Flask SAML2 Documentation

Release 0.3.0

Tidetech

Nov 26, 2021

CONTENTS:

Flask SAML2 helps you build Identity Providers and Service Providers (SP) for your applications. Each application will likely be either an Identity Provider (IdP) or a Service Provider (SP), so follow along with the relevant sections of the documentation for the particular needs of your application.

CHAPTER

ONE

INSTALLING

Install `flask-saml2` using pip:

```
$ pip install flask-saml2
```

1.1 Dependencies

`flask-saml2` relies on some libraries that have external dependencies. These external dependencies must be installed before `flask_saml2` and it's dependencies can be installed.

1.1.1 OpenSSL

`flask-saml2` relies on the pyopenssl library, which requires the `openssl` library to be installed. Please consult the documentation on installing pyopenssl for installation requirements.

1.1.2 lxml

`flask-saml2` relies on lxml. Please consult the Installing lxml and install all of the external dependencies for `lxml` before installing `flask-saml2`.

CHAPTER

TWO

EXAMPLES

The `flask_saml2` repository comes with an example implementation of an Identity Provider and a Service Provider, configured to work with one another.

To run the example implementation, clone the `flask_saml2` repository and follow the instructions in the README.

The example uses a hard coded list of users in the Identity Provider. A real implementation would most likely use an external user database, with authentication perhaps managed by Flask-Login.

CHAPTER

THREE

IDENTITY PROVIDERS

When users need to authenticate themselves with a Service Provider (SP), the SP will redirect the user to an Identity Provider (IdP). The users will authenticate with the Identity Provider, and will be redirected back to the Service Provider with a payload that identifies the user.

Flask SAML2 implements all parts of the IdP workflow, except for authenticating your users against your user database (or however your users are managed). Developers should create an *IdentityProvider* subclass for their application that integrates with some other form of authentication, such as Flask-Login. Once a user is authenticated with the IdP, relevant user details will be composed into a payload which will be sent via the users browser back to the SP.

The method *IdentityProvider.create_blueprint()* generates a Flask `Blueprint`, which needs to be registered in your application via `app.register_blueprint(idp.create_blueprint())`.

Any Service Providers the IdP handles need to be registered as well. These will be instances of *SPHandler*.

An functional example IdP and Flask application that uses a static list of users can be found in the `examples/` directory of the repository.

3.1 IdentityProvider

class flask_saml2.idp.**IdentityProvider**

Developers should subclass *IdentityProvider* and provide methods to interoperate with their specific environment. All user interactions are performed through methods on this class.

Every subclass should implement *is_user_logged_in()*, *login_required()*, *logout()*, and *get_current_user()* as a minimum. Other methods can be overridden as required.

idp_digester_class

alias of `flask_saml2.signing.Sha1Digester`

idp_signer_class

alias of `flask_saml2.signing.RsaSha1Signer`

get_idp_config()

Get the configuration for this IdP. Defaults to `SAML2_IDP` from `flask.Flask.config`. The configuration should be a dict like:

```
{
    # Should the IdP automatically redirect the user back to the
    # Service Provider once authenticated.
    'autosubmit': True,
    # The X509 certificate and private key this IdP uses to
    # encrypt, validate, and sign payloads.
    'certificate': ...,
```

(continues on next page)

(continued from previous page)

```
    'private_key': ...,
}
```

To load the `certificate` and `private_key` values, see

- *certificate_from_string()*
- *certificate_from_file()*
- *private_key_from_string()*
- *private_key_from_file()*

Return type dict

get_idp_entity_id()
The unique identifier for this Identity Provider. By default, this uses the metadata URL for this IdP.

See *get_metadata_url()*.

Return type str

get_idp_certificate()
Get the public certificate for this IdP. If this IdP does not sign its requests, returns None.

Return type Optional[X509]

get_idp_private_key()
Get the private key for this IdP. If this IdP does not sign its requests, returns None.

Return type Optional[PKey]

get_idp_autosubmit()
Should the IdP autosubmit responses to the Service Provider?

Return type bool

get_idp_signer()
Get the signing algorithm used by this IdP.

Return type Optional[*Signer*]

get_idp_digester()
Get the method used to compute digests for the IdP.

Return type *Digester*

get_service_providers()
Get an iterable of service provider `config` dicts. `config` should be a dict specifying a SPHandler subclass and optionally any constructor arguments:

```
>>> list(idp.get_service_providers())
[{
    'CLASS': 'my_app.service_providers.MySPSPHandler',
    'OPTIONS': {
        'acs_url': 'https://service.example.com/auth/acs/',
    },
}]
```

Defaults to `current_app.config['SAML2_SERVICE_PROVIDERS']`.

Return type Iterable[Tuple[str, dict]]

get_sso_url()
Get the URL for the Single Sign On endpoint for this IdP.

get_slo_url()
Get the URL for the Single Log Out endpoint for this IdP.

get_metadata_url()
Get the URL for the metadata XML document for this IdP.

login_required()
Check if a user is currently logged in to this session, and `flask.abort()` with a redirect to the login page if not. It is suggested to use *is_user_logged_in()*.

is_user_logged_in()
Return True if a user is currently logged in. Subclasses should implement this method

Return type bool

logout()
Terminate the session for a logged in user. Subclasses should implement this method.

get_current_user()
Get the user that is currently logged in.

Return type ~User

get_user_nameid(*user*, *attribute*)
Get the requested name or identifier from the user. `attribute` will be a `urn:oasis:names:tc:SAML:2.0:nameid-format`-style urn.

Subclasses can override this to allow more attributes to be extracted. By default, only email addresses are extracted using *get_user_email()*.

get_user_email(*user*)
Get the email address for a user.

get_sp_handlers()
Get the SPHandler for each service provider defined.

Return type Iterable[SPHandler]

render_template(*template*, ***context*)
Render an HTML template. This method can be overridden to inject more context variables if required.

Return type str

get_metadata_context()
Get any extra context for the metadata template. Suggested extra context variables include 'org' and 'contacts'.

Return type dict

is_valid_redirect(*url*)
Check if a URL is a valid and safe URL to redirect to, according to any of the SPHandlers. Only used from the non-standard logout page, for non-compliant Service Providers such as Salesforce.

Return type bool

create_blueprint()
Create a blueprint for this IdP. This blueprint needs to be registered with a Flask application to expose the IdP functionality.

3.2 SPHandler

An *flask_saml2.idp.IdentityProvider* handles requests from Service Providers via *flask_saml2.idp.SPHandler* instances.

See *Configuration* for configuration options.

class flask_saml2.idp.**SPHandler**(*idp*, ***, *entity_id*, *acs_url=None*, *certificate=None*, *display_name=None*)

Handles incoming SAML requests from a specific Service Provider for a running Identity Provider.

Sub-classes should provide Service Provider-specific functionality.

assertion_template
alias of flask_saml2.idp.xml_templates.AssertionTemplate

response_template
alias of flask_saml2.idp.xml_templates.ResponseTemplate

get_sp_signer()
Get the *Signer* to use for this SP. Default to the one used by the IdP. If a particular SP requires a particular signing method, that SP can override it.

Return type *Signer*

get_sp_digester()
Get the *Digester* to use for this SP. Default to the one used by the IdP. If a particular SP requires a particular digest method, that SP can override it.

Return type *Digester*

build_assertion(*request*, *issue_instant*)
Build parameters for the assertion template.

Return type dict

build_response(*request*, *issue_instant*)
Build parameters for the response template.

Return type dict

encode_response(*response*)
Encodes the response XML template suitable for sending to the SP.

format_assertion(*assertion_params*)
Make a AssertionTemplate to respond to this SP.

Return type *XmlTemplate*

format_response(*response_params*, *assertion*)
Make a ResponseTemplate to respond to this SP.

Return type *XmlTemplate*

get_assertion_id()
Generates an ID for this assertion.

get_audience(*request*)
Gets the audience assertion parameter from the request data.

Return type str

get_response_id()
Generate an ID for the response.

get_response_context(*request*, *response*, *relay_state*)
Make a dictionary of parameters for the response template.

get_subject()
Get the subject of the assertion, based on the currently authenticated user and `SPHandler.subject_format`.

extract_request_parameters(*request*)
Fetches various parameters from the request into a dict.

Return type `dict`

validate_request(*request*)
Validates the SAML request against the configuration of this Service Provider handler . Sub-classes should override this and raise a *CannotHandleAssertion* exception if the validation fails.

Raises:

CannotHandleAssertion: if the ACS URL specified in the SAML request doesn't match the one specified in the SP handler config.

validate_destination(*request*)
Validate an `<AuthnRequest>` Destination attribute, if it is set.

validate_entity_id(*request*)
Validate that the `<AuthnRequest>` Issuer attribute matches this Service Provider.

validate_acs_url(*request*)
Validate that the `<AuthnRequest>` AssertionConsumerServiceURL attribute matches the expected ACS URL for this Service Provider.

validate_user()
Validates the User. Sub-classes should override this and throw a CannotHandleAssertion exception if the validation does not succeed.

decode_saml_string(*saml_string*)
Decode an incoming SAMLRequest into an XML string.

Return type `bytes`

parse_authn_request(*saml_request*)
Get a `AuthnRequestParser` to handle this request.

Return type `AuthnRequestParser`

parse_logout_request(*saml_request*)
Get a `LogoutRequestParser` to handle this request.

Return type `LogoutRequestParser`

make_response(*request*)
Process the request and make a `ResponseTemplate`.

Return type *`XmlTemplate`*

is_valid_redirect(*url*)
Is this URL a valid redirect target back to this service provider?

format_datetime(*value*)
Format a datetime for this SP. Some SPs are picky about their date formatting, and don't support the format produced by `datetime.datetime.isoformat()`.

Return type `str`

3.2.1 Specific implementations

Some handlers for common Service Providers have been bundled with this project:

class flask_saml2.idp.sp.salesforce.**SalesforceSPHandler**(*idp*, ***, *entity_id*, *acs_url=None*, *certificate=None*, *display_name=None*)

Salesforce.com `SPHandler` implementation.

class flask_saml2.idp.sp.google_apps.**GoogleAppsSPHandler**(*idp*, ***, *entity_id*, *acs_url=None*, *certificate=None*, *display_name=None*)

Google Apps `SPHandler` implementation.

class flask_saml2.idp.sp.dropbox.**DropboxSPHandler**(*idp*, ***, *entity_id*, *acs_url=None*, *certificate=None*, *display_name=None*)

Dropbox `SPHandler` implementation.

3.3 Configuration

The IdP needs two configuration options by default, `SAML2_IDP` and `SAML2_SERVICE_PROVIDERS`. `SAML2_IDP` configures the IdP itself, while `SAML2_SERVICE_PROVIDERS` specifies all the SPs this IdP supports.

```
from flask_saml2.utils import certificate_from_file, private_key_from_file

SAML2_IDP = {
    'autosubmit': True,
    'certificate': certificate_from_file('keys/idp_certificate.pem'),
    'private_key': private_key_from_file('keys/idp_private_key.pem'),
}

SAML2_SERVICE_PROVIDERS = [
    {
        'CLASS': 'myapp.SPHandler',
        'OPTIONS': {
            'display_name': 'Example Service Provider',
            'entity_id': 'http://service.example.com/saml/metadata.xml',
            'acs_url': 'http://service.example.com/saml/acs/',
            'certificate': certificate_from_file('keys/example_sp_certificate.pem'),
        },
    },
]
```

`SAML2_IDP` is documented in *IdentityProvider.get_idp_config()*.

`SAML2_SERVICE_PROVIDERS` is a list of SPs the IdP will authenticate users for. Each SP is represented as a dict. `CLASS` is the dotted Python path to a *SPHandler* subclass, and `OPTIONS` is a dict of keyword arguments to its constructor. Refer to *SPHandler* for more information on constructor arguments.

CHAPTER

FOUR

SERVICE PROVIDERS

A Service Provider (SP) is a website that users visit, that uses a separate Identity Provider (IdP) to authenticate users.

Flask SAML2 provides all of the functionality required to implement your own SP that can authenticate using one or more external IdPs. These IdPs can be written using *flask_saml2.idp*, or come from external providers.

The method *ServiceProvider.create_blueprint()* generates a Flask `Blueprint`, which needs to be registered in your application via `app.register_blueprint(sp.create_blueprint())`.

Any Identity Providers the SP can authenticate with need to be registered as well. These will be instances of *IdPHandler*.

An functional example SP and Flask application can be found in the `examples/` directory of the repository.

4.1 ServiceProvider

class flask_saml2.sp.**ServiceProvider**

Developers should subclass *ServiceProvider* and provide methods to interoperate with their specific environment. All user interactions are performed through methods on this class.

There are no methods that must be overridden, but overriding *get_default_login_return_url()* and *get_logout_return_url()* is recommended.

session_auth_data_key = 'saml_auth_data'

What key to store authentication details under in the session.

blueprint_name = 'flask_saml2_sp'

The name of the blueprint to generate.

login_successful(*auth_data*, *relay_state*)

Called when a user is successfully logged on. Subclasses should override this if they want to do more with the returned user data. Returns a `flask.Response`, which is usually a redirect to *get_default_login_return_url()*, or a redirect to the protected resource the user initially requested. Subclasses may override this method and return a different response, but they *must* call `super()`.

Return type `Response`

get_sp_config()

Get the configuration for this SP. Defaults to `SAML2_SP` from `flask.Flask.config`. The configuration should be a dict like:

```
{
    # The X509 certificate and private key this SP uses to
    # encrypt, validate, and sign payloads.
```

(continues on next page)

(continued from previous page)

```
    'certificate': ...,
    'private_key': ...,
}
```

To load the `certificate` and `private_key` values, see

- *certificate_from_string()*
- *certificate_from_file()*
- *private_key_from_string()*
- *private_key_from_file()*

Return type `dict`

get_sp_entity_id()
The unique identifier for this Service Provider. By default, this uses the metadata URL for this SP.

See *get_metadata_url()*.

Return type `str`

get_sp_certificate()
Get the public certificate for this SP.

Return type `Optional`[`X509`]

get_sp_private_key()
Get the private key for this SP.

Return type `Optional`[`PKey`]

get_sp_signer()
Get the signing algorithm used by this SP.

Return type `Optional`[*Signer*]

get_sp_digester()
Get the digest algorithm used by this SP.

Return type *Digester*

should_sign_requests()
Should this SP sign its SAML statements. Defaults to True if the SP is configured with both a certificate and a private key.

Return type `bool`

get_identity_providers()
Get an iterable of identity provider `config` dicts.‘‘config‘‘ should be a dict specifying an IdPHandler subclass and optionally any constructor arguments:

```
>>> list(sp.get_identity_providers())
[{
    'CLASS': 'my_app.identity_providers.MyIdPIdPHandler',
    'OPTIONS': {
        'entity_id': 'https://idp.example.com/metadata.xml',
    },
}]
```

Defaults to `current_app.config['SAML2_IDENTITY_PROVIDERS']`.

Return type Iterable[Tuple[str, dict]]

get_login_url()
The URL of the endpoint that starts the login process.

Return type str

get_acs_url()
The URL for the Assertion Consumer Service for this SP.

Return type str

get_sls_url()
The URL for the Single Logout Service for this SP.

Return type str

get_metadata_url()
The URL for the metadata xml for this SP.

Return type str

get_default_login_return_url()
The default URL to redirect users to once the have logged in.

Return type Optional[str]

get_login_return_url()
Get the URL to redirect the user to now that they have logged in.

Return type Optional[str]

get_logout_return_url()
The URL to redirect users to once they have logged out.

Return type Optional[str]

is_valid_redirect_url(*url*)
Is this URL valid and safe to redirect to? Defaults to only allowing URLs on the current server.

Return type str

make_idp_handler(*config*)
Construct an *IdPHandler* from a config dict from *get_identity_providers()*.

Return type IdPHandler

get_idp_handlers()
Get the *IdPHandler* for each service provider defined.

Return type Iterable[IdPHandler]

get_default_idp_handler()
Get the default IdP to sign in with. When logging in, if there is a default IdP, the user will be automatically logged in with that IdP. Return None if there is no default IdP. If there is no default, a list of IdPs to sign in with will be presented by the login view.

Return type Optional[IdPHandler]

get_idp_handler_by_entity_id(*entity_id*)
Find the *IdPHandler* instance with a matching entity ID.

Return type IdPHandler

get_idp_handler_by_current_session()
Get the *IdPHandler* used to authenticate the currently logged in user.

Return type IdPHandler

login_required()
Check if a user is currently logged in to this session, and flask.abort() with a redirect to the login page if not. It is suggested to use *is_user_logged_in()*.

is_user_logged_in()
Check if the user is currently logged in / authenticated with an IdP.

Return type bool

logout()
Terminate the session for a logged in user.

render_template(*template*, ***context*)
Render an HTML template. This method can be overridden to inject more context variables if required.

Return type str

set_auth_data_in_session(*auth_data*)
Store authentication details from the *IdPHandler* in the browser session.

clear_auth_data_in_session()
Clear the authentication details from the session. This will effectively log the user out.

get_auth_data_in_session()
Get an AuthData instance from the session data stored for the currently logged in user.

Return type AuthData

make_absolute_url(*url*)
Take a local URL and make it absolute by prepending the current SERVER_NAME.

Return type str

get_metadata_context()
Get any extra context for the metadata template. Suggested extra context variables include 'org' and 'contacts'.

Return type dict

create_blueprint()
Create a Flask flask.Blueprint for this Service Provider.

Return type Blueprint

4.2 IdPHandler

A flask_saml2.idp.ServiceProvider handles requests from Identity Providers via flask_saml2.idp.IdPHandler instances.

See *Configuration* for configuration options.

class flask_saml2.sp.**IdPHandler**(*sp*, *, *entity_id*, *display_name=None*, *sso_url=None*, *slo_url=None*, *certificate=None*, ***kwargs*)
Represents an Identity Provider that the running Service Provider knows about. This class should be subclassed for Identity Providers that need specific configurations.

get_idp_sso_url()
Get the Single Sign On URL for this IdP.

get_idp_slo_url()
Get the Single Log Out URL for this IdP.

get_sp_acs_url()
Get the Attribute Consumer Service URL on the current SP this IdP should send responses to.

get_authn_request(*template=<class 'flask_saml2.sp.xml_templates.AuthnRequest'>*, ***parameters*)
Make a AuthnRequest to send to this IdP.

get_logout_request(*auth_data*, *template=<class 'flask_saml2.sp.xml_templates.LogoutRequest'>*, ***parameters*)
Make a LogoutRequest for the authenticated user to send to this IdP.

make_login_request_url(*relay_state=None*)
Make a LoginRequest url and query string for this IdP.

Return type str

decode_saml_string(*saml_string*)
Decode an incoming SAMLResponse into an XML string.

Return type bytes

encode_saml_string(*saml_string*)
Encoding an XML string into a SAMLRequest.

Return type str

get_response_parser(*saml_response*)
Make a ResponseParser instance to handle this response.

get_auth_data(*response*)
Create an AuthData instance from a SAML Response. The response is validated first.

Return type AuthData

format_datetime(*value*)
Format a datetime for this IdP. Some IdPs are picky about their date formatting, and don't support the format produced by datetime.datetime.isoformat().

Return type str

4.3 Configuration

The SP needs two configuration options by default, SAML2_SP and SAML2_IDENTITY_PROVIDERS. SAML2_SP configures the Service Provider itself, while SAML2_IDENTITY_PROVIDERS specifies all the IdPs the SP can authenticate with.

```
from flask_saml2.utils import certificate_from_file, private_key_from_file

SAML2_SP = {
    'certificate': certificate_from_file('keys/sp_certificate.pem'),
    'private_key': private_key_from_file('keys/sp_private_key.pem'),
}

SAML2_IDENTITY_PROVIDERS = [
    {
        'CLASS': 'myapp.IdPHandler',
        'OPTIONS': {
```

(continues on next page)

(continued from previous page)

```
            'display_name': 'Example Identity Provider',
            'entity_id': 'https://idp.example.com/saml/metadata.xml',
            'sso_url': 'https://idp.example.com/saml/login/',
            'slo_url': 'https://idp.example.com/saml/logout/',
            'certificate': certificate_from_file('keys/idp_certificate.pem'),
        },
    },
]
```

`SAML2_SP` is documented in *`ServiceProvider.get_sp_config()`*.

`SAML2_IDENTITY_PROVIDERS` is a list of IdPs the SP can use for authentication. Each IdP is represented as a dict. `CLASS` is the dotted Python path to a *`IdPHandler`* subclass, and `OPTIONS` is a dict of keyword arguments to its constructor. Refer to *`IdPHandler`* for more information on constructor arguments.

4.4 Example

To make your application into a Service Provider, create a `ServiceProvider` subclass, instantiate it, and register it's `Blueprint` with your `Flask application`:

```
from flask import Flask
from flask_saml2.sp import ServiceProvider


class MyServiceProvider(ServiceProvider):
    def get_default_login_return_url(self):
        return url_for('dashboard')

    def get_logout_return_url(self):
        return url_for('index')

sp = ServiceProvider()

app = Flask()
app.register_blueprint(sp.create_blueprint(), url_prefix='/saml/')
app.run()
```

CHAPTER

FIVE

EXCEPTIONS

All the SAML-specific exceptions this library can throw.

exception flask_saml2.exceptions.**SAML2Exception**
Base exception for all flask_saml2 exceptions.

exception flask_saml2.exceptions.**MessageException**(*msg*)
An exception with a nicely formatted error message.

exception flask_saml2.exceptions.**CannotHandleAssertion**(*msg*)
This SP or IdP handler can not handle this assertion.

exception flask_saml2.exceptions.**UserNotAuthorized**(*msg*)
User not authorized for SAML 2.0 authentication.

exception flask_saml2.exceptions.**ImproperlyConfigured**(*msg*)
Someone done goofed when configuring this application.

CHAPTER

SIX

INTERNAL MODULES

These modules are used by the *IdentityProvider* and *ServiceProvider* classes. They may be useful to you if you are writing a custom handler to support a particular upstream IdP or downstream SP.

6.1 Encoding and decoding

Utilities to encode and decode zlib and base64 data.

flask_saml2.codex.**decode_base64_and_inflate**(*b64string*)
Turn a base64-encoded zlib-compressed blob back in to the original bytes. The opposite of *deflate_and_base64_encode()*.

Return type bytes

flask_saml2.codex.**deflate_and_base64_encode**(*string_val*)
zlib-compress and base64-encode some data. The opposite of *decode_base64_and_inflate()*.

Return type bytes

flask_saml2.codex.**decode_saml_xml**(*data*)
Decodes some base64-encoded and possibly zipped string into an XML string.

Return type bytes

6.2 Signing and digest tools

Functions and classes that deal with signing data and making digests.

class flask_saml2.signing.**Digester**
Base class for all the digest methods. SAML2 digest methods have an identifier in the form of a URL, and must produce a text digest.

Subclasses should set the *uri* attribute and provide a *make_digest()* method.

Implemented digest methods: Sha1Digester, Sha256Digester.

Example:

```
>>> from flask_saml2.signing import Sha1Digester
>>> digester = Sha1Digester()
>>> digester(b'Hello, world!')
'lDpwLQbzRZmu4fjajvn3KWAx1pk='
```

uri = None
The URI identifing this digest method

make_digest(*data*)
Make a binary digest of some binary data using this digest method.

Return type bytes

class flask_saml2.signing.**Signer**
Sign some data with a particular algorithm. Each Signer may take different constructor arguments, but each will have a uri attribute and will sign data when called.

Implemented signers: `RsaSha1Signer`.

Example:

```
>>> from flask_saml2.signing import RsaSha1Signer
>>> from flask_saml2.utils import private_key_from_file
>>> key = private_key_from_file('tests/keys/sample/idp-private-key.pem')
>>> signer = RsaSha1Signer(private_key)
>>> signer(b'Hello, world!')
'YplglOQDPLiozAWoY9ykgQ4eicojNnU+KjRrwGp67jHM5FGkQZ71Pk1Bgo631WA5B1hopQByRh/
↪elqtEEN+vRA=='
```

uri = None
The URI identifing this signing method

class flask_saml2.signing.**SignedInfoTemplate**(*params={}*)
A <SignedInfo> node, such as:

```
<ds:SignedInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#
↪"></ds:CanonicalizationMethod>
    <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"></
↪ds:SignatureMethod>
    <ds:Reference URI="#${REFERENCE_URI}">
        <ds:Transforms>
            <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-
↪signature"></ds:Transform>
            <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"></
↪ds:Transform>
        </ds:Transforms>
        <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"></
↪ds:DigestMethod>
        <ds:DigestValue>${SUBJECT_DIGEST}</ds:DigestValue>
    </ds:Reference>
</ds:SignedInfo>
```

generate_xml()
Generate the XML node for this template. Generally accessed through `xml`.

class flask_saml2.signing.**SignatureTemplate**(*params={}*)
A <Signature> node, such as:

```
<ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    ${SIGNED_INFO}
    <ds:SignatureValue>${RSA_SIGNATURE}</ds:SignatureValue>
    <ds:KeyInfo>
        <ds:X509Data>
            <ds:X509Certificate>${CERTIFICATE}</ds:X509Certificate>
```

(continues on next page)

(continued from previous page)

```
        </ds:X509Data>
    </ds:KeyInfo>
</ds:Signature>
```

classmethod sign(*subject*, *certificate*, *digester*, *signer*, *reference_uri*)
Create a *SignatureTemplate* by signing a `subject` string.

Parameters

- **subject** (`str`) – The string to sign. This is usually the canonical string representation of the XML node this `<Signature>` verifies.
- **certificate** (`X509`) – The certificate to sign the data with
- **digester** (*Digester*) – The algorithm used to make the digest
- **signer** (*Signer*) – The algorithm used to sign the data
- **reference_uri** (`str`) – The ID of the element that is signed

See also: *SignableTemplate.sign()*

generate_xml()
Generate the XML node for this template. Generally accessed through `xml`.

class flask_saml2.signing.**SignableTemplate**(*params={}*)
An `XmlTemplate` that supports being signed, by adding an *<Signauture>* element.

signature_index = 1
The element index where the signature should be inserted

id_parameter = None
The parameter that contains the element ID

See *get_id()* and *sign()*

sign(*certificate*, *digester*, *signer*)
Cryptographically sign this template by inserting a *<Signature>* element.

The ID of the node to sign is fetched from *get_id()*.

Parameters

- **certificate** (`X509`) – The certificate to sign the data with
- **digester** (*Digester*) – The algorithm used to make the digest
- **signer** (*Signer*) – The algorithm used to sign the data

Return type `ElementBase`

make_signature(*certificate*, *digester*, *signer*)
Create XML `<Signature>` node for the `subject` text.

Return type *SignatureTemplate*

add_signature(*signature*)
Insert a *<Signature>* into this node.

get_id()
Get the ID of the root node, required to *sign()* this node. By default, grabs the ID from the parameter named in *id_parameter*.

Return type `str`

flask_saml2.signing.**sign_query_parameters**(*signer*, *bits*)
Sign the bits of a query string.

```
>>> signer = ...  # A Signer instance
>>> bits = [('Foo', '1'), ('Bar', '2')]
>>> sign_query_parameters(signer, bits)
"Foo=1&Bar=2&SigAlg=...&Signature=..."
```

Return type str

6.3 XML tools

6.3.1 XML parsing

The *flask_saml2.xml_parser* provides tools for parsing XML documents from an IdP or a SP. If the documents are signed, they will be verified as part of parsing.

class flask_saml2.xml_parser.**XmlParser**(*xml_string*, *certificate*)
Parse a possibly-signed XML document. Subclasses must implement *is_signed()*.

certificate = None
The certificate the document is signed with

xml_string = None
The input XML document as a string

xml_tree = None
The parsed XML document

parse_request(*xml_string*)
Parse the SAML request. :raises: ValueError

Return type None

is_signed()
Is this request signed? Looks for a <ds:Signature> element. Different sources will generate different signed XML documents, so this method must be implemented differently for each source.

parse_signed(*xml_tree*, *certificate*)
Replaces all parameters with only the signed parameters. You should provide an x509 certificate obtained out-of-band, usually via the SAML metadata. Otherwise the signed data will be verified with only the certificate provided in the request. This is INSECURE and more-or-less only useful for testing.

Return type ElementBase

6.3.2 XML templates

class flask_saml2.xml_templates.**XmlTemplate**(*params={}*)
Base XML template class. A template can represent a single node, a tree, or a whole XML document.

namespace = None
XML namespace for this node or document

property xml
The XML node this template constructed. Generated using *generate_xml()*.

generate_xml()
Generate the XML node for this template. Generally accessed through *xml*.

Return type ElementBase

get_xml_string()
Render the XML node to a string. The string representation is rendered as canonical c14n XML, to make verification and signing possible.

Return type str

element(*tag*, ***, *namespace=None*, *attrs=None*, *children=None*, *text=None*)
Shortcut for creating an ElementTree Element, with optional attributes, children, and text.

Parameters

- **str** (*text*) – tag to give XML element
- **str** – Namespace to use for the element. Defaults to *get_namespace()* if None.
- **dict** (*attrs*) – Element attributes. If an attribute value is None, the attribute is ignored.
- **list** (*children*) – Element children. If an item in children is None, the item is ignored.
- **str** – Element text content, if any.

Return type ElementBase

Returns xml.etree.ElementTree.Element

get_namespace_map()
Get all the namespaces potentially used by this node, as a etree nsmap.

Return type Mapping[str, str]

get_namespace()
Get the namespace URI for this node. Looks up the namespace alias *namespace* in *get_namespace_map()*.

Return type str

class flask_saml2.xml_templates.**NameIDTemplate**(*params={}*)
A <NameID> node, such as:

```
<NameID Format="${SUBJECT_FORMAT}" SPNameQualifier="${SP_NAME_QUALIFIER}">
    ${SUBJECT}
</NameID>
```

generate_xml()
Generate the XML node for this template. Generally accessed through xml.

6.4 Utilities

class flask_saml2.utils.**cached_property**(*func*, *name=None*, *doc=None*)
A decorator that converts a function into a lazy property. The function wrapped is called the first time to retrieve the result and then that calculated result is used the next time you access the value:

```
class Foo(object):
    @cached_property
    def foo(self):
```

(continues on next page)

(continued from previous page)

```
        # calculate something important here
        return 42
```

The class has to have a `__dict__` in order for this property to work.

`flask_saml2.utils.`**`import_string`**(*path*)
Import a dotted Python path to a class or other module attribute. `import_string('foo.bar.MyClass')` will return the class `MyClass` from the package `foo.bar`.

Return type `Any`

`flask_saml2.utils.`**`get_random_id`**()
Generate a random ID string. The random ID will start with the '_' character.

Return type `str`

`flask_saml2.utils.`**`utcnow`**()
Get the current time in UTC, as an aware `datetime.datetime`.

Return type `datetime`

`flask_saml2.utils.`**`certificate_to_string`**(*certificate*)
Take an x509 certificate and encode it to a string suitable for adding to XML responses.

Parameters **`certificate`** (`X509`) – A certificate, perhaps loaded from *`certificate_from_file()`*.

Return type `str`

`flask_saml2.utils.`**`certificate_from_string`**(*certificate*, *format=1*)
Load an X509 certificate from a string. This just strips off the header and footer text.

Parameters

- **`str`** – A certificate string.
- **`format`** – The format of the certificate, from crypto — Generic cryptographic module.

Return type `X509`

`flask_saml2.utils.`**`certificate_from_file`**(*filename*, *format=1*)
Load an X509 certificate from `filename`.

Parameters

- **`filename`** (`Union`[`str`, `Path`]) – The path to the certificate on disk.
- **`format`** – The format of the certificate, from crypto — Generic cryptographic module.

Return type `X509`

`flask_saml2.utils.`**`private_key_from_string`**(*private_key*, *format=1*)
Load a private key from a string.

Parameters

- **`str`** – A private key string.
- **`format`** – The format of the private key, from crypto — Generic cryptographic module.

Return type `PKey`

`flask_saml2.utils.`**`private_key_from_file`**(*filename*, *format=1*)
Load a private key from `filename`.

Parameters

- **filename** (Union[str, Path]) – The path to the private key on disk.
- **format** – The format of the private key, from crypto — Generic cryptographic module.

Return type PKey

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

INDEX

G

I

L

M

N

P

R

S

U

V

X