated the process, the index view.

   2.2 in the second link we access to `/?attrs` (attrs view), we will expetience have the same process described at 2.1 with the diference that as `RelayState` is set the attrs url.

3. The SAML Response is processed in the ACS `/?acs`, if the Response is not valid, the process stops here and a message is shown. Otherwise we are redirected to the `RelayState` view. a) / or b) `/?attrs`

4. We are logged in the app and the user attributes are showed. At this point, we can test the single log out functionality.

The single log out funcionality could be tested by 2 ways.

```
5.1 SLO Initiated by SP. Click on the ``logout`` link at the SP, after that a Logout Request

5.2 SLO Initiated by IdP. In this case, the action takes place on the IdP side, the logout p
```

Notice that all the SAML Requests and Responses are handled at a unique view (index) and how GET parameters are used to know the action that must be done.