Gateway guide

This guide shows you how to set up examples that use PingGateway. It is for access management designers and administrators who develop, build, deploy, and maintain PingGateway for their organizations.

This guide assumes familiarity with the following topics:

- Hypertext Transfer Protocol (HTTP), including how clients and servers exchange messages, and the role that a reverse proxy (gateway) plays
- JavaScript Object Notation (JSON), which is the format for PingGateway configuration files
- Managing services on operating systems and application servers
- Configuring network connections on operating systems
- Managing Public Key Infrastructure (PKI) used to establish HTTPS connections
- Access management for web applications

Depending on the features you use, you should also have basic familiarity with the following topics:

- Structured Query Language (SQL) if you use PingGateway with relational databases
- Configuring AM if you use password capture and replay, or if you plan to follow the OAuth 2.0 or SAML 2.0 tutorials
- The Groovy programming language if you plan to extend PingGateway with scripts
- The Java programming language if you plan to extend PingGateway with plugins, and Apache Maven for building plugins

Example installation for this guide

Unless otherwise stated, the examples in this guide assume the following installation:

- PingGateway accessible on `http://ig.example.com:8080` and `https://ig.example.com:8443`, as described in Quick install.
- Sample application installed on http://app.example.com:8081, as described in Use the sample application.
- AM installed on http://am.example.com:8088/openam, with the default configuration.

If you use a different configuration, substitute in the procedures accordingly.

Set up AM

This documentation contains procedures for setting up items in AM that you can use with PingGateway. You can find more information in the Access Management documentation.

Authenticate a PingGateway agent to AM

IMPORTANT

From AM 7.3

When AM 7.3 is installed with a default configuration, as described in Evaluation, PingGateway is automatically authenticated to AM by an authentication tree. Otherwise, PingGateway is authenticated to AM by an AM authentication module.

Authentication chains and modules were deprecated in AM 7. When they are removed in a future release of AM, it will be necessary to configure an appropriate authentication tree when you aren't using the default configuration.

You can find more information in AM documentation on Authentication nodes and trees.

This section describes how to create an authentication tree to authenticate a PingGateway agent to AM. The tree has the following requirements:

- It must be called `Agent`
- Its nodes must pass the agent credentials to the Agent Data Store Decision node.

When you define a tree in AM, that same tree is used for all instances of PingGateway, Java agent, and Web agent. Consider this point if you change the tree configuration.

1. On the **Realms** page of the AM admin UI, choose the realm in which to create the authentication tree.
2. On the **Realm Overview** page, click **Authentication** > **Trees** > **+ Create tree**.
3. Create a tree named `Agent`.

   The authentication tree designer is displayed, with the `Start` entry point connected to the `Failure` exit point and a `Success` node.

   The authentication tree designer provides the following features on the toolbar:

Button	Usage
⊞	Lay out and align nodes according to the order they are connected.
⛶	Toggle the designer window between normal and full-screen layout.
🗑	Remove the selected node. Note that the `Start` entry point cannot be deleted.

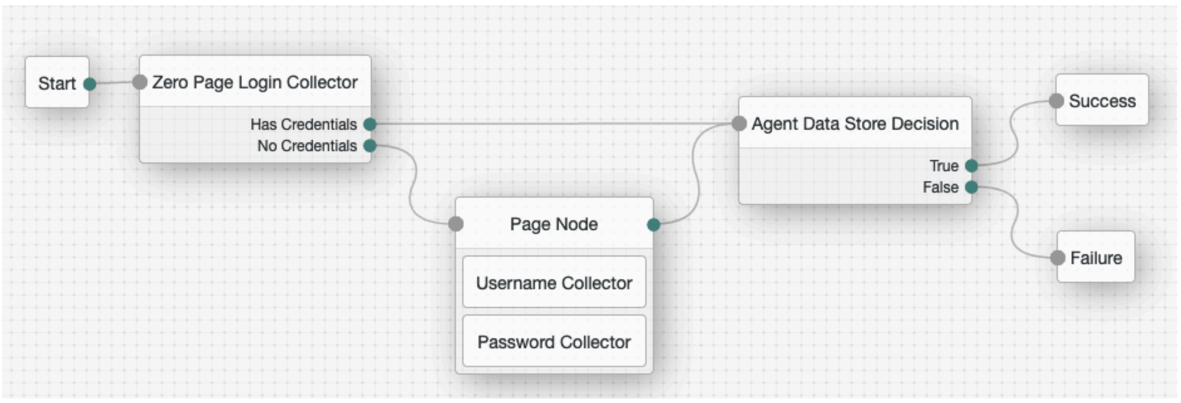
4. Using the 🔍 **Filter** bar, find and then drag the following nodes from the **Components** panel into the designer area:
    - Zero Page Login Collector node to check whether the agent credentials are provided in the incoming authentication request, and use their values in the following nodes.

      This node is required for compatibility with Java agent and Web agent.
    - Page node to collect the agent credentials if they aren't provided in the incoming authentication request, and use their values in the following nodes.
    - Agent Data Store Decision node to verify the agent credentials match the registered PingGateway agent profile.

IMPORTANT

Many nodes can be configured in the panel on the right side of the page. Unless otherwise stated, don't configure the nodes and use only the default values.

5. Drag the following nodes from the **Components** panel into the Page node:
    - Username Collector node to prompt the user to enter their username.
    - Password Collector node to prompt the user to enter their password.
6. Connect the nodes as follows and save the tree:



Register a PingGateway agent in AM

In AM 7 and later versions, follow these steps to register an agent that acts on behalf of PingGateway.

1. In the AM admin UI, select the top-level realm, and then select **Applications** > **Agents** > **Identity Gateway**.
2. Add an agent with the following configuration, leaving other options blank or with the default value:

   For SSO | For CDSSO

   1. **Agent ID** : `ig_agent`
   2. **Password** : `password`

3. (Optional - From AM 7.5) Use AM's secret service to manage the agent profile password. If AM finds a matching secret in a secret store, it uses that secret instead of the agent password configured in Step 2.

   a. In the agent profile page, set a label for the agent password in **Secret Label Identifier**.

      AM uses the identifier to generate a secret label for the agent.

      The secret label has the format `am.application.agents.identifier.secret` , where identifier is the **Secret Label Identifier**.

      The **Secret Label Identifier** can contain only characters `a-z` , `A-Z` , `0-9` , and periods (`.`). It can't start or end with a period.

   b. Select **Secret Stores** and configure a secret store.

   c. Map the label to the secret. Learn more from AM's mapping.

   Note the following points for using AM's secret service:

   - Set a **Secret Label Identifier** that clearly identifies the agent.
   - If you update or delete the **Secret Label Identifier**, AM updates or deletes the corresponding mapping for the previous identifier provided no other agent shares the mapping.
   - When you rotate a secret, update the corresponding mapping.

Set up a demo user in AM

AM is provided with a demo user in the top-level realm, with the following credentials:

- ID/username: `demo`

- Last name: `user`
- Password: `Ch4ng31t`
- Email address: `demo@example.com`
- Employee number: `123`

For information about how to manage identities in AM, refer to AM's Identity stores.

Find the AM session cookie name

In routes that use AmService, PingGateway retrieves AM's SSO cookie name from the `ssoTokenHeader` property or from AM's `/serverinfo/*` endpoint.

In other circumstances where you need to find the SSO cookie name, access `http://am-base-url/serverinfo/*`. For example, access the AM endpoint with `curl`:

```
$ curl http://am.example.com:8088/openam/json/serverinfo/*
```

External tools used in this guide

The examples in this guide use some of the following third-party tools:

- **`curl`**: https://curl.haxx.se
- **`HTTPie`**: https://httpie.org
- **`jq`**: https://stedolan.github.io/jq/
- **`keytool`**: https://docs.oracle.com/en/java/javase/17/docs/specs/man/keytool.html

Authentication

Single sign-on (SSO)

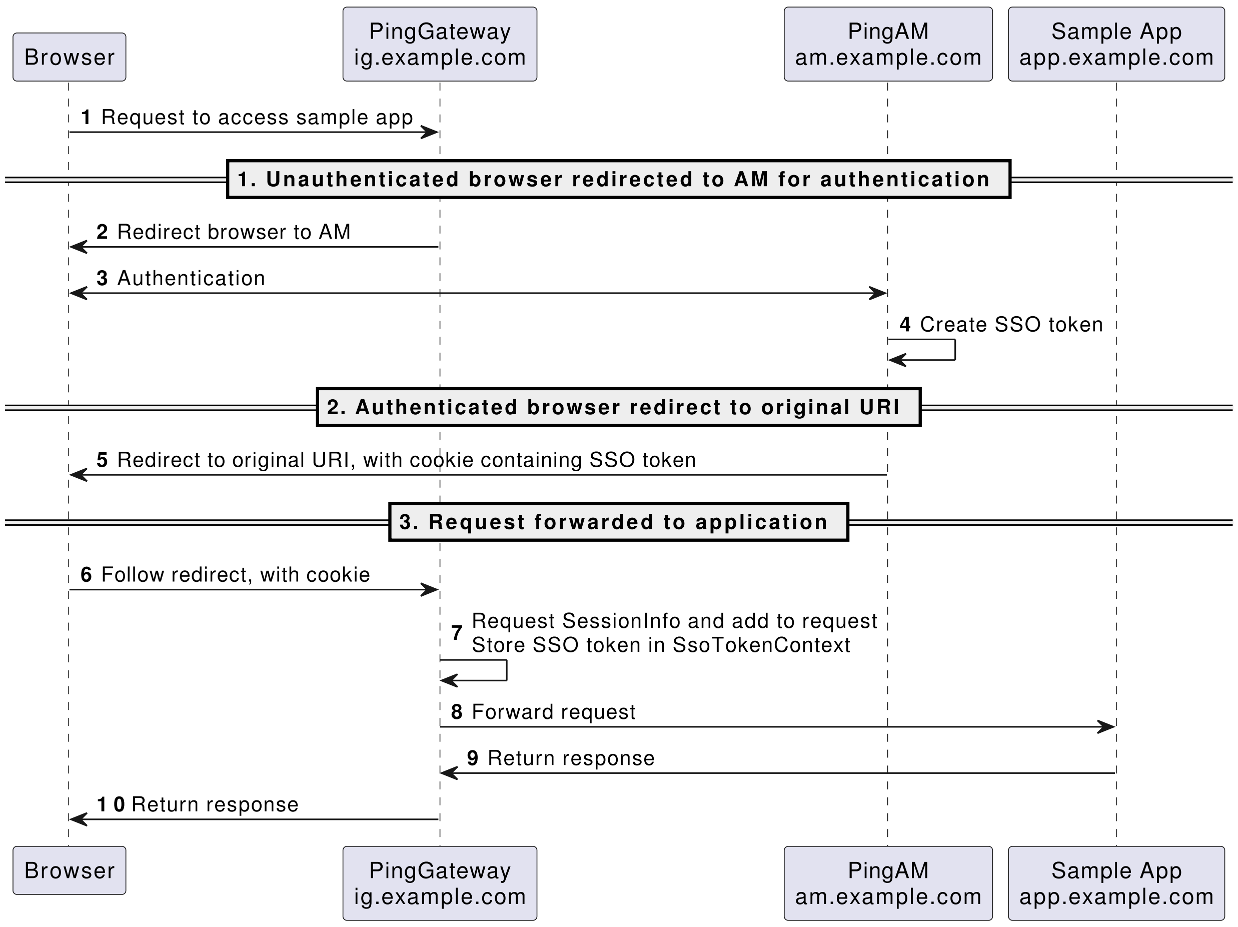
The following sections describe how to set up SSO for requests in the same domain:

IMPORTANT

To require users to authenticate in the correct realm for security reasons, configure SSO or CDSSO with a PolicyEnforcementFilter that refers to an AM policy where the realm is enforced. For an example, refer to Require authentication to a realm.

In SSO using the SingleSignOnFilter, PingGateway processes a request using AM authentication. PingGateway and the authentication provider must run on the same domain.

The following sequence diagram shows the flow of information during SSO between PingGateway and AM as the authentication provider.


Browser
PingGateway ig.example.com
PingAM am.example.com
Sample App app.example.com
1 Request to access sample app
1. Unauthenticated browser redirected to AM for authentication
2 Redirect browser to AM
3 Authentication
4 Create SSO token
2. Authenticated browser redirect to original URI
5 Redirect to original URI, with cookie containing SSO token
3. Request forwarded to application
6 Follow redirect, with cookie
7 Request SessionInfo and add to request Store SSO token in SsoTokenContext
8 Forward request
9 Return response
10 Return response

- The browser sends an unauthenticated request to access the sample application.
- PingGateway intercepts the request, and redirects the browser to AM for authentication.
- AM authenticates the user, creates an SSO token.
- AM redirects the request back to the original URI with the token in a cookie and the browser follows the redirect to PingGateway.

- PingGateway validates the token it gets from the cookie. It adds the AM session info to the request and stores the SSO token in the context for downstream filters and handlers.
- PingGateway forwards the request to the sample application and the sample application returns the requested resource to the browser.

Next steps

- Use the default journey
- Use a specific journey

Use the default journey

This page shows how to authenticate with SSO using the default AM authentication journey (tree).

Before you begin, prepare AM, PingGateway, and the sample application. Learn more in the example installation for this guide.

1. Set up AM:
   a. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:
      - `https://ig.example.com:8443/*`
      - `https://ig.example.com:8443/*?*`

   b. Register a PingGateway agent with the following values, as described in Register a PingGateway agent in AM:
      - **Agent ID**: `ig_agent`
      - **Password**: `password`

        IMPORTANT

        Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

   c. (Optional) Authenticate the agent to AM as described in Authenticate a PingGateway agent to AM.

      IMPORTANT

      PingGateway agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

   d. Select **Configure** > **Global Services** > **Platform**, and add `example.com` as an AM cookie domain.

      By default, AM sets host-based cookies. After authentication with AM, requests can be redirected to AM instead of to the resource.

2. Set up PingGateway:

   a. Set up PingGateway for HTTPS, as described in Configure PingGateway for TLS (server-side).

   b. Set an environment variable for the PingGateway agent password, and then restart PingGateway:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

      The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

   c. Add the following route to PingGateway to serve the sample application .css and other static resources:

      Linux Windows

```
$HOME/.openig/config/routes/00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or
matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

   d. Add the following route to PingGateway:

      Linux Windows

```
$HOME/.openig/config/routes/sso.json
```

```
{
  "name": "sso",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/sso$')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SingleSignOnFilter-1",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

For information about how to set up the PingGateway route in Studio, refer to Policy enforcement in Structured Editor or Protecting a web app with Freeform Designer.

3. Test the setup:

   a. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/sso.

   b. Log in to AM as user `demo`, password `Ch4ng31t`.

The SingleSignOnFilter passes the request to sample application, which displays the sample application home page.

Use a specific journey

This page shows how to authenticate with SSO and the example AM authentication journey (tree) instead of the default authentication journey.

1. Set up the example in Use the default journey.
2. Add the following route to PingGateway:

Linux Windows

```
$HOME/.openig/config/routes/sso-authservice.json
```

```
{
  "name": "sso-authservice",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/sso-
authservice')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SingleSignOnFilter-1",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1",
            "authenticationService": "Example"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the features of the route compared to `sso.json`:

- The route matches requests to `/home/sso-authservice`.
- The `authenticationService` property of SingleSignOnFilter refers to `Example`, the name of the example authentication tree in AM. This authentication tree is used for authentication instead of the AM admin UI.

3. Test the setup:

   a. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/sso-authservice.

   b. If you see warnings that the site isn't secure, respond to the warnings to access the site.

   c. Note that the login page is different from Use the default journey.

Cross-domain single sign-on (CDSSO)

This page shows how to set up CDSSO for requests in a different domain:

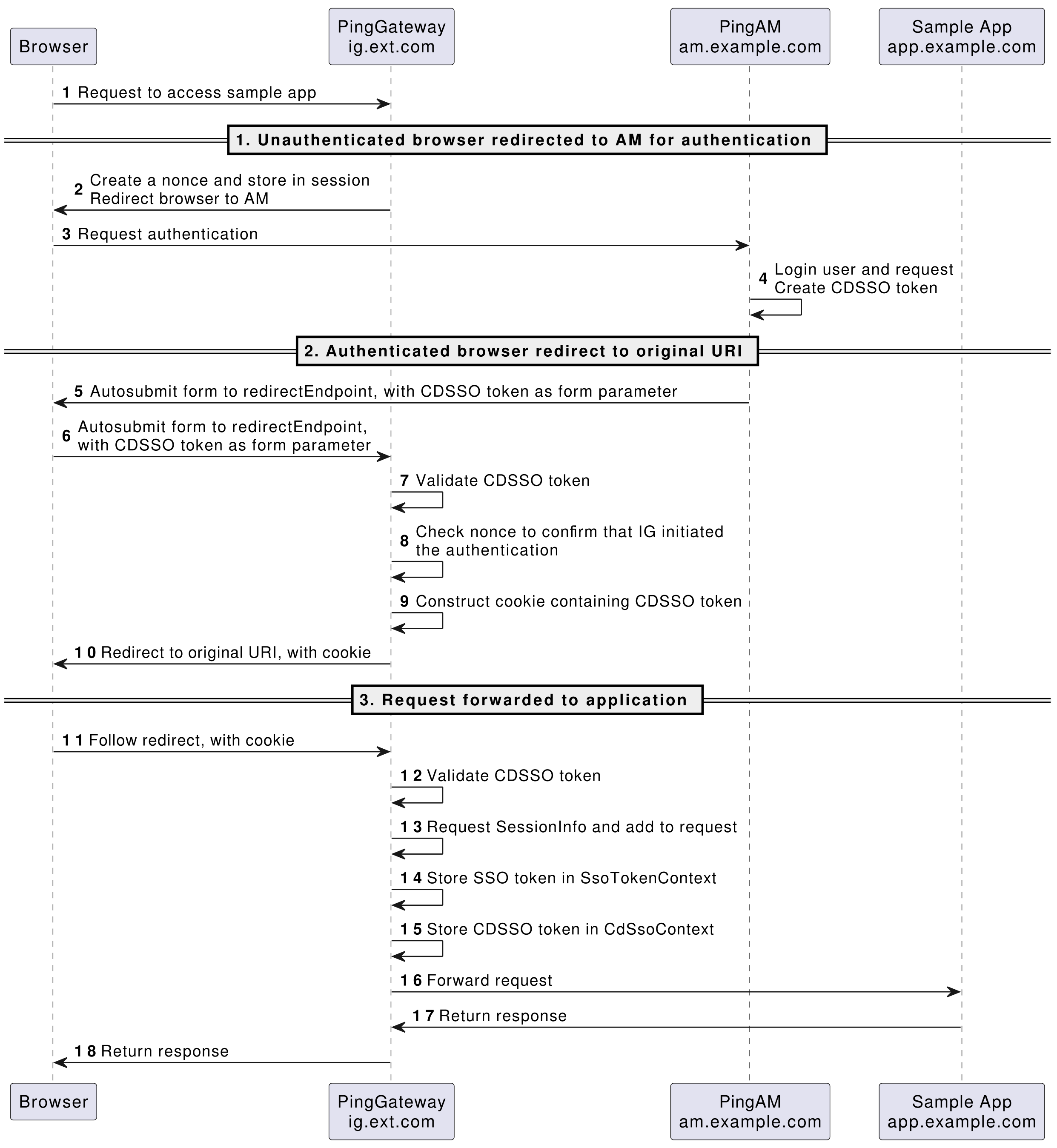
IMPORTANT

To require users to authenticate in the correct realm for security reasons, configure SSO or CDSSO with a PolicyEnforcementFilter that refers to an AM policy where the realm is enforced. You can find an example in Require authentication to a realm.

The SSO mechanism depicted in the following diagram can be used when PingGateway and AM are running in the same domain. When PingGateway and AM are running in different domains, AM cookies aren't visible to PingGateway because of the same-origin policy.

CDSSO using the CrossDomainSingleSignOnFilter provides a mechanism to push tokens issued by AM to PingGateway running in a different domain.

The following sequence diagram shows the flow of information between PingGateway, AM, and the sample application during CDSSO. In this example, AM is running on `am.example.com`, and PingGateway is running on `ig.ext.com`.


Browser
PingGateway ig.ext.com
PingAM am.example.com
Sample App app.example.com
1 Request to access sample app
1. Unauthenticated browser redirected to AM for authentication
2 Create a nonce and store in session Redirect browser to AM
3 Request authentication
4 Login user and request Create CDSSO token
2. Authenticated browser redirect to original URI
5 Autosubmit form to redirectEndpoint, with CDSSO token as form parameter
6 Autosubmit form to redirectEndpoint, with CDSSO token as form parameter
7 Validate CDSSO token
8 Check nonce to confirm that IG initiated the authentication
9 Construct cookie containing CDSSO token
10 Redirect to original URI, with cookie
3. Request forwarded to application
11 Follow redirect, with cookie
12 Validate CDSSO token
13 Request SessionInfo and add to request
14 Store SSO token in SsoTokenContext
15 Store CDSSO token in CdSsoContext
16 Forward request
17 Return response
18 Return response

1. The browser sends an unauthenticated request to access the sample app.

2-3. PingGateway intercepts the request, and redirects the browser to AM for authentication.

4. AM authenticates the user and creates a CDSSO token.

5. AM responds to a successful authentication with an HTML autosubmit form containing the issued token.

6. The browser loads the HTML and autosubmit form parameters to the PingGateway callback URL for the redirect endpoint.

7. When `verificationSecretId` in CrossDomainSingleSignOnFilter is configured, PingGateway uses it to verify signature of AM session tokens.

When `verificationSecretId` isn't configured, PingGateway discovers and uses the AM JWK set to verify the signature of AM session tokens.

If that fails, the CrossDomainSingleSignOnFilter fails to load.

8. PingGateway checks the nonce found inside the CDSSO token to confirm that the callback comes from an authentication initiated by PingGateway.

9. PingGateway constructs a cookie, and fulfills it with a cookie name, path, and domain, using the CrossDomainSingleSignOnFilter property `authCookie`. The domain must match that set in the AM PingGateway agent.

10-11. PingGateway redirects the request back to the original URI, with the cookie, and the browser follows the redirect back to PingGateway.

12. PingGateway validates the SSO token inside of the CDSSO token

13-15. PingGateway adds the AM session info to the request, and stores the SSO token and CDSSO token in the contexts for use by downstream filters and handlers.

16-18. PingGateway forwards the request to the sample application, and the sample application returns the requested resource to the browser.

Before you begin, prepare AM, PingGateway, and the sample application. Learn more in the example installation for this guide.

1. Set up AM:

   a. Register a PingGateway agent with the following values, as described in Register a PingGateway agent in AM:

      - **Agent ID**: `ig_agent_cdsso`
      - **Password**: `password`
      - **Redirect URL for CDSSO**: `https://ig.ext.com:8443/home/cdsso/redirect`

      IMPORTANT

      Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

   b. (Optional) Authenticate the agent to AM as described in Authenticate a PingGateway agent to AM.

IMPORTANT

IMPORTANT

PingGateway agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

c. Select **Services** > **Add a Service**, and add a **Validation Service** with the following **Valid goto URL Resources**:

   - `https://ig.ext.com:8443/*`
   - `https://ig.ext.com:8443/*?*`

d. Select **Configure** > **Global Services** > **Platform**, and add `example.com` as an AM cookie domain.

   By default, AM sets host-based cookies. After authentication with AM, requests can be redirected to AM instead of to the resource.

2. Set up PingGateway:

   a. Set up PingGateway for HTTPS, as described in Configure PingGateway for TLS (server-side).

   b. Add the following `session` configuration to `admin.json`.

   This ensures the browser passes the session cookie in the form-POST to the redirect endpoint (step 6 of Information flow during CDSSO):

```
{
  "connectors": […],
  "session": {
    "type": "InMemorySessionManager",
    "config": {
      "cookie": {
        "sameSite": "none",
        "secure": true
      }
    }
  },
  "heap": […]
}
```

   This step is required for the following reasons:

   - When `sameSite` is `strict` or `lax`, the browser doesn't send the session cookie, which contains the nonce used in validation. If PingGateway doesn't find the nonce, it assumes that the authentication failed.
   - When `secure` is `false`, the browser is likely to reject the session cookie.

Learn more in AdminHttpApplication (`admin.json`).

c. Set an environment variable for the PingGateway agent password, and then restart PingGateway:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

d. Add the following route to PingGateway to serve the sample application .css and other static resources:

Linux | Windows

```
$HOME/.openig/config/routes/00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or
matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

e. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/cdsso.json
```

```
{
  "name": "cdsso",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path,
'^/home/cdsso')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "http://am.example.com:8088/openam",
        "realm": "/",
        "agent": {
          "username": "ig_agent_cdsso",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "sessionCache": {
          "enabled": false
        }
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "CrossDomainSingleSignOnFilter-1",
          "type": "CrossDomainSingleSignOnFilter",
          "config": {
            "redirectEndpoint": "/home/cdsso/redirect",
            "authCookie": {
              "path": "/home",
              "name": "ig-token-cookie"
            },
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/home/cdsso`.
- The agent password for AmService is provided by a SystemAndEnvSecretStore in the heap.

- Because the CrossDomainSingleSignOnFilter’s `verificationSecretId` isn’t configured, PingGateway discovers and uses the AM JWK set to verify the signature of AM session tokens. If that fails, the CrossDomainSingleSignOnFilter fails to load.

3. Test the setup:

   a. In your browser’s privacy or incognito mode, go to https://ig.ext.com:8443/home/cdsso.

      The CrossDomainSingleSignOnFilter redirects the request to AM for authentication.

   b. Log in to AM as user `demo`, password `Ch4ng31t`.

      When you have authenticated, AM calls `/home/cdsso/redirect`, and includes the CDSSO token. The CrossDomainSingleSignOnFilter passes the request to sample app, which returns the home page.

Password replay with AM

Password replay brings SSO to legacy web applications without the need to edit, upgrade, or recode them.

Use PingGateway with an appropriate AM authentication tree to capture and replay username password credentials. PingGateway and AM share a secret key to encrypt and decrypt the password and keep it safe.

When running PingOne Advanced Identity Cloud, read Password replay instead.

Request flow

The following figure illustrates the flow of requests when an unauthenticated user accesses a protected application. After authenticating with AM, the user is logged into the application with the username and password from the AM login session.

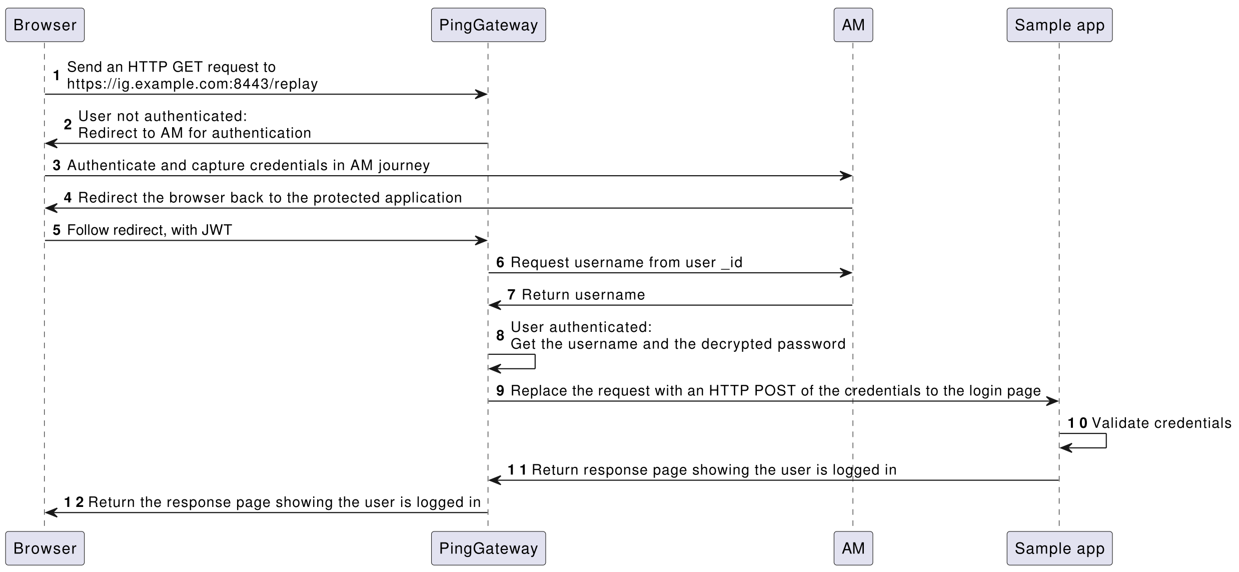

Browser
PingGateway
AM
Sample app
1 Send an HTTP GET request to https://ig.example.com:8443/replay
2 User not authenticated: Redirect to AM for authentication
3 Authenticate and capture credentials in AM journey
4 Redirect the browser back to the protected application
5 Follow redirect, with JWT
6 Request username from user _id
7 Return username
8 User authenticated: Get the username and the decrypted password
9 Replace the request with an HTTP POST of the credentials to the login page
10 Validate credentials
11 Return response page showing the user is logged in
12 Return the response page showing the user is logged in

Figure 1. Password replay sequence diagram

- PingGateway intercepts the browser's HTTP GET request.
- PingGateway redirects the user to AM for authentication.
- AM authenticates the user and stores the encrypted password in a JWT.
- AM redirects the browser back to the protected application with the JWT.
- PingGateway intercepts the browser's HTTP GET request again:
  - The user is authenticated.
  - PingGateway gets the password from the JWT and decrypts it.
  - PingGateway gets the username from AM based on the user `_id`.
- PingGateway replaces the request with an HTTP POST to the application, taking the credentials from the context.
- The sample application validates the credentials from the HTTP POST request.
- The sample application responds with the user's profile page.
- PingGateway passes the response from the sample application to the browser.

Tasks

Before you begin

Before you begin, prepare AM, PingGateway, and the sample application. Learn more in the example installation for this guide.

Task 1: Prepare a shared secret

PingGateway and AM share a secret symmetric key to encrypt and decrypt the password.

TIP

The following tasks use the AM default `aestest` 256-bit AES test key as the shared secret.

Don’t use the test key in production. In a production deployment, generate a random AES 256-bit key to use as a shared secret. How you generate the secret key is up to you. For example:

```
$ openssl rand -base64 32
YWJjZGVmZ2hpamtsbW5vcHFyc3R1dnd5eHowMTIzNDU=
```

Keep the secret safe.

You’ll set the AM secret label `scripted.node.aes.key` to reference it in a script to encrypt the user’s credentials. You’ll also share the secret with PingGateway to decrypt the user’s credentials before replaying them.

Task 2: Configure AM

This example uses the `default-passwords-store` secret store for the shared secret. In a production deployment, use your own secret store and shared secret:

1. Go to http://am.example.com:8088/openam/encode.jsp and use the page to encode the base64-encoded AES key.

   In this example, the base64-encoded AES key is `YWJjZGVmZ2hpamtsbW5vcHFyc3R1dnd5eHowMTIzNDU=`.

2. Write the result to a `default-passwords-store` file named `scripted.node.aes.key`:

   ```
   $ echo -n <encoded-aes-key> >
   /path/to/openam/security/secrets/encrypted/scripted.node.aes.key
   ```

   Only secret labels that begin with the string `scripted.node.` are accessible to scripts.

3. Select **Realms** > **Top Level Realm** > **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:

   - `https://ig.example.com:8443/*`
   - `https://ig.example.com:8443/*?*`

4. In the agent profile for PingGateway, add a redirect URI for CDSSO that PingGateway can use in the route for password replay.

Go to **Applications** > **Agents** > **Identity Gateway** > **ig_agent**, set **Redirect URLs for CDSSO** to `https://ig.example.com:8443/replay/redirect` , and click **Save Changes**.

5. Go to **Scripts**, click **+ New Script** and create a **Decision node script for authentication trees** with the **Legacy** engine named `Password replay` and the following JavaScript content.

   The script adds a JWT encrypted with the shared secret key to the session:

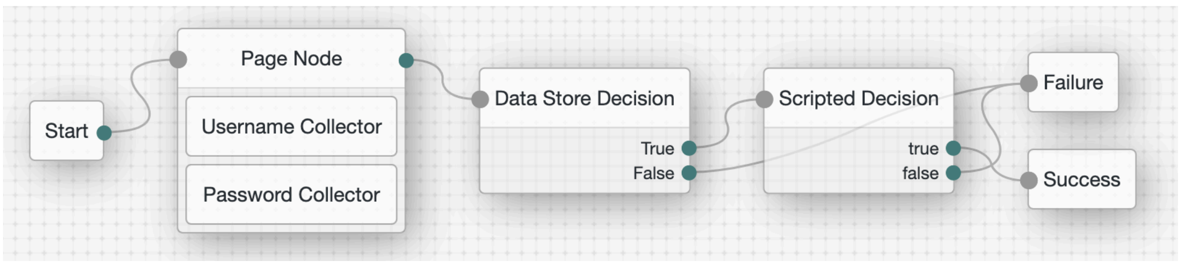
   ▼ *Show script*

```
var fr = JavaImporter(
  org.forgerock.openam.auth.node.api.Action,
  org.forgerock.openam.auth.node.api,
  javax.security.auth.callback.TextOutputCallback,
  org.forgerock.json.jose.builders.JwtBuilderFactory,
  org.forgerock.json.jose.jwt.JwtClaimsSet,
  org.forgerock.json.jose.jwe.JweAlgorithm,
  org.forgerock.json.jose.jwe.EncryptionMethod,
  javax.crypto.spec.SecretKeySpec,
  org.forgerock.secrets.SecretBuilder,
  org.forgerock.util.encode.Base64
);

var NodeOutcome = {
  ERROR: 'false',
  SUCCESS: 'true'
};

var config = {
  encryptionKey:
secrets.getGenericSecret("scripted.node.aes.key").getAsUtf8(
)
};

function getKey () {
  logger.message("encKey: " + config.encryptionKey)
  return new
fr.SecretKeySpec(fr.Base64.decode(config.encryptionKey),
'AES');
}

function buildJwt (claims) {
  logger.message('Building response JWT');
  var encryptionKey = getKey();
  var jwtClaims = new fr.JwtClaimsSet;
```

```
    jwtClaims.setClaims(claims);
    var jwt = new fr.JwtBuilderFactory()
                    .jwe(encryptionKey)
                    .headers()
                    .alg(fr.JweAlgorithm.DIRECT)
                    .enc(fr.EncryptionMethod.A128CBC_HS256)
                    .done()
                    .claims(jwtClaims)
                    .build();
    return jwt;
  }

  try {
    password=nodeState.get("password").asString()
    var registrationClaims = { password: password };
    passwordJwt = buildJwt(registrationClaims);
    action =
  fr.Action.goTo(NodeOutcome.SUCCESS).putSessionProperty("sunI
  dentityUserPassword", passwordJwt).build();
  } catch (e) {
    logger.error('ERROR ' + e);
    action = fr.Action.send(new
  fr.TextOutputCallback(fr.TextOutputCallback.ERROR,
  e.toString())).build();
  }
```

You can download the script as password-replay-am.js.

6. Go to **Authentication** > **Trees**, click **+ Create Tree**, and create an authentication tree named `Password replay` configured with the nodes shown in this screenshot:



   - The Page node presents a page with input fields to prompt for the username and password.
     - The Username collector node collects and injects the `userName` into the shared node state.
     - The Password collector node collects and injects the `password` into the shared node state.

- The Data Store Decision node uses the username and password to determine whether authentication is successful.
- The Scripted Decision node references your script and has the same outcomes: `true` and `false`.

7. Make sure the allowlist for scripted decision nodes includes all the classes the script requires.

   Go to **Configure** > **Global Services** > **Scripting** > **Secondary Configurations** > **AUTHENTICATION_TREE_DECISION_NODE** > **Secondary Configurations** > **engineConfiguration**, add these Java classes to the allowlist, then click **Save Changes**:

   - `javax.crypto.spec.SecretKeySpec`
   - `org.forgerock.json.jose.builders.EncryptedJwtBuilder`
   - `org.forgerock.json.jose.builders.JweHeaderBuilder`
   - `org.forgerock.json.jose.builders.JwtBuilderFactory`
   - `org.forgerock.json.jose.jwe.EncryptionMethod`
   - `org.forgerock.json.jose.jwe.JweAlgorithm`
   - `org.forgerock.json.jose.jwt.JwtClaimsSet`
   - `org.forgerock.secrets.SecretBuilder`
   - `org.forgerock.util.encode.Base64`

8. Make sure the test user can log in with the tree before continuing.

   To verify the tree works as expected, in your browser's privacy or incognito mode, go to http://am.example.com:8088/openam/XUI/?service=Password%20replay#login and log in as test user `demo` with password `Ch4ng31t`.

Task 3: Configure PingGateway

The password replay configuration includes the PingGateway password to connect to AM, the shared secret, and the route for password replay.

1. Set up PingGateway for HTTPS.

   Learn more in Configure PingGateway for TLS (server-side).

2. Set an environment variable locally on the computer running PingGateway to access the agent password:

   ```
   # The base64-encoded PingGateway agent "password":
   $ export AGENT_SECRET_ID='cGFzc3dvcmQ='
   ```

PingGateway retrieves the password with a SystemAndEnvSecretStore, which requires it to be base64-encoded.

The password must match the agent profile password you set in AM. PingGateway uses the password to connect to AM.

3. Set an environment variable locally on the computer running PingGateway to access the shared secret key:

```
# The base64-encoded shared secret key:
$ export
AES_KEY='YWJjZGVmZ2hpamtsbW5vcHFyc3R1dnd5eHowMTIzNDU='
```

PingGateway retrieves the shared secret with a SystemAndEnvSecretStore, which requires it to be base64-encoded.

The shared secret key must match the secret key the AM script uses to encrypt the password to replay. PingGateway uses the secret key to decrypt the password to replay.

4. Restart PingGateway to read the secrets from the environment.

5. Add a route to PingGateway to serve the sample application CSS and other static resources:

Linux | Windows

```
$HOME/.openig/config/routes/00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or
matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

6. Add a route for password replay:

Linux | Windows

```
$HOME/.openig/config/routes/04-replay.json
```

```
{
  "name": "04-replay",
  "condition": "${find(request.uri.path, '^/replay')}",
  "properties": {
    "amInstanceUrl": "http://am.example.com:8088/openam/"
  },
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore",
      "config": {
        "mappings": [
          {
            "secretId": "aes.key",
            "format": {
              "type": "SecretKeyPropertyFormat",
              "config": {
                "format": "BASE64",
                "algorithm": "AES"
              }
            }
          }
        ]
      }
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "&{amInstanceUrl}"
      }
    },
    {
      "name": "CapturedUserPasswordFilter-1",
      "type": "CapturedUserPasswordFilter",
      "config": {
        "ssoToken": "${contexts.ssoToken.value}",
        "keySecretId": "aes.key",
        "keyType": "AES",
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "amService": "AmService-1"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "CrossDomainSingleSignOnFilter-1",
          "type": "CrossDomainSingleSignOnFilter",
          "config": {
            "redirectEndpoint": "/replay/redirect",
            "authCookie": {
              "path": "/replay",
              "name": "ig-token-cookie"
            },
            "amService": "AmService-1",
            "authenticationService": "Password replay"
          }
        },
        {
          "name": "UserProfileFilter-1",
          "type": "UserProfileFilter",
          "config": {
            "username": "${contexts.ssoToken.info.uid}",
            "userProfileService": {
              "type": "UserProfileService",
              "config": {
                "amService": "AmService-1",
                "profileAttributes": [
                  "username"
                ]
              }
            }
          }
        },
        {
          "name": "PasswordReplayFilter-1",
          "type": "PasswordReplayFilter",
          "config": {
            "loginPage": "${true}",
            "credentials": "CapturedUserPasswordFilter-1",
            "request": {
              "method": "POST",
              "uri": "http://app.example.com:8081/login",
              "form": {
                "username": [
                  "${contexts.userProfile.username}"
                ],
                "password": [
                  "${contexts.capturedPassword.value}"
                ]
              }
            }
          },
          "capture": [
            "all"
          ]
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

The route:

- Matches requests whose path starts with `/replay`.
- Sets an `amInstanceUrl` property to the URL for AM.
- Loads the `aes.key` secret key from the local `AES_KEY` environment variable.
- Connects to AM as `ig_agent` with the `agent.secret.id` password from the local `AGENT_SECRET_ID` environment variable.
- Extracts the captured password from the SSO token context and decrypts it with the `aes.key` secret key.
- Uses a CrossDomainSingleSignOnFilter to redirect to the AM `Password replay` tree for authentication, getting the authentication information from the `ig-token-cookie`.
- Queries AM to retrieve the username for logging in.

  You can retrieve other profile attributes with the UserProfileFilter, such as the email address or first and last names. The sample application expects the username in this example, so the route gets the username.
- Logs in to the sample application with the username and password.
- Returns the result to the browser.

In production, remove `"capture": ["all"]` from the PasswordReplayFilter to avoid recording credentials in the logs.

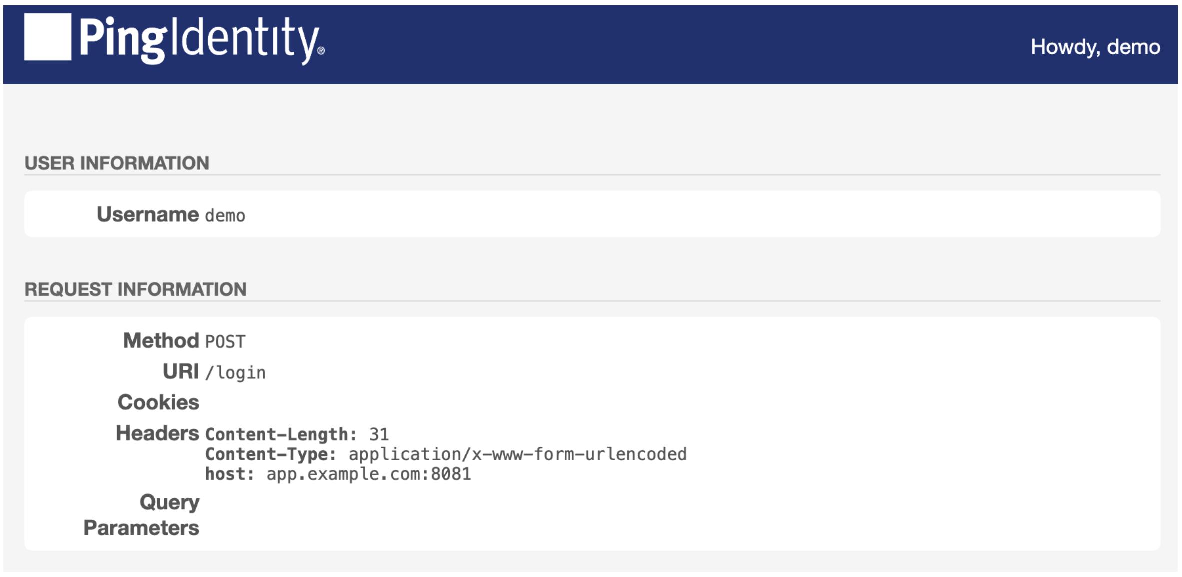
Validation

1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/replay/.

   PingGateway redirects to the AM authentication tree.

2. Log in as test user `demo` with password `Ch4ng31t`.

   PingGateway successfully replays the credentials against the sample application. The sample application displays its user profile page:



USER INFORMATION

Username demo

REQUEST INFORMATION

Method POST
URI /login
Cookies
Headers
```
Content-Length: 31
Content-Type: application/x-www-form-urlencoded
host: app.example.com:8081
```
Query Parameters

3. Review the PingGateway output.

   On success, the output displays the credentials and the profile page:

```
...INFO  o.f.o.d.c.C.c.PasswordReplayFilter-1 @04-replay -

[CONTINUED]--- (filtered-request) exchangeId:<id> -
transactionId:<id> --->

[CONTINUED]POST http://app.example.com:8081/login HTTP/1.1
[CONTINUED]Content-Length: 31
[CONTINUED]Content-Type: application/x-www-form-urlencoded

[CONTINUED]password=Ch4ng31t&username=demo

...INFO  o.f.o.d.c.C.c.PasswordReplayFilter-1 @04-replay -

[CONTINUED]<--- (response) exchangeId:<id> - transactionId:
<id> ---

[CONTINUED]HTTP/1.1 200 OK
```

```
...

[CONTINUED]<!DOCTYPE html>
...
```

You have successfully demonstrated password replay with PingGateway and AM.

If password replay fails, consider the outcome of the HTTP POST from PingGateway to the sample application:

HTTP 401 Unauthorized

PingGateway is not replaying the credentials.

Review the PingGateway output to determine whether the username or password is missing when PingGateway replays the credentials.

If the password is missing, make sure PingGateway and AM share the same AES secret key.

HTTP 403 Forbidden

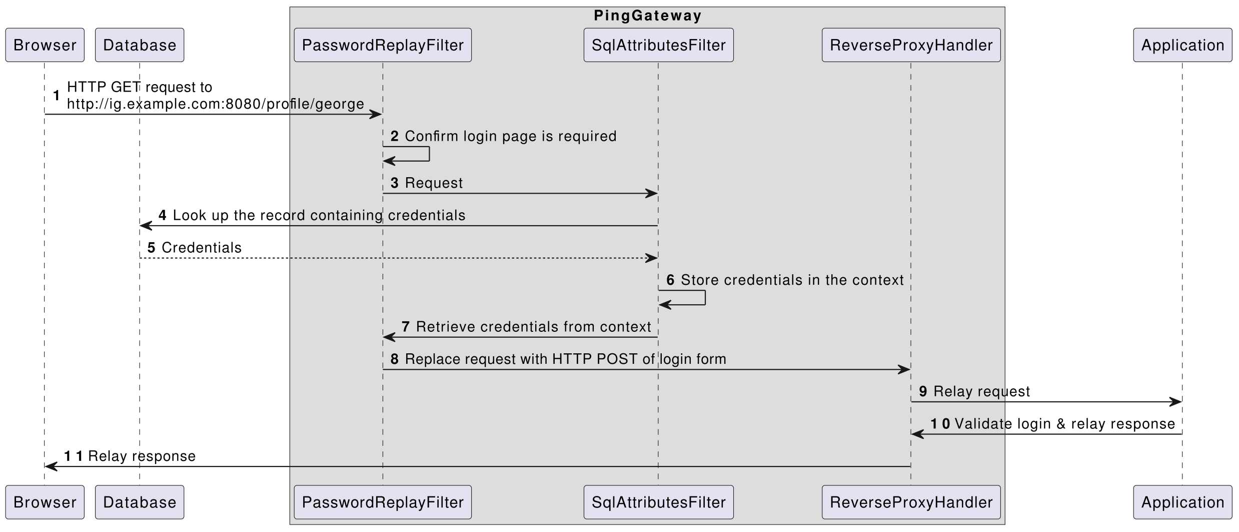
PingGateway is not replaying the right credentials.

Make sure you're using a username-password combination known to the sample application.

Password replay from a database

This page shows how to configure PingGateway to get credentials from a database. This example is tested with H2 1.4.197.

The following figure illustrates the flow of requests when PingGateway uses credentials from a database to log a user in to the sample application:


PingGateway
Browser
Database
PasswordReplayFilter
SqlAttributesFilter
ReverseProxyHandler
Application
1 HTTP GET request to http://ig.example.com:8080/profile/george
2 Confirm login page is required
3 Request
4 Look up the record containing credentials
5 Credentials
6 Store credentials in the context
7 Retrieve credentials from context
8 Replace request with HTTP POST of login form
9 Relay request
10 Validate login & relay response
11 Relay response

- PingGateway intercepts the browser’s HTTP GET request.
- The PasswordReplayFilter confirms that a login page is required, and passes the request to the SqlAttributesFilter.
- The SqlAttributesFilter uses the email address to look up credentials in H2, and stores them in the request context attributes map.
- The PasswordReplayFilter retrieves the credentials from the attributes map, builds the login form, and performs the HTTP POST request to the sample app.
- The sample application validates the credentials, and responds with a profile page.

Before you start, prepare PingGateway and the sample application as described in the Quick install.

1. Set up the database:
   a. On your system, add the following data in a comma-separated value file:

      Linux | Windows

      ```
      /tmp/userfile.txt
      ```

      ```
      username,password,fullname,email
      george,C0stanza,George Costanza,george@example.com
      kramer,N3wman12,Kramer,kramer@example.com
      bjensen,H1falutin,Babs Jensen,bjensen@example.com
      demo,Ch4ng31t,Demo User,demo@example.com
      kvaughan,B5ibery12,Kirsten Vaughan,kvaughan@example.com
      scarter,S9rain12,Sam Carter,scarter@example.com
      wolkig,Geh3imnis!,Wilhelm Wolkig,wolkig@example.com
      ```

   b. Download the H2 database, unpack it, and start it:

      ```
      $ sh /path/to/h2/bin/h2.sh
      ```

      H2 starts, listening on port 8082, and opens the H2 Console in a browser.

   c. In the H2 Console, select the following options, and then select Connect to access the console:

      - **Saved Settings** : `Generic H2 (Server)`
      - **Setting Name** : `Generic H2 (Server)`
      - **Driver Class**: `org.h2.Driver`
      - **JDBC URL**: `jdbc:h2:~/ig-credentials`
      - **User Name**: `sa`

   - **Password** : `password`

     TIP

     If you have run this example before but can't access the console now, try deleting your local `~/ig-credentials` files and starting H2 again.

d. In the console, add the following text, and then run it to create the user table:

```
DROP TABLE IF EXISTS USERS;
CREATE TABLE USERS AS SELECT * FROM
CSVREAD('/tmp/userfile.txt');
```

e. In the console, add the following text, and then run it to verify that the table contains the same users as the file:

```
SELECT * FROM users;
```

f. Add the .jar file `/path/to/h2/bin/h2-*.jar` to the PingGateway configuration:

   - Create the directory `$HOME/.openig/extra`, where `$HOME/.openig` is the instance directory, and add .jar files to the directory.

2. Set up PingGateway:

   a. Set up PingGateway for HTTPS, as described in Configure PingGateway for TLS (server-side).

   b. Set an environment variable for the database password, and then restart PingGateway:

```
$ export DATABASE_PASSWORD='cGFzc3dvcmQ='
```

   The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

   c. Add the following route to PingGateway to serve the sample application .css and other static resources:

   Linux | Windows

```
$HOME/.openig/config/routes/00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or
matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

d. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/03-sql.json
```

```
{
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "JdbcDataSource-1",
      "type": "JdbcDataSource",
      "config": {
        "driverClassName": "org.h2.Driver",
        "jdbcUrl": "jdbc:h2:tcp://localhost/~/ig-
credentials",
        "username": "sa",
        "passwordSecretId": "database.password",
        "secretsProvider": "SystemAndEnvSecretStore-1"
      }
    }
  ],
  "name": "sql",
  "condition": "${find(request.uri.path, '^/profile')}",
  "handler": {
    "type": "Chain",
    "baseURI": "http://app.example.com:8081",
    "config": {
      "filters": [
        {
          "type": "PasswordReplayFilter",
          "config": {
            "loginPage": "${find(request.uri.path,
'^/profile/george') and (request.method == 'GET')}",
            "credentials": {
              "type": "SqlAttributesFilter",
              "config": {
                "dataSource": "JdbcDataSource-1",
                "preparedStatement":
                "SELECT username, password FROM users
WHERE email = ?;",
                "parameters": [
                  "george@example.com"
                ]
              }
            },
            "request": {
              "method": "POST",
              "uri": "http://app.example.com:8081/login",
              "form": {
                "username": [
                  "${contexts.sqlAttributes.row.USERNAME}"
                ],
                "password": [
                  "${contexts.sqlAttributes.row.PASSWORD}"
                ]
              }
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/profile`.
- The PasswordReplayFilter specifies a loginPage page property:
- When a request is an HTTP GET, and the request URI path is `/profile/george`, the expression resolves to `true`. The request is directed to a login page.

  The `SqlAttributesFilter` specifies the data source to access, a prepared statement to look up the user's record, and a parameter to pass into the statement.

The `request` object retrieves the username and password from the context, and replaces the browser's original HTTP GET request with an HTTP POST login request, containing the credentials to authenticate.

The request is for `username, password`, but H2 returns the fields as `USERNAME and PASSWORD`. The configuration reflects this difference.

- For other requests, the expression resolves to `false`. The request passes to the ReverseProxyHandler, which directs it to the profile page of the sample app.

3. Test the setup:

   a. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/profile.

      If you see warnings that the site isn't secure, respond to the warnings to access the site.

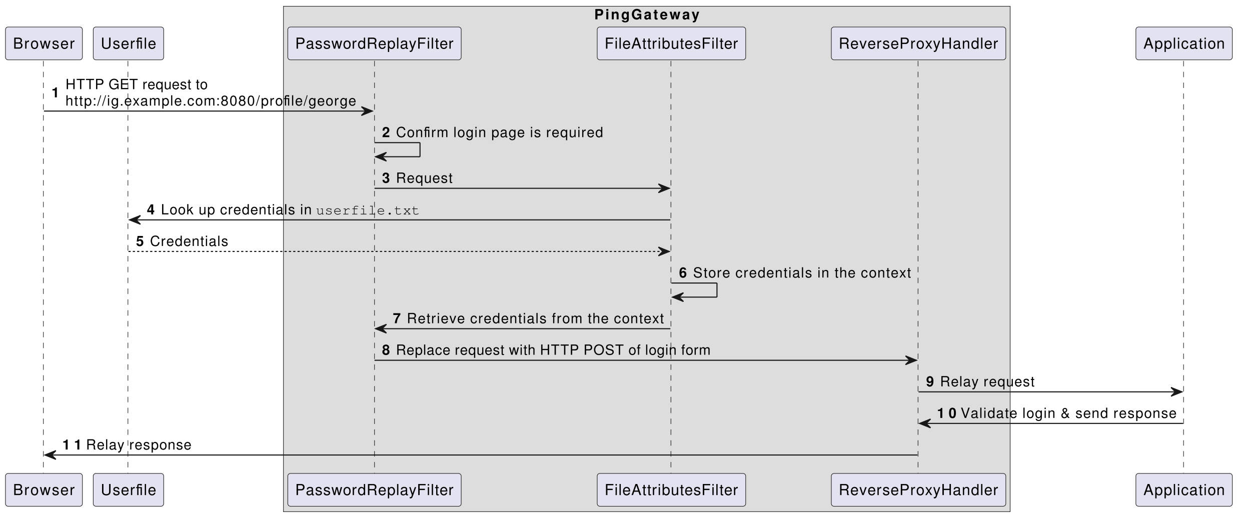
      Because the property `loginPage` resolves to `false`, the PasswordReplayFilter passes the request directly to the ReverseProxyHandler. The sample app returns the login page.

   b. Go to https://ig.example.com:8443/profile/george.

      Because the property `loginPage` resolves to `true`, the PasswordReplayFilter processes the request to obtain the login credentials. The sample app returns the profile page for George.

Password replay from a file

The following figure illustrates the flow of requests when PingGateway uses credentials in a file to log a user in to the sample application:


PingGateway
Browser
Userfile
PasswordReplayFilter
FileAttributesFilter
ReverseProxyHandler
Application
1 HTTP GET request to http://ig.example.com:8080/profile/george
2 Confirm login page is required
3 Request
4 Look up credentials in userfile.txt
5 Credentials
6 Store credentials in the context
7 Retrieve credentials from the context
8 Replace request with HTTP POST of login form
9 Relay request
10 Validate login & send response
11 Relay response

- PingGateway intercepts the browser's HTTP GET request, which matches the route condition.
- The PasswordReplayFilter confirms that a login page is required, and
- The FileAttributesFilter uses the email address to look up the user credentials in a file, and stores the credentials in the request context attributes map.
- The PasswordReplayFilter retrieves the credentials from the attributes map, builds the login form, and performs the HTTP POST request to the sample app.
- The sample application validates the credentials, and responds with a profile page.
- The ReverseProxyHandler passes the response to the browser.

Before you start, prepare PingGateway and the sample application as described in the Quick install.

1. On your system, add the following data in a comma-separated value file:

   Linux Windows

   ```
   /tmp/userfile.txt
   ```

   ```
   username,password,fullname,email
   george,C0stanza,George Costanza,george@example.com
   kramer,N3wman12,Kramer,kramer@example.com
   bjensen,H1falutin,Babs Jensen,bjensen@example.com
   demo,Ch4ng31t,Demo User,demo@example.com
   kvaughan,B5ibery12,Kirsten Vaughan,kvaughan@example.com
   scarter,S9rain12,Sam Carter,scarter@example.com
   wolkig,Geh3imnis!,Wilhelm Wolkig,wolkig@example.com
   ```

2. Set up PingGateway:

   a. Set up PingGateway for HTTPS, as described in Configure PingGateway for TLS (server-side).

   b. Add the following route to PingGateway to serve the sample application .css and other static resources:

      Linux Windows

      ```
      $HOME/.openig/config/routes/00-static-resources.json
      ```

      ```
      {
        "name" : "00-static-resources",
      ```

```
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or
matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

c. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/02-file.json
```

```
{
  "name": "02-file",
  "condition": "${find(request.uri.path, '^/profile')}",
  "capture": "all",
  "handler": {
    "type": "Chain",
    "baseURI": "http://app.example.com:8081",
    "config": {
      "filters": [
        {
          "type": "PasswordReplayFilter",
          "config": {
            "loginPage": "${find(request.uri.path,
'^/profile/george') and (request.method == 'GET')}",
            "credentials": {
              "type": "FileAttributesFilter",
              "config": {
                "file": "/tmp/userfile.txt",
                "key": "email",
                "value": "george@example.com"
              }
            },
            "request": {
              "method": "POST",
              "uri": "http://app.example.com:8081/login",
              "form": {
                "username": [

"${contexts.fileAttributes.record.username}"
                ],
                "password": [
"${contexts.fileAttributes.record.password}"
                ]
              }
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/profile`.
- The `PasswordReplayFilter` specifies a `loginPage` page property:
  - When a request is an HTTP GET, and the request URI path is `/profile/george`, the expression resolves to `true`. The request is directed to a login page.

    The `FileAttributesFilter` looks up the key and value in `/tmp/userfile.txt`, and stores them in the context.

    The `request` object retrieves the username and password from the context, and replaces the browser's original HTTP GET request with an HTTP POST login request, containing the credentials to authenticate.
  - For other requests, the expression resolves to `false`. The request passes to the ReverseProxyHandler, which directs it to the profile page of the sample app.

3. Test the setup:

   a. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/profile/george.

   b. If you see warnings that the site isn't secure, respond to the warnings to access the site.

      Because the property `loginPage` resolves to `true`, the PasswordReplayFilter processes the request to obtain the login credentials. The sample app returns the profile page for George.

   c. Go to https://ig.example.com:8443/profile/bob, or to any other URI starting with `https://ig.example.com:8443/profile`.

Because the property `loginPage` resolves to `false`, the PasswordReplayFilter passes the request directly to the ReverseProxyHandler. The sample app returns the login page.

Session cache eviction

When WebSocket notifications are enabled in PingGateway, PingGateway receives notifications when the following events occur:

- A user logs out of AM
- An AM session is modified, closed, or times out
- An AM admin forces logout of user sessions (from AM 7.3)

The following procedure gives an example of how to change the configurations in Single sign-on (SSO) and Cross-domain single sign-on (CDSSO) to receive WebSocket notifications for session logout and to evict entries related to the session from the cache. For information about WebSocket notifications, refer to WebSocket notifications.

Before you start, set up and test the example in Single sign-on (SSO).

1. Websocket notifications are enabled by default. If they are disabled, enable them by adding the following configuration to the AmService in your route:

   ```
   "notifications": {
     "enabled": true
   }
   ```

2. Enable the session cache by adding the following configuration to the AmService in your route:

   ```
   "sessionCache": {
     "enabled": true
   }
   ```

3. In `logback.xml` add the following logger for WebSocket notifications, and then restart PingGateway:

   ```
   <logger name="org.forgerock.openig.tools.notifications.ws"
   level="TRACE" />
   ```

   For information, refer to Changing the log level for different object types.

4. On the AM console, log the demo user out of AM to end the AM session.

5. Note that the PingGateway system logs are updated with Websocket notifications about the logout:

```
... | TRACE | vert.x-eventloop-thread-4 |
o.f.o.t.n.w.l.DirectAmLink | @system | Received a message: {
"topic": ... "eventType": "LOGOUT" } }
... | TRACE | vert.x-eventloop-thread-4 |
o.f.o.t.n.w.SubscriptionService | @system | Notification
received... "eventType": "LOGOUT" }}
... | TRACE | vert.x-eventloop-thread-4 |
o.f.o.t.n.w.SubscriptionService | @system | Notification sent
to a [/agent/session.v2] listener
```

Policy enforcement

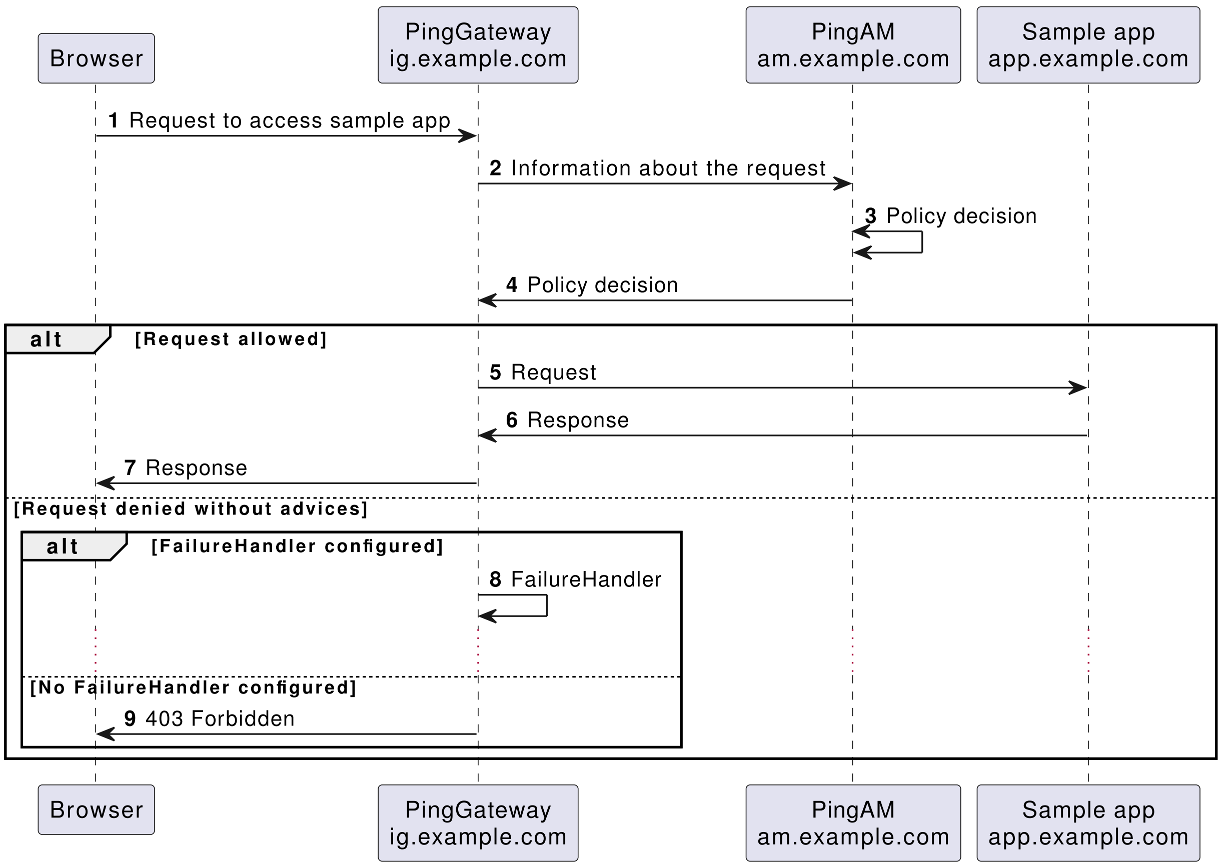
PingGateway as a policy enforcement point (PEP) uses the PolicyEnforcementFilter to intercept requests for a resource and provide information about the request to AM.

AM as a policy decision point (PDP) evaluates requests based on their context and the configured policies. AM then returns decisions indicating what actions are allowed or denied and any advice, subject attributes, or static attributes for the specified resources.

You can find more information in the PolicyEnforcementFilter and AM’s Authentication and SSO documentation.

Deny requests without advice

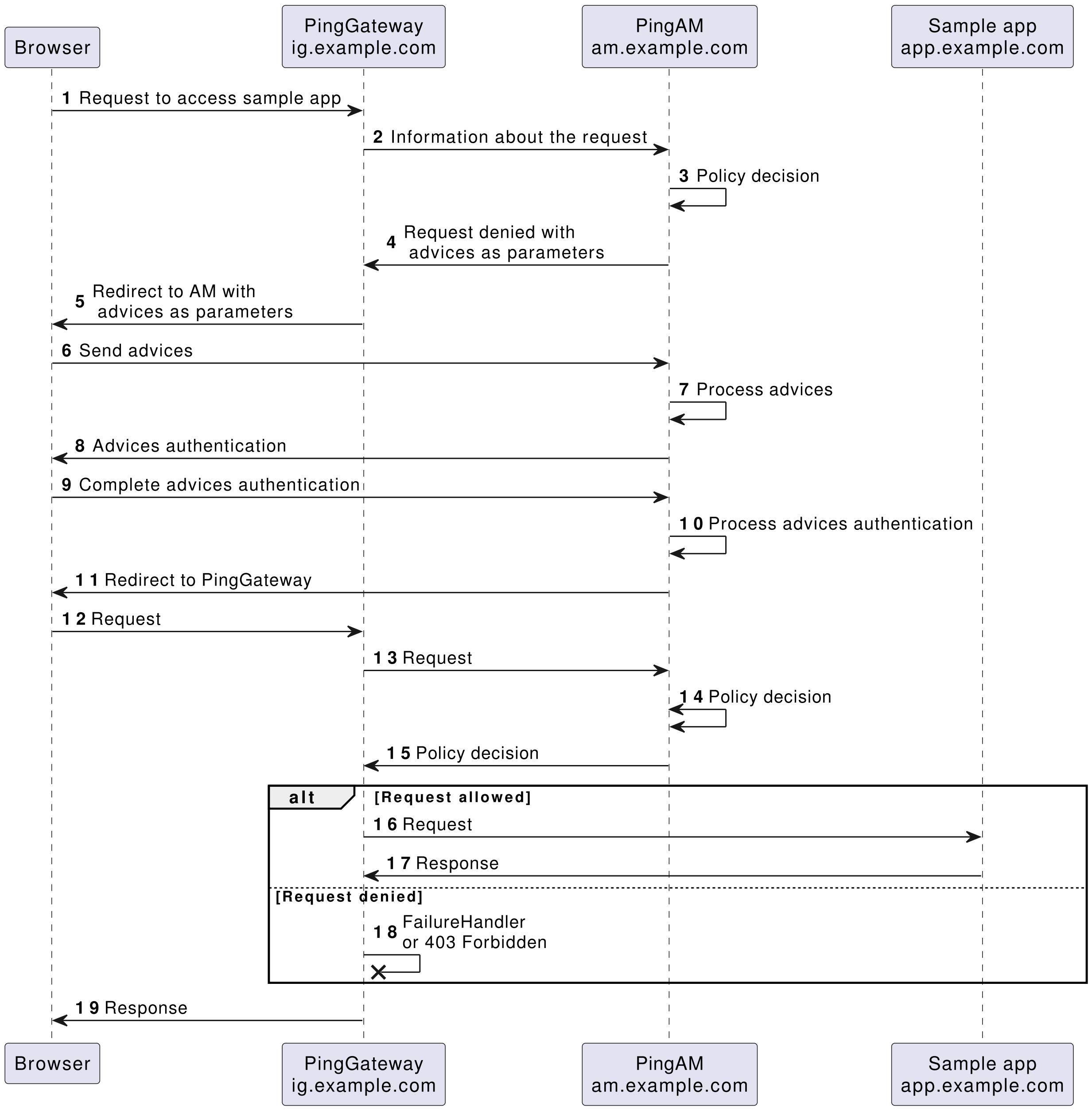
The following image shows a simplified flow of information when AM denies a request without advice.


Browser
PingGateway ig.example.com
PingAM am.example.com
Sample app app.example.com
1 Request to access sample app
2 Information about the request
3 Policy decision
4 Policy decision
alt [Request allowed]
5 Request
6 Response
7 Response
[Request denied without advices]
alt [FailureHandler configured]
8 FailureHandler
[No FailureHandler configured]
9 403 Forbidden

Deny requests with advice as parameters in a redirect response

The following image shows a simplified flow of information when AM denies a request with advice and PingGateway returns the advices as parameters in a redirect response.

This is the default flow, most used for web applications.



Browser
PingGateway ig.example.com
PingAM am.example.com
Sample app app.example.com
1 Request to access sample app
2 Information about the request
3 Policy decision
4 Request denied with advices as parameters
5 Redirect to AM with advices as parameters
6 Send advices
7 Process advices
8 Advices authentication
9 Complete advices authentication
10 Process advices authentication
11 Redirect to PingGateway
12 Request
13 Request
14 Policy decision
15 Policy decision
alt
[Request allowed]
16 Request
17 Response
[Request denied]
18 FailureHandler or 403 Forbidden
19 Response

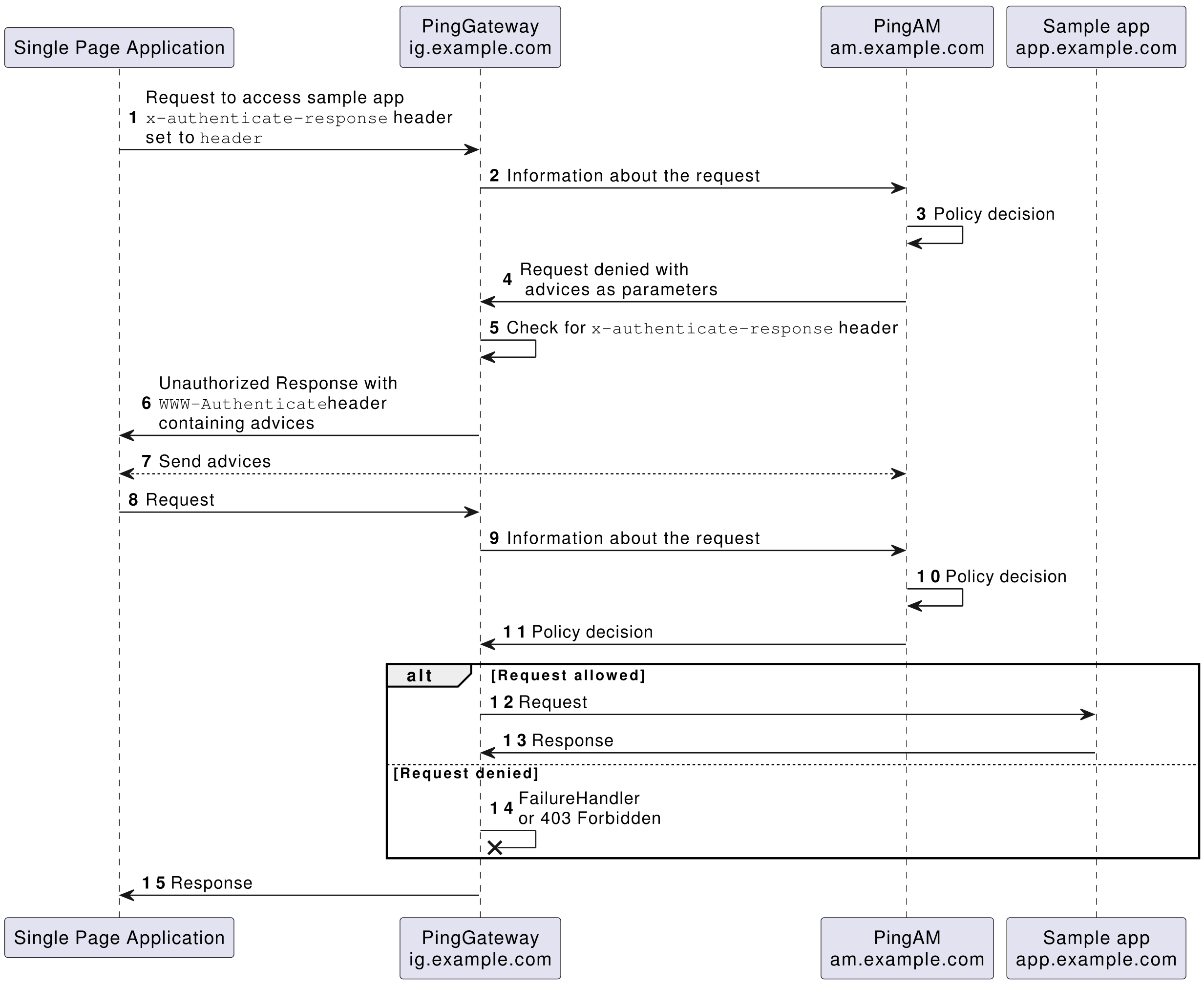
Deny requests with advice in a header

The following image shows a simplified flow of information when the request to PingGateway includes an `x-authenticate-response` header with the value `header`. If the header has any other value, the flow in Deny requests with advice as parameters in a redirect response takes place.

To change the name of the `x-authenticate-response` header, refer to the `authenticateResponseRequestHeader` property of the PolicyEnforcementFilter.

In this flow, AM denies the request with advice and PingGateway sends the response with the advice in the `WWW-authenticate` header.

Use this method for SDKs and single page applications. Placing advice in a header gives these applications more options for handling the advice.


Single Page Application
PingGateway ig.example.com
PingAM am.example.com
Sample app app.example.com
1 Request to access sample app x-authenticate-response header set to header
2 Information about the request
3 Policy decision
4 Request denied with advices as parameters
5 Check for x-authenticate-response header
6 Unauthorized Response with WWW-Authenticateheader containing advices
7 Send advices
8 Request
9 Information about the request
10 Policy decision
11 Policy decision
alt
[Request allowed]
12 Request
13 Response
[Request denied]
14 FailureHandler or 403 Forbidden
15 Response

Consider the following example GET with an `x-authenticate-response` header with the value `HEADER`:

```
[CONTINUED]GET https://ig.example.com:8443/home HTTP/1.1
[CONTINUED]accept-encoding: gzip, deflate
[CONTINUED]Connection: close
[CONTINUED]cookie: iPlanetDirectoryPro=0Dx...e3A.*....;
amlbcookie=01
[CONTINUED]Host: ig.example.com:8443
[CONTINUED]x-authenticate-response: HEADER
```

PingGateway returns a `WWW-Authenticate` header containing advice:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: SSOADVICE
realm="/",advices="eyJ...XX0=",am_uri="http://openam.example.com:8
080/am/"
transfer-encoding: chunked
connection: close
```

The advice decodes to a transaction condition advice:

```
{"TransactionConditionAdvice":["493...3c4"]}
```

Next steps

- Enforce AM policy decisions
  - Decisions in the same domain
  - Require authentication to a realm
  - Decisions in different domains
  - Decisions with claimsSubject
  - Notifications and the policy cache
- Harden authorization
  - Step up the authentication level
  - Authorize a single transaction

Enforce AM policy decisions

The following pages show how to use PingGateway to enforce AM policy decisions:

- Decisions in the same domain
- Require authentication to a realm
- Decisions in different domains
- Decisions with claimsSubject
- Notifications and the policy cache

Decisions in the same domain

The following procedure gives an example of how to create a policy in AM and configure an agent that can request policy decisions, when PingGateway and AM are in the same domain.

Before you start, set up and test the example in Use the default journey.

1. Set up AM:
   a. Select **Authorization** > **Policy Sets** > **New Policy Set**, and add a policy set with the following values:
      - **Id** : `PEP-SSO`

    - **Resource Types** : `URL`

b. In the policy set, add a policy with the following values:

    - **Name** : `PEP-SSO`
    - **Resource Type** : `URL`
    - **Resource pattern** : `*://*:*/*`
    - **Resource value** : `http://app.example.com:8081/home/pep-sso*`

      This policy protects the home page of the sample application.

c. On the **Actions** tab, add an action to allow HTTP `GET` .

d. On the **Subjects** tab, remove any default subject conditions, add a subject condition for all `Authenticated Users` .

2. Add the following route to PingGateway:

Linux Windows

```
$HOME/.openig/config/routes/04-pep.json
```

```
{
  "name": "pep-sso",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/pep-sso')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SingleSignOnFilter-1",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        },
        {
          "name": "PolicyEnforcementFilter-1",
          "type": "PolicyEnforcementFilter",
          "config": {
            "application": "PEP-SSO",
            "ssoTokenSubject": "${contexts.ssoToken.value}",
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

For information about how to set up the PingGateway route in Studio, refer to Policy enforcement in Structured Editor or Protecting a web app with Freeform Designer.

For an example route that uses `claimsSubject` instead of `ssoTokenSubject` to identify the subject, refer to Example policy enforcement using claimsSubject.

3. Test the setup:

   a. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/pep-sso.

   b. If you see warnings that the site isn't secure, respond to the warnings to access the site.

      Because you haven't previously authenticated to AM, the request does not contain a cookie with an SSO token. The SingleSignOnFilter redirects you to AM for authentication.

   c. Log in to AM as user `demo`, password `Ch4ng31t`.

      When you have authenticated, AM redirects you back to the request URL, and PingGateway requests a policy decision using the AM session cookie.

      AM returns a policy decision that grants access to the sample application.

Require authentication to a realm

This example creates a policy that requires users to authenticate in a specific realm.

To reduce the attack surface on the top level realm, ForgeRock advises you to create federation entities, agent profiles, authorizations, OAuth2/OIDC, and STS services in a subrealm. For this reason, the AM policy, AM agent, and services are in a subrealm.

1. Set up AM:
   a. In the AM admin UI, click **Realms** and add a realm named `alpha`. Leave all other values as default.

      For the rest of the steps in this procedure, make sure you are managing the alpha realm by checking that the **alpha** icon is displayed on the top left.
   b. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:
      - `https://ig.example.com:8443/*`
      - `https://ig.example.com:8443/*?*`
   c. Register a PingGateway agent with the following values, as described in Register a PingGateway agent in AM:
      - **Agent ID**: `ig_agent`
      - **Password**: `password`

        IMPORTANT

        Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.
   d. (Optional) Authenticate the agent to AM as described in Authenticate a PingGateway agent to AM.

      IMPORTANT

      PingGateway agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.
   e. Add a policy:
      i. Select **Authorization** > **Policy Sets** > **New Policy Set**, and add a policy set with the following values:
         - **Id** : `PEP-SSO-REALM`
         - **Resource Types** : `URL`

ii. In the policy set, add a policy with the following values:

- **Name** : `PEP-SSO-REALM`
- **Resource Type** : `URL`
- **Resource pattern** : `*://*:*/*`
- **Resource value** : `http://app.example.com:8081/home/pep-sso-realm`

  This policy protects the home page of the sample application.

iii. On the **Actions** tab, add an action to allow HTTP `GET` .

iv. On the **Subjects** tab, remove any default subject conditions, add a subject condition for all `Authenticated Users` .

v. On the **Environments** tab, add an environment condition that requires the user to authenticate to the ☁ **alpha** realm:

- **Type** : `Authentication to a Realm`
- **Authenticate to a Realm** : `/alpha`

2. Set up PingGateway:

   a. Set up PingGateway for HTTPS, as described in Configure PingGateway for TLS (server-side).

   b. Set an environment variable for the PingGateway agent password, and then restart PingGateway:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

   The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

   c. Add the following route to PingGateway to serve the sample application .css and other static resources:

Linux | Windows

```
$HOME/.openig/config/routes/00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or
matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

d. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/04-pep-sso-realm.json
```

```
{
  "name": "pep-sso-realm",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/pep-sso-
realm')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/",
        "realm": "/alpha"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SingleSignOnFilter-1",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        },
        {
          "name": "PolicyEnforcementFilter-1",
          "type": "PolicyEnforcementFilter",
          "config": {
            "application": "PEP-SSO-REALM",
            "ssoTokenSubject":
"${contexts.ssoToken.value}",
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following differences compared to `04-pep-sso.json`:

- The AmService is in the `alpha` realm. That means that the user authenticates to AM in that realm.
- The PolicyEnforcementFilter realm is not specified, so it takes the same value as the AmService realm. If refers to a policy in the AM `alpha` realm.

3. Test the setup:

   a. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/pep-sso-realm.

   b. If you see warnings that the site isn't secure, respond to the warnings to access the site.

   c. Log in to AM as user `demo`, password `Ch4ng31t`.

      Because you are authenticating in the `alpha` realm, AM returns a policy decision that grants access to the sample application.

      If you were to send the request from a different realm, AM would redirect the request with an `AuthenticateToRealmConditionAdvice`.

Decisions in different domains

The following procedure gives an example of how to create a policy in AM and configure an agent that can request policy decisions, when PingGateway and AM are in different domains.

Before you start, set up and test the example in Cross-domain single sign-on (CDSSO).

1. Set up AM:

   a. In the AM admin UI, select **Applications** > **Agents** > **Identity Gateway**, and change the redirect URL for `ig_agent_cdsso` :

      - **Redirect URL for CDSSO** : `https://ig.ext.com:8443/home/pep-cdsso/redirect`

   b. Select **Authorization** > **Policy Sets** > **New Policy Set**, and add a policy set with the following values:

      - **Id** : `PEP-CDSSO`
      - **Resource Types** : `URL`
        - In the new policy set, add a policy with the following values:
      - **Name** : `CDSSO`
      - **Resource Type** : `URL`
      - **Resource pattern** : `*://*:*/*`
      - **Resource value** : `http://app.example.com:8081/home/pep-cdsso*`

        This policy protects the home page of the sample application.
      - On the **Actions** tab, add an action to allow HTTP `GET` .
      - On the **Subjects** tab, remove any default subject conditions, add a subject condition for all `Authenticated Users` .

2. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/04-pep-cdsso.json
```

```
{
  "name": "pep-cdsso",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/pep-cdsso')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent_cdsso",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "CrossDomainSingleSignOnFilter-1",
          "type": "CrossDomainSingleSignOnFilter",
          "config": {
            "redirectEndpoint": "/home/pep-cdsso/redirect",
            "authCookie": {
              "path": "/home",
              "name": "ig-token-cookie"
            },
            "amService": "AmService-1"
          }
        },
        {
          "name": "PolicyEnforcementFilter-1",
          "type": "PolicyEnforcementFilter",
          "config": {
            "application": "PEP-CDSSO",
            "ssoTokenSubject": "${contexts.cdsso.token}",
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

WARNING

When `verificationSecretId` isn't configured, PingGateway discovers and uses the AM JWK set to verify the signature of AM session tokens. If the JWK set isn't available, PingGateway doesn't verify the tokens.

3. Test the setup:

    a. In your browser's privacy or incognito mode, go to to https://ig.ext.com:8443/home/pep-cdsso.

    b. If you see warnings that the site isn't secure, respond to the warnings to access the site.

       PingGateway redirects you to AM for authentication.

    c. Log in to AM as user `demo`, password `Ch4ng31t`.

       When you have authenticated, AM redirects you back to the request URL, and PingGateway requests a policy decision. AM returns a policy decision that grants access to the sample application.

Decisions with claimsSubject

This example extends Decisions in the same domain to enforce a policy decision from AM using the `claimsSubject` instead of `ssoTokenSubject` to identify the subject.

Before you start, set up and test the example in Decisions in the same domain.

1. Set up AM:

    a. Select the policy `PEP-SSO` and add a new resource:

    - **Resource Type**: `URL`
    - **Resource pattern**: `*://*:*/*`
    - **Resource value**: `http://app.example.com:8081/home/pep-claims`

    b. In the same policy, add the following subject condition:

    - `Any of`
    - **Type** : `OpenID Connect/JwtClaim`
    - **claimName** : `iss`
    - **claimValue** : `am.example.com`

2. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/04-pep-claims.json
```

```
{
  "name": "pep-claims",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/pep-
claims')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "http://am.example.com:8088/openam",
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SingleSignOnFilter-1",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        },
        {
          "name": "PolicyEnforcementFilter-1",
          "type": "PolicyEnforcementFilter",
          "config": {
            "application": "PEP-SSO",
            "claimsSubject": {
              "sub": "${contexts.ssoToken.info.uid}",
              "iss": "am.example.com"
            },
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

3. Test the setup:

   a. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/pep-claims.

   b. If you see warnings that the site isn't secure, respond to the warnings to access the site.

   c. Log in to AM as user `demo`, password `Ch4ng31t`.

      AM returns a policy decision that grants access to the sample application.

Notifications and the policy cache

When WebSocket notifications are enabled, PingGateway receives notifications whenever AM creates, deletes, or changes a policy.

The following procedure gives an example of how to change the configuration in Decisions in the same domain and Decisions in different domains to evict outdated entries from the policy cache. For information about WebSocket notifications, refer to WebSocket notifications.

Before you start, set up and test the example in Decisions in the same domain.

1. Websocket notifications are enabled by default. If they are disabled, enable them by adding the following configuration to the AmService in your route:

```
"notifications": {
  "enabled": true
}
```

2. Enable policy cache in the PolicyEnforcementFilter in your route:

```
"cache": {
  "enabled": true
}
```

3. In `logback.xml` add the following logger for WebSocket notifications, and then restart PingGateway:

```
<logger name="org.forgerock.openig.tools.notifications.ws"
level="TRACE" />
```

For information, refer to Changing the log level for different object types.

4. Test the setup:
   a. In your browser's privacy or incognito mode, go to https://ig.ext.com:8443/home/pep-sso.
   b. If you see warnings that the site isn't secure, respond to the warnings to access the site.
   c. Log in to AM as user `demo`, password `Ch4ng31t`.
   d. In a separate terminal, log on to AM as admin, and change the PEP-SSO policy. For example, in the **Actions** tab, add an action to allow HTTP `DELETE`.
   e. Note that the PingGateway system logs are updated with Websocket notifications about the change:

```
... | TRACE | vert.x-eventloop-thread-14 |
o.f.o.t.n.w.l.DirectAmLink | @system | Received a message:
... "policy": "PEP-SSO", "policySet": "PEP-SSO",
"eventType": "UPDATE" } }
... | TRACE | vert.x-eventloop-thread-14 |
o.f.o.t.n.w.SubscriptionService | @system | Notification
received, ... "policy": "PEP-SSO", "policySet": "PEP-SSO",
"eventType": "UPDATE" }}
... | TRACE | vert.x-eventloop-thread-14 |
o.f.o.t.n.w.SubscriptionService | @system | Notification
sent to a [/agent/policy] listener
```

Harden authorization

To protect sensitive resources, AM policies can use additional conditions to harden the authorization. When AM communicates a policy decision to PingGateway, the decision includes advice to indicate extra conditions the user must meet.

Conditions can include requirements to access the resource over a secure channel, access during working hours, or a higher authentication level. For more information, refer to AM's Authorization guide.

The following pages build on the policies to Enforce AM policy decisions:

- Step up the authentication level

- Authorize a single transaction

Step up the authentication level

When you step up the authentication level for an AM session, the authorization is verified and then captured as part of the AM session, and the user agent is authorized to that authentication level for the duration of the AM session.

This page uses the policies you created in Decisions in the same domain and Decisions in different domains, adding an authorization policy with an environment condition to step up the authentication level. Except for the paths where noted, procedures for single domain and cross-domain are the same.

After the user agent redirects the user to AM, if the user is not already authenticated they are presented with a login page. If the user is already authenticated, or after they authenticate, they are presented with a second page asking for an OTP verification code to step up the authentication level. You need a mobile device such as a phone to generate the OTP code.

Update AM settings

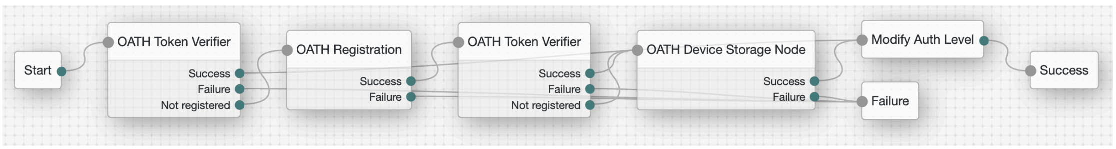
Before you start, set up one of the examples in Decisions in the same domain or Decisions in different domains.

1. In the AM admin UI, add a tree to step up authentication.

   a. Select **Authentication** > **Trees** > **+ Create Tree**.

   b. Name the new tree `StepUpAuthentication`.

   c. Add nodes to the tree as in the following image:



      - The **OATH Token Verifier node** has the default settings.

        It validates the user’s OTP verification code or leads the unregistered user to register an OTP generator application.

      - The **OATH Registration node** registers an OTP generator application for the user.

      - The second **OATH Token Verifier node** verifies an initial OTP after registration.

      - The **OATH Device Storage node** stores the OTP generator application setting in the user’s profile.

        - Update the Modify Auth Level node to set **Value to Add**: `1`, raising the authentication level.

    d. Click **Save**.

2. Update the policy to use the new authentication tree.

    a. Select a policy set:

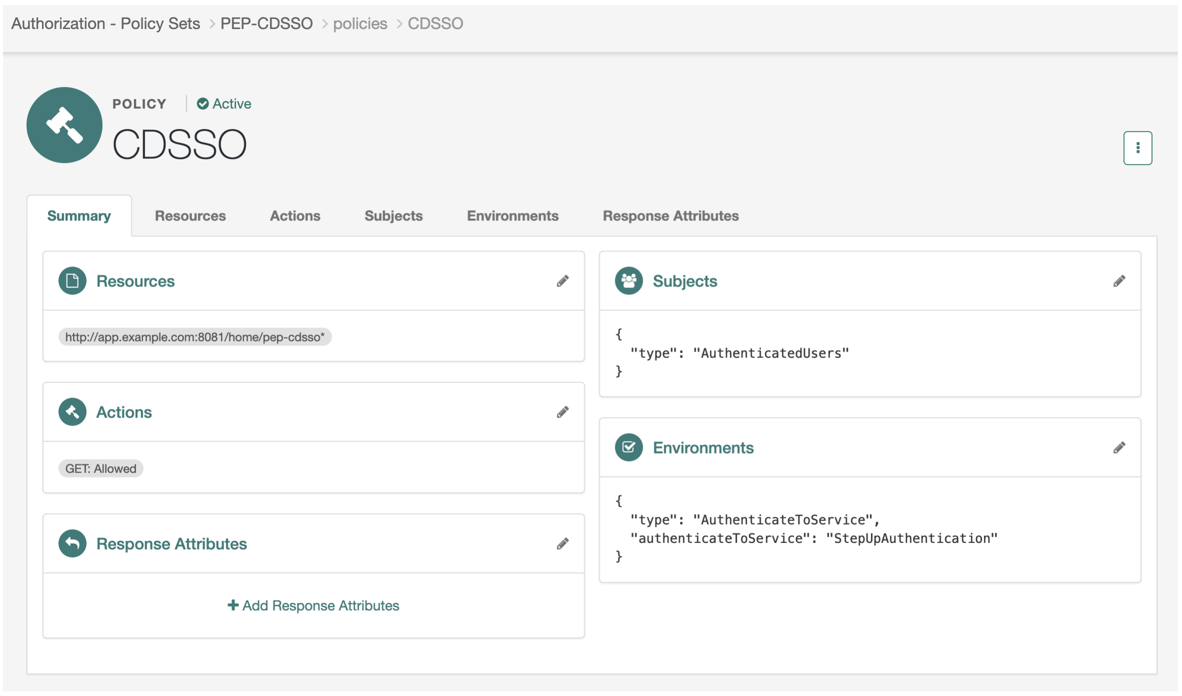
        - For SSO, select **Authorization** > **Policy Sets** > **PEP-SSO**.
        - For CDSSO, select **Authorization** > **Policy Sets** > **PEP-CDSSO**.

    b. In the policy, select **Environments** and add the following environment condition:

        - `All of`
        - **Type**: `Authentication by Service`
        - **Authenticate to Service**: `StepUpAuthentication`

    c. Click ✓ and **Save Changes**.

    The summary of the policy looks similar to the following image:



http://app.example.com:8081/home/pep-cdsso*

```
{
  "type": "AuthenticatedUsers"
}
```

GET: Allowed

```
{
  "type": "AuthenticateToService",
  "authenticateToService": "StepUpAuthentication"
}
```

Validation

1. In your browser's privacy or incognito mode, go to the appropriate URL:

    - For SSO, go to https://ig.example.com:8443/home/pep-sso.
    - For CDSSO, go to https://ig.ext.com:8443/home/pep-cdsso.

2. Log in to AM as user `demo`, password `Ch4ng31t`.

    AM creates a session with the default authentication level `0`, and PingGateway requests a policy decision.

The updated policy requires authentication level `1`, which is higher than the AM session's current authentication level. AM issues a redirect with a `AuthenticateToServiceConditionAdvice` to authenticate at level `1`.

3. If you haven't registered an OTP generator application, follow the instructions in your browser to register one.
4. Enter the OTP verification code from the application you registered on your device.

   AM upgrades the authentication level for the session to 1, and grants access to the sample application. If you try to access the sample application again in the same session, you don't need to provide the verification code.

Authorize a single transaction

Transactional authorization requires a user to perform additional actions for one-time access to a resource.

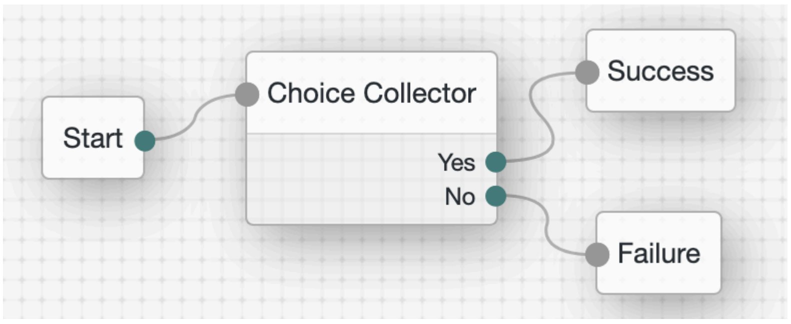
Performing the additional action successfully grants access to the protected resource, but only once. Additional attempts to access the resource require the user to perform the configured actions again.

This section builds on the example in Step up the authentication level, adding a simple authorization policy with a `Transaction` environment condition. Each time the user agent tries to access the protected resource, they confirm the transaction again.

Update AM settings

Before you start, configure AM as described in Step up the authentication level. The PingGateway configuration is not changed.

1. In the AM admin UI, add a tree to confirm the transaction.
   a. Select **Authentication** > **Trees** > **+ Create Tree**.
   b. Name the new tree `ConfirmTransaction`.
   c. Set up the tree as in the following image:



   The **Choice Collector node** has these settings:

- **Choices**: `Yes` and `No`
- **Default Choice**: `No`
- **Prompt**: `Confirm transaction?`

d. Click **Save**.

2. Update the policy to use the new authentication tree.

a. Select the policy set:

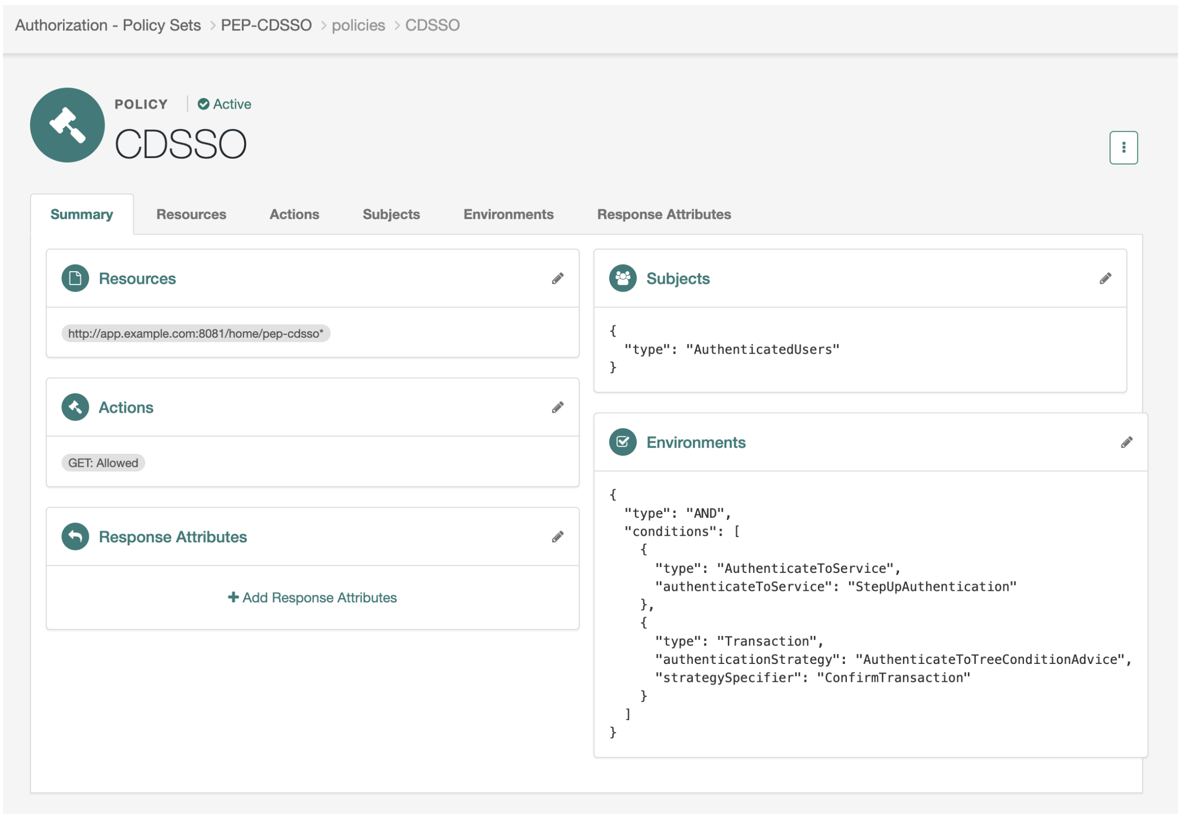
- For SSO, select **Authorization** > **Policy Sets** > **PEP-SSO**.
- For CDSSO, select **Authorization** > **Policy Sets** > **PEP-CDSSO**.

b. In the policy, select **Environments** and add another environment condition:

- `All of`
- **Type**: `Transaction`
- **Authentication strategy**: `Authenticate To Tree`
- **Strategy specifier**: `ConfirmTransaction`

c. Click ✓ and **Save Changes**.

The summary of the policy looks similar to the following image:



```
{
  "type": "AuthenticatedUsers"
}
```

```
{
  "type": "AND",
  "conditions": [
    {
      "type": "AuthenticateToService",
      "authenticateToService": "StepUpAuthentication"
    },
    {
      "type": "Transaction",
      "authenticationStrategy": "AuthenticateToTreeConditionAdvice",
      "strategySpecifier": "ConfirmTransaction"
    }
  ]
}
```

Validation

1. In your browser's privacy or incognito mode, go to the appropriate URL:

   ◦ For SSO, go to https://ig.example.com:8443/home/pep-sso.

   - For CDSSO, go to https://ig.ext.com:8443/home/pep-cdsso.

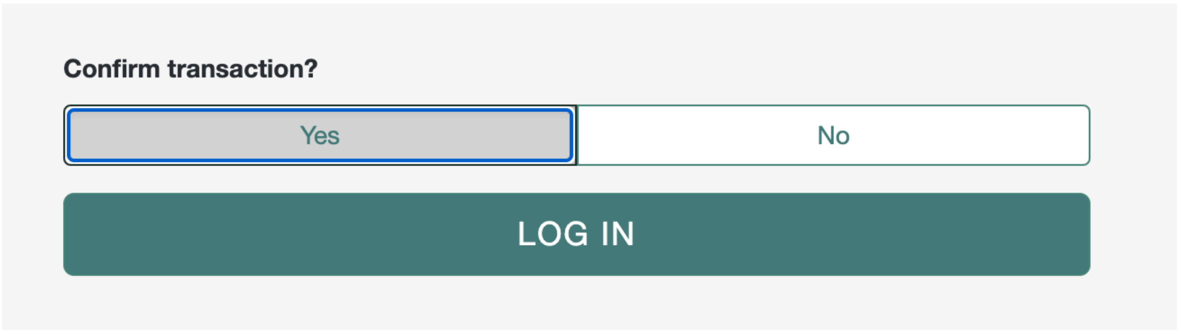
2. Log in to AM as user `demo`, password `Ch4ng31t`.

   AM creates a session with the default authentication level `0`, and PingGateway requests a policy decision.

3. Enter the OTP verification code from the application you registered on your device.

   AM steps up the authentication level and displays a `Confirm transaction?` choice.

4. Confirm the transaction by selecting **Yes** and logging in.



   AM returns a policy decision granting one-time access to the sample application. If you reload the sample application page, you must confirm the new transaction.

OAuth 2.0

As an authorization server, PingGateway *authenticates* resource owners and obtains their *authorization* to return access tokens to clients.

Before you configure OAuth 2.0 in your environment, familiarize yourself with the OAuth 2.0 authorization framework and related standards.

OAuth 2.0 concepts

RFC 6749, the OAuth 2.0 authorization framework lets a third-party application obtain limited access to a resource (usually user data) on behalf of the resource owner or the application itself.

The main actors in the OAuth 2.0 authorization framework are the following:

Actor	Description
Resource owner (RO)	The owner of the resource. For example, a user who stores their photos in a photo-sharing service. The resource owner uses a *user-agent*, usually a web-browser, to communicate with the client.
Client	The third-party application that wants to access the resource. The client makes requests on behalf of the resource owner and with their authorization. For example, a printing service that needs to access the resource owner's photos to print them. Learn more from PingGateway as an OAuth 2.0 client.
Authorization server (AS)	The authorization service that authenticates the resource owner and/or the client, issues access tokens to the client, and tracks their validity. Access tokens prove that the resource owner authorizes the client to act on their behalf over specific resources for a limited period of time. PingOne Advanced Identity Cloud or PingAM can act as an authorization server.
Resource server (RS)	The service hosting the protected resources. For example, a photo-sharing service. The resource server must be able to validate the tokens issued by the authorization server. Learn more from PingGateway as an OAuth 2.0 resource server.

PingGateway as an OAuth 2.0 client

PingGateway as an OAuth 2.0 client supports the OAuth 2.0 filters and flows in the following table:

Filter	OAuth 2.0 flow	Description
AuthorizationCodeOAuth2ClientFilter (previously named OAuth2ClientFilter)	Authorization Code Grant	This filter requires the user agent to authorize the request interactively to obtain an access token and optional ID token. The access token is maintained only for the OAuth 2.0 session, and is valid only for the configured scopes. This filter can act as an OpenID Connect relying party or as an OAuth 2.0 client. Use for Web applications running on a server.
ResourceOwnerOAuth2ClientFilter	Resource Owner Password Credentials Grant	According to information in the The OAuth 2.0 Authorization Framework, minimize use of this grant type and use other grant types when possible. This filter supports the transformation of client credentials and user credentials to obtain an access token from the Authorization Server. It injects the access token into the inbound request as a Bearer Authorization header. The access token is valid only for the configured scopes. Use for clients trusted with the resource owner credentials.
ClientCredentialsOAuth2ClientFilter	Client Credentials Grant	This filter is similar to the Resource Owner Password Credentials grant type, but the resource owner is not part of the flow and the client accesses only information relevant to itself. Use when the client is the resource owner, or the client doesn’t act on behalf of the resource owner.

PingGateway as an OAuth 2.0 resource server

The following image illustrates the steps for a client application to access a user's protected resources, with AM as the Authorization Server and PingGateway as the resource server:

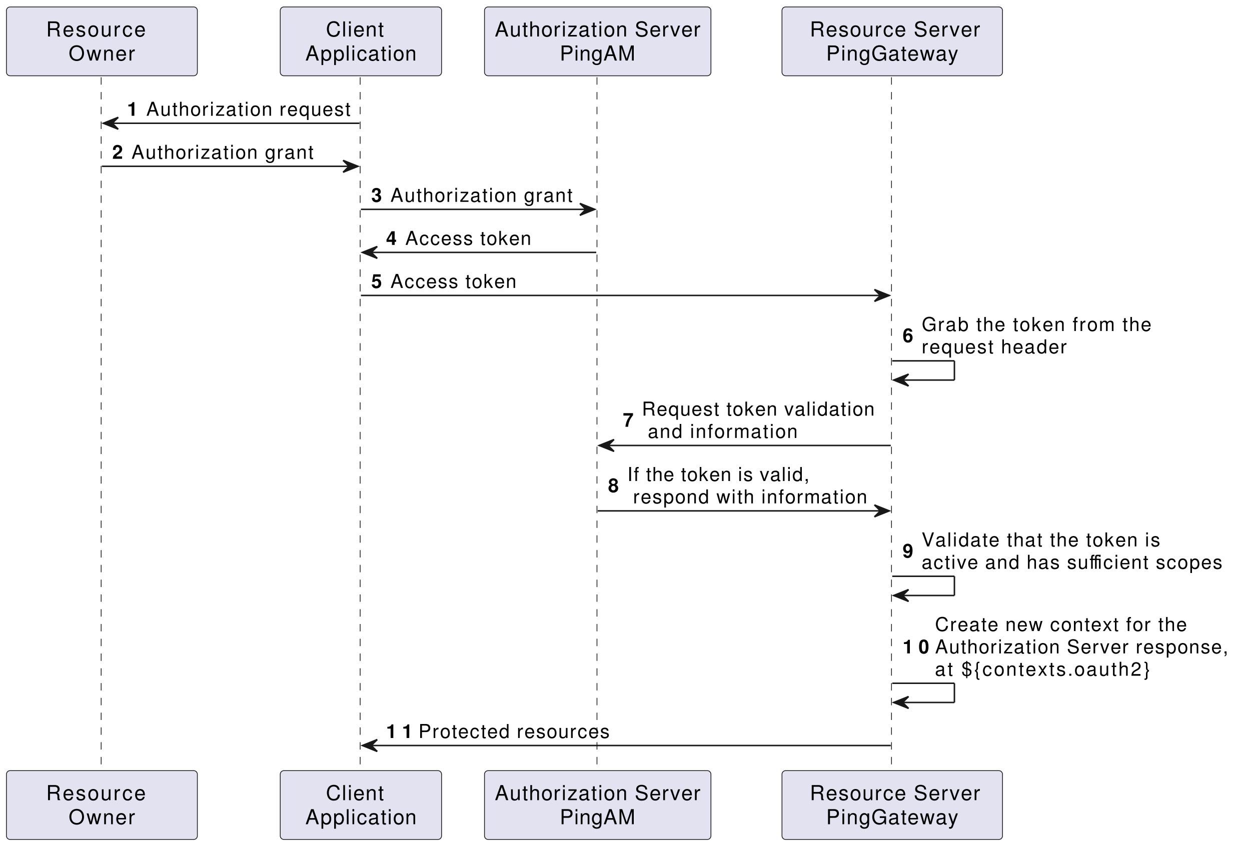

Resource Owner
Client Application
Authorization Server PingAM
Resource Server PingGateway
1 Authorization request
2 Authorization grant
3 Authorization grant
4 Access token
5 Access token
6 Grab the token from the request header
7 Request token validation and information
8 If the token is valid, respond with information
9 Validate that the token is active and has sufficient scopes
10 Create new context for the Authorization Server response, at ${contexts.oauth2}
11 Protected resources
Resource Owner
Client Application
Authorization Server PingAM
Resource Server PingGateway

Figure 2. PingGateway as an OAuth 2.0 resource server handling OAuth 2.0 requests

- The application obtains an *authorization grant*, representing the resource owner's consent. Learn more about the different OAuth 2.0 grant mechanisms supported by AM in the AM documentation on OAuth 2.0 grant flows.
- The application authenticates to the Authorization Server and requests an *access token*. The Authorization Server returns an access token to the application.

  An OAuth 2.0 access token is an opaque string issued by the authorization server. When the client interacts with the resource server, the client presents the access token in the `Authorization` header. For example:

  ```
  Authorization: Bearer 7af...da9
  ```

  Access tokens are the credentials to access protected resources. The advantage of access tokens over passwords or other credentials is that access tokens can be granted and revoked without exposing the user's credentials.

  The access token represents the authorization to access protected resources. Because an access token is a bearer token, anyone who has the access token can use it to get the resources. Access tokens must therefore be protected, so that requests involving them go over HTTPS.

In OAuth 2.0, the token scopes are strings that identify the scope of access authorized to the client, but can also be used for other purposes.

- The application supplies the access token to the resource server, which then resolves and validates the access token by using an access token resolver, as described in Access token resolvers.

  If the access token is valid, the resource server permits the client access to the requested resource.

The OAuth2ResourceServerFilter grants access to a resource by using an OAuth 2.0 access token from the HTTP Authorization header of a request.

When auditing is enabled, OAuth 2.0 token tracking IDs can be logged in access audit events for routes that contain an OAuth2ResourceServerFilter. Learn more in Audit the deployment and Audit framework.

Next steps

- Validate access tokens with introspection
- Script required scopes
- Validate stateless access tokens
  - With JwkSetSecretStore
  - Signed tokens with KeyStoreSecretStore
  - Encrypted tokens with KeyStoreSecretStore
- Mutual TLS
  - mTLS with client certificates
  - mTLS with trusted headers
- OAuth 2.0 context for authentication
- Cache access tokens
- Client credentials grant
- Resource owner password credentials grant

Validate access tokens with introspection

This page sets up PingGateway as an OAuth 2.0 resource server using the introspection endpoint.

You can find more information about configuring AM as an OAuth 2.0 authorization service in the AM OAuth 2.0 documentation.

IMPORTANT

This procedure uses the *Resource Owner Password Credentials* grant type. As suggested in The OAuth 2.0 Authorization Framework, use other grant types whenever possible.

Before you begin, prepare AM, PingGateway, and the sample application. Learn more in the example installation for this guide.

1. Set up AM:
   a. Select **Applications** > **Agents** > **Identity Gateway**, and register a PingGateway agent with the following values:
      - **Agent ID**: `ig_agent`
      - **Password**: `password`
      - **Token Introspection**: `Realm Only`

      IMPORTANT

      Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

   b. (Optional) Authenticate the agent to AM as described in Authenticate a PingGateway agent to AM.

      IMPORTANT

      PingGateway agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

   c. Create an OAuth 2.0 Authorization Server:
      i. Select **Services** > **Add a Service** > **OAuth2 Provider**.
      ii. Add a service with the default values.

   d. Create an OAuth 2.0 Client to request OAuth 2.0 access tokens:
      i. Select **Applications** > **OAuth 2.0** > **Clients**, and add a client with the following values:
         - **Client ID**: `client-application`
         - **Client secret**: `password`
         - **Scope(s)**: `mail` , `employeenumber`
      ii. (Optional) On the **Core** tab, switch to using a client secret associated with a secret label by setting a **Secret Label Identifier** and mapping the label to a secret.

To learn more, read Create a client profile and Map and rotate secrets in the AM documentation.

iii. On the **Advanced** tab, select the following value:

- **Grant Types**: `Resource Owner Password Credentials`

2. Set up PingGateway

   a. Set up PingGateway for HTTPS, as described in Configure PingGateway for TLS (server-side).

   b. Set an environment variable for the PingGateway agent password, and then restart PingGateway:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

   c. Add the following route to PingGateway:

Linux Windows

```
$HOME/.openig/config/routes/rs-introspect.json
```

```
{
  "name": "rs-introspect",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/rs-
introspect$')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [
              "mail",
              "employeenumber"
            ],
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name":
"TokenIntrospectionAccessTokenResolver-1",
              "type":
"TokenIntrospectionAccessTokenResolver",
              "config": {
                "amService": "AmService-1",
                "providerHandler": {
                  "type": "Chain",
                  "config": {
                    "filters": [
                      {
                        "type":
"HttpBasicAuthenticationClientFilter",
                        "config": {
                          "username": "ig_agent",
                          "passwordSecretId":
"agent.secret.id",
                          "secretsProvider":
"SystemAndEnvSecretStore-1"
                        }
                      }
                    ],
                    "handler": "ForgeRockClientHandler"
                  }
                }
              }
            }
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": "<html><body><h2>Decoded access_token:
${contexts.oauth2.accessToken.info}</h2></body></html>"
        }
      }
    }
  }
}
```

You can find more information about how to set up the PingGateway route in Token validation using the introspection endpoint in Structured Editor.

Notice the following features of the route:

- The route matches requests to `/rs-introspect`.
- The `OAuth2ResourceServerFilter` expects an OAuth 2.0 access token in the `authorization` header of the incoming authorization request, with the scopes `mail` and `employeenumber`.

  The `accessTokenResolver` uses the AM server declared in the heap. The introspection endpoint to validate the access token is extrapolated from the URL of the AM server.

  For convenience in this test, `requireHttps` is false. In production environments, set it to true.
- After the filter validates the access token, it creates a new context from the Authorization Server response. The context is named `oauth2`, and can be reached at `contexts.oauth2` or `contexts['oauth2']`.

  The context contains information about the access token, which can be reached at `contexts.oauth2.accessToken.info`. Filters and handlers further down the chain can access the token info through the context.

  If there is no access token in the request, or token validation does not complete successfully, the filter returns an HTTP error status to the user agent, and PingGateway doesn't continue processing the request. This is done as specified in the RFC, The OAuth 2.0 Authorization Framework: Bearer Token Usage.

        - The HttpBasicAuthenticationClientFilter adds the credentials to the outgoing token introspection request.
        - The StaticResponseHandler returns the content of the access token from the context `${contexts.oauth2.accessToken.info}`.

3. Test the setup:

    a. In a terminal window, use a **`curl`** command similar to the following to retrieve an access token:

    ```
    $ mytoken=$(curl -s \
    --user "client-application:password" \
    --data
    "grant_type=password&username=demo&password=Ch4ng31t&scope
    =mail%20employeenumber" \
    http://am.example.com:8088/openam/oauth2/access_token | jq
    -r ".access_token")
    ```

    b. Validate the access token returned in the previous step:

    ```
    $ curl -v \
    --cacert /path/to/secrets/ig.example.com-certificate.pem \
    --header "Authorization: Bearer ${mytoken}" \
    https://ig.example.com:8443/rs-introspect

    {
      active = true,
      scope = employeenumber mail,
      realm=/,
      client_id = client-application,
      user_id = demo,
      token_type = Bearer,
      exp = 158...907,
      ...
    }
    ```

Script required scopes

This example builds on the example in Validate access tokens with introspection to use a script to define the scopes that a request requires in an access token.

- If the request path is `/rs-tokeninfo`, the request requires only the scope `mail`.
- If the request path is `/rs-tokeninfo/employee`, the request requires the scopes `mail and employeenumber`.

Before you start, set up and test the example in Validate access tokens with introspection.

1. Add the following route to PingGateway:

   Linux | Windows

   ```
   $HOME/.openig/config/routes/rs-dynamicscope.json
   ```

   ```
   {
     "name": "rs-dynamicscope",
     "baseURI": "http://app.example.com:8081",
     "condition": "${find(request.uri.path, '^/rs-
   dynamicscope')}",
     "heap": [
       {
         "name": "SystemAndEnvSecretStore-1",
         "type": "SystemAndEnvSecretStore"
       },
       {
         "name": "AmService-1",
         "type": "AmService",
         "config": {
           "agent": {
             "username": "ig_agent",
             "passwordSecretId": "agent.secret.id"
           },
           "secretsProvider": "SystemAndEnvSecretStore-1",
           "url": "http://am.example.com:8088/openam/"
         }
       }
     ],
     "handler": {
       "type": "Chain",
       "config": {
         "filters": [
           {
             "name": "OAuth2ResourceServerFilter-1",
             "type": "OAuth2ResourceServerFilter",
             "config": {
               "scopes": {
                 "name": "myscript",
                 "type": "ScriptableResourceAccess",
                 "config": {
   ```

```
                "type": "application/x-groovy",
                "source": [
                  "// Minimal set of required scopes",
                  "def scopes = [ 'mail' ] as Set",
                  "if (request.uri.path =~ /employee$/) {",
                  "  // Require another scope to access this
resource",
                  "  scopes += 'employeenumber'",
                  "}",
                  "return scopes"
                ]
              }
            },
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name": "token-resolver-1",
              "type": "TokenIntrospectionAccessTokenResolver",
              "config": {
                "amService": "AmService-1",
                "providerHandler": {
                  "type": "Chain",
                  "config": {
                    "filters": [
                      {
                        "type":
"HttpBasicAuthenticationClientFilter",
                        "config": {
                          "username": "ig_agent",
                          "passwordSecretId":
"agent.secret.id",
                          "secretsProvider":
"SystemAndEnvSecretStore-1"
                        }
                      }
                    ],
                    "handler": "ForgeRockClientHandler"
                  }
                }
              }
            }
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": "<html><body><h2>Decoded access_token:
${contexts.oauth2.accessToken.info}</h2></body></html>"
        }
      }
    }
  }
}
```

2. Test the setup with the `mail` scope only:

   a. In a terminal, use a **`curl`** command to retrieve an access token with the scope `mail`:

```
$ mytoken=$(curl -s \
--user "client-application:password" \
--data
"grant_type=password&username=demo&password=Ch4ng31t&scope
=mail" \
http://am.example.com:8088/openam/oauth2/access_token | jq
-r ".access_token")
```

   b. Confirm that the access token is returned for the `/rs-dynamicscope` path:

```
$ curl -v \
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--header "Authorization: Bearer ${mytoken}" \
https://ig.example.com:8443/rs-dynamicscope

{
  active = true,
  scope = mail,
  client_id = client-application,
  user_id = demo,
  token_type = Bearer,
  exp = 158...907,
  sub = demo,
  iss = http://am.example.com:8088/openam/oauth2, ...
  ...
}
```

c. Confirm that the access token **is not** returned for the `/rs-dynamicscope/employee` path:

```
$ curl -v \
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--header "Authorization: Bearer ${mytoken}" \
https://ig.example.com:8443/rs-dynamicscope/employee
```

3. Test the setup with the scopes `mail` and `employeenumber`:

a. In a terminal window, use a **`curl`** command similar to the following to retrieve an access token with the scopes `mail` and `employeenumber`:

```
$ mytoken=$(curl -s \
--user "client-application:password" \
--data
"grant_type=password&username=demo&password=Ch4ng31t&scope
=mail%20employeenumber" \
http://am.example.com:8088/openam/oauth2/access_token | jq
-r ".access_token")
```

b. Confirm that the access token **is** returned for the `/rs-dynamicscope/employee` path:

```
$ curl -v
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--header "Authorization: Bearer ${mytoken}"
https://ig.example.com:8443/rs-dynamicscope/employee
```

Validate stateless access tokens

The StatelessAccessTokenResolver confirms that stateless access tokens provided by AM are well-formed, have a valid issuer, have the expected access token name, and have a valid signature.

After the StatelessAccessTokenResolver resolves an access token, the OAuth2ResourceServerFilter checks that the token is within the expiry time, and that it provides the required scopes. For more information, refer to StatelessAccessTokenResolver.

The following pages show how to validate signed and encrypted access tokens:

- With JwkSetSecretStore
- Signed tokens with KeyStoreSecretStore

- Encrypted tokens with KeyStoreSecretStore

With JwkSetSecretStore

This page shows how to validate signed access tokens with the StatelessAccessTokenResolver using a JwkSetSecretStore.

IMPORTANT

This procedure uses the *Resource Owner Password Credentials* grant type. As suggested in The OAuth 2.0 Authorization Framework, use other grant types whenever possible.

1. Set up AM:
   a. Configure an OAuth 2.0 Authorization Provider:
      i. Select **Services**, and add an OAuth 2.0 Provider.
      ii. Accept the default values and select **Create**. The service is added to the **Services** list.
      iii. On the **Core** tab, select the following option:
         - **Use Client-Based Access & Refresh Tokens** : `on`
      iv. On the **Advanced** tab, select the following options:
         - **Client Registration Scope Allowlist** : `myscope`
         - **OAuth2 Token Signing Algorithm** : `RS256`
         - **Encrypt Client-Based Tokens** : Deselected
   b. Create an OAuth2 Client to request OAuth 2.0 access tokens:
      i. Select **Applications** > **OAuth 2.0** > **Clients**, and add a client with the following values:
         - **Client ID** : `client-application`
         - **Client secret** : `password`
         - **Scope(s)** : `myscope`
      ii. (Optional) On the **Core** tab, switch to using a client secret associated with a secret label by setting a **Secret Label Identifier** and mapping the label to a secret.

         To learn more, read Create a client profile and Map and rotate secrets in the AM documentation.
      iii. On the **Advanced** tab, select the following values:
         - **Grant Types** : `Resource Owner Password Credentials`
         - **Response Types** : `code token`

iv. On the **Signing and Encryption** tab, include the following setting:

- **ID Token Signing Algorithm** : `RS256`

2. Set up PingGateway:

a. Set up PingGateway for HTTPS, as described in Configure PingGateway for TLS (server-side).

b. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/rs-stateless-signed.json
```

```
{
  "name": "rs-stateless-signed",
  "condition": "${find(request.uri.path, '/rs-stateless-
signed')}",
  "heap": [
    {
      "name": "SecretsProvider-1",
      "type": "SecretsProvider",
      "config": {
        "stores": [
          {
            "type": "JwkSetSecretStore",
            "config": {
              "jwkUrl":
"http://am.example.com:8088/openam/oauth2/connect/jwk_uri"
            }
          }
        ]
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "capture": "all",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": ["myscope"],
            "requireHttps": false,
            "accessTokenResolver": {
              "type": "StatelessAccessTokenResolver",
              "config": {
                "secretsProvider": "SecretsProvider-1",
                "issuer":
"http://am.example.com:8088/openam/oauth2",
                "verificationSecretId":
"any.value.in.regex.format"
              }
            }
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": "<html><body><h2>Decoded access_token:
${contexts.oauth2.accessToken.info}</h2></body></html>"
        }
      }
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/rs-stateless-signed`.
- A SecretsProvider in the heap declares a JwkSetSecretStore to manage secrets for signed access tokens.
- The JwkSetSecretStore specifies the URL to a JWK set on AM, that contains the signing keys.
- The OAuth2ResourceServerFilter expects an OAuth 2.0 access token in the header of the incoming authorization request, with the scope `myscope`.
- The StatelessAccessTokenResolver uses the SecretsProvider to verify the signature of the provided access token.
- After the OAuth2ResourceServerFilter validates the access token, it creates the OAuth2Context context. You can find more information in OAuth2Context.

- If there is no access token in a request, or token validation doesn't complete successfully, the filter returns an HTTP error status to the user agent, and PingGateway doesn't continue processing the request. This is done as specified in the RFC The OAuth 2.0 Authorization Framework: Bearer Token Usage.
- The StaticResponseHandler returns the content of the access token from the context.

3. Test the setup for a signed access token:
   a. Get an access token for the demo user, using the scope `myscope`:

   ```
   $ mytoken=$(curl -s \
   --user "client-application:password" \
   --data
   "grant_type=password&username=demo&password=Ch4ng31t&scope
   =myscope" \
   http://am.example.com:8088/openam/oauth2/access_token | jq
   -r ".access_token")
   ```

   b. Display the token:

   ```
   $ echo ${mytoken}
   ```

   Note that the token is structured as a signed token.

   c. Access the route by providing the token returned in the previous step:

   ```
   $ curl -v \
   --cacert /path/to/secrets/ig.example.com-certificate.pem \
   --header "Authorization: Bearer ${mytoken}" \
   https://ig.example.com:8443/rs-stateless-signed

   ...
       Decoded access_token: {
       sub=(usr!demo),
       cts=OAUTH2_STATELESS_GRANT,
       ...
   ```

Signed tokens with KeyStoreSecretStore

This page shows how to validate signed access tokens with the StatelessAccessTokenResolver using a KeyStoreSecretStore.

Set up signing keys

1. Locate the following directories for keys, keystores, and certificates, and in a terminal create variables for them:
   - Directory where the keystore is created: `keystore_directory`
   - AM keystore directory: `am_keystore_directory`
   - PingGateway keystore directory: `ig_keystore_directory`
2. Set up the keystore for signing keys:
   a. Generate a private key called `signature-key`, and a corresponding public certificate called `x509certificate.pem`:

   ```
   $ openssl req -x509 \
   -newkey rsa:2048 \
   -nodes \
   -subj
   "/CN=ig.example.com/OU=example/O=com/L=fr/ST=fr/C=fr" \
   -keyout $keystore_directory/signature-key.key \
   -out $keystore_directory/x509certificate.pem \
   -days 365

   ...
   writing new private key to '$keystore_directory/signature-
   key.key'
   ```

   b. Convert the private key and certificate files into a PKCS#12 file, called `signature-key`, and store them in a keystore named `keystore.p12`:

   ```
   $ openssl pkcs12 \
   -export \
   -in $keystore_directory/x509certificate.pem \
   -inkey $keystore_directory/signature-key.key \
   -out $keystore_directory/keystore.p12 \
   -passout pass:password \
   -name signature-key
   ```

   c. List the keys in `keystore.p12`:

   ```
   $ keytool -list \
   -v \
   -keystore "$keystore_directory/keystore.p12" \
   -storepass "password" \
   -storetype PKCS12
   ```

```
...
Your keystore contains 1 entry
Alias name: signature-key
```

3. Set up keys for AM:

   a. Copy the signing key `keystore.p12` to AM:

   ```
   $ cp $keystore_directory/keystore.p12
   $am_keystore_directory/AM_keystore.p12
   ```

   b. List the keys in the AM keystore:

   ```
   $ keytool -list \
   -v \
   -keystore "$am_keystore_directory/AM_keystore.p12" \
   -storepass "password" \
   -storetype PKCS12

   ...
   Your keystore contains 1 entry
   Alias name: signature-key
   ```

   c. Add a file called `keystore.pass`, containing the store password `password`:

   ```
   $ cd $am_keystore_directory
   $ echo -n 'password' > keystore.pass
   ```

   NOTE

   Make sure the password file contains only the password, with no trailing spaces or carriage returns.

   The filename corresponds to the secret ID of the store password and entry password for the KeyStoreSecretStore.

   d. Restart AM.

4. Set up keys for PingGateway:

   a. Import the public certificate to the IG keystore, with the alias `verification-key`:

   ```
   $ keytool -import \
   -trustcacerts \
   -rfc \
   -alias verification-key \
   ```

```
-file "$keystore_directory/x509certificate.pem" \
-keystore "$ig_keystore_directory/IG_keystore.p12" \
-storetype PKCS12 \
-storepass "password"

...
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

b. List the keys in the PingGateway keystore:

```
$ keytool -list \
-v \
-keystore "$ig_keystore_directory/IG_keystore.p12" \
-storepass "password" \
-storetype PKCS12

...
Your keystore contains 1 entry
Alias name: verification-key
```

c. In the PingGateway configuration, set an environment variable for the keystore password:

```
$ export KEYSTORE_SECRET_ID='cGFzc3dvcmQ='
```

d. Restart PingGateway.

Validate tokens

IMPORTANT

This procedure uses the *Resource Owner Password Credentials* grant type. As suggested in The OAuth 2.0 Authorization Framework, use other grant types whenever possible.

1. Set up AM:

   a. Create a KeyStoreSecretStore to manage the new AM keystore:

      i. In AM, select **Secret Stores**, and then add a secret store with the following values:

         - **Secret Store ID** : `keystoresecretstore`
         - **Store Type** : `Keystore`
         - **File** : `am_keystore_directory/AM_keystore.p12`

      - **Keystore type** : `PKCS12`
      - **Store password secret label** : `keystore.pass`
      - **Entry password secret label** : `keystore.pass`

   ii. Select the **Mappings** tab, and add a mapping with the following values:

      - **Secret Label** : `am.services.oauth2.stateless.signing.RSA`
      - **Aliases** : `signature-key`

        The mapping sets `signature-key` as the active alias to use for signature generation.

b. Create a FileSystemSecretStore to manage secrets for the KeyStoreSecretStore:

   i. Select **Secret Stores**, and then create a secret store with the following configuration:

      - **Secret Store ID** : `filesystemsecretstore`
      - **Store Type** : `File System Secret Volumes`
      - **Directory** : `am_keystore_directory`
      - **File format** : `Plain text`

c. Configure an OAuth 2.0 Authorization Provider:

   i. Select **Services**, and add an OAuth 2.0 Provider.

   ii. Accept all of the default values, and select **Create**. The service is added to the **Services** list.

   iii. On the **Core** tab, select the following option:

      - **Use Client-Based Access & Refresh Tokens** : `on`

   iv. On the **Advanced** tab, select the following options:

      - **Client Registration Scope Allowlist** : `myscope`
      - **OAuth2 Token Signing Algorithm** : `RS256`
      - **Encrypt Client-Based Tokens** : Deselected

d. Create an OAuth2 Client to request OAuth 2.0 access tokens:

   i. Select **Applications** > **OAuth 2.0** > **Clients**, and add a client with the following values:

      - **Client ID** : `client-application`
      - **Client secret** : `password`
      - **Scope(s)** : `myscope`

   ii. (Optional) On the **Core** tab, switch to using a client secret associated with a secret label by setting a **Secret Label Identifier** and mapping the label to a secret.

To learn more, read Create a client profile and Map and rotate secrets in the AM documentation.

iii. On the **Advanced** tab, select the following values:

- **Grant Types** : `Resource Owner Password Credentials`
- **Response Types** : `code token`

iv. On the **Signing and Encryption** tab, include the following setting:

- **ID Token Signing Algorithm** : `RS256`

2. Set up PingGateway:

a. Set up PingGateway for HTTPS, as described in Configure PingGateway for TLS (server-side).

b. Add the following route to PingGateway, replacing ig_keystore_directory:

Linux | Windows

```
$HOME/.openig/config/routes/rs-stateless-signed-
ksss.json
```

```
{
  "name": "rs-stateless-signed-ksss",
  "condition" : "${find(request.uri.path, '/rs-stateless-
signed-ksss')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "KeyStoreSecretStore-1",
      "type": "KeyStoreSecretStore",
      "config": {
        "file": "<ig_keystore_directory>/IG_keystore.p12",
        "storeType": "PKCS12",
        "storePasswordSecretId": "keystore.secret.id",
        "entryPasswordSecretId": "keystore.secret.id",
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "mappings": [
          {
            "secretId":
"stateless.access.token.verification.key",
            "aliases": [ "verification-key" ]
          }
        ]
      }
    }
  ],
  "handler" : {
    "type" : "Chain",
    "capture" : "all",
    "config" : {
      "filters" : [ {
        "name" : "OAuth2ResourceServerFilter-1",
        "type" : "OAuth2ResourceServerFilter",
        "config" : {
          "scopes" : [ "myscope" ],
          "requireHttps" : false,
          "accessTokenResolver": {
            "type": "StatelessAccessTokenResolver",
            "config": {
              "secretsProvider": "KeyStoreSecretStore-1",
              "issuer":
"http://am.example.com:8088/openam/oauth2",
              "verificationSecretId":
"stateless.access.token.verification.key"
            }
          }
        }
      } ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": "<html><body><h2>Decoded access_token:
${contexts.oauth2.accessToken.info}</h2></body></html>"
        }
      }
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/rs-stateless-signed-ksss`.

- The keystore password is provided by the SystemAndEnvSecretStore in the heap.
- The OAuth2ResourceServerFilter expects an OAuth 2.0 access token in the header of the incoming authorization request, with the scope `myscope`.
- The `accessTokenResolver` uses a `StatelessAccessTokenResolver` to resolve and verify the authenticity of the access token. The secret is provided by the KeyStoreSecretStore in the heap.
- After the OAuth2ResourceServerFilter validates the access token, it creates the `OAuth2Context` context. You can find more information in OAuth2Context.
- If there is no access token in a request, or if the token validation doesn't complete successfully, the filter returns an HTTP error status to the user agent, and PingGateway stops processing the request, as specified in the RFC, The OAuth 2.0 Authorization Framework: Bearer Token Usage.
- The StaticResponseHandler returns the content of the access token from the context.

3. Test the setup for a signed access token:

   a. Get an access token for the demo user, using the scope `myscope`:

   ```
   $ mytoken=$(curl -s \
   --user "client-application:password" \
   --data
   "grant_type=password&username=demo&password=Ch4ng31t&scope
   =myscope" \
   http://am.example.com:8088/openam/oauth2/access_token | jq
   -r ".access_token")
   ```

   b. Display the token:

   ```
   $ echo ${mytoken}
   ```

   c. Access the route by providing the token returned in the previous step:

   ```
   $ curl -v \
   --cacert /path/to/secrets/ig.example.com-certificate.pem \
   --header "Authorization: Bearer ${mytoken}" \
   https://ig.example.com:8443/rs-stateless-signed-ksss

   ...
   Decoded access_token: {
   sub=(usr!demo),
   ```

```
cts=OAUTH2_STATELESS_GRANT,
...
```

Encrypted tokens with KeyStoreSecretStore

This page shows how to validate encrypted access tokens with the StatelessAccessTokenResolver using a JwkSetSecretStore.

Set up encryption keys

1. Locate the following directories for keys, keystores, and certificates, and in a terminal create variables for them:
   - Directory where the keystore is created: `keystore_directory`
   - AM keystore directory: `am_keystore_directory`
   - PingGateway keystore directory: `ig_keystore_directory`
2. Set up keys for AM:
   a. Generate the encryption key:

   ```
   $ keytool -genseckey \
   -alias encryption-key \
   -dname "CN=ig.example.com, OU=example, O=com, L=fr, ST=fr, C=fr" \
   -keystore "$am_keystore_directory/AM_keystore.p12" \
   -storetype PKCS12 \
   -storepass "password" \
   -keyalg AES \
   -keysize 256
   ```

   b. List the keys in the AM keystore:

   ```
   $ keytool -list \
   -v \
   -keystore "$am_keystore_directory/AM_keystore.p12" \
   -storepass "password" \
   -storetype PKCS12

   ...
   Your keystore contains 1 entry
   Alias name: encryption-key
   ```

   c. Add a file called `keystore.pass`, with the content `password`:

```
$ cd $am_keystore_directory
$ echo -n 'password' > keystore.pass
```

> NOTE
>
> Make sure the password file contains only the password, with no trailing spaces or carriage returns.

The filename corresponds to the secret ID of the store password and entry password for the KeyStoreSecretStore.

d. Restart AM.

3. Set up keys for PingGateway:

a. Import `encryption-key` into the PingGateway keystore, with the alias `decryption-key`:

```
$ keytool -importkeystore \
-srcalias encryption-key \
-srckeystore "$am_keystore_directory/AM_keystore.p12" \
-srcstoretype PKCS12 \
-srcstorepass "password" \
-destkeystore "$ig_keystore_directory/IG_keystore.p12" \
-deststoretype PKCS12 \
-destalias decryption-key \
-deststorepass "password" \
-destkeypass "password"
```

b. List the keys in the PingGateway keystore:

```
$ keytool -list \
-v \
-keystore "$ig_keystore_directory/IG_keystore.p12" \
-storepass "password" \
-storetype PKCS12

...
Your keystore contains 1 entry
Alias name: decryption-key
```

c. In the PingGateway configuration, set an environment variable for the keystore password:

```
$ export KEYSTORE_SECRET_ID='cGFzc3dvcmQ='
```

d. Restart PingGateway.

Validate tokens

1. Set up AM:
   a. Set up AM as described in Validate tokens.
   b. Add a mapping for the encryption keystore:
      i. Select **Secret Stores** > `keystoresecretstore`.
      ii. Select the **Mappings** tab, and add a mapping with the following values:
         - **Secret Label** : `am.services.oauth2.stateless.token.encryption`
         - **Alias** : `encryption-key`
   c. Enable token encryption on the OAuth 2.0 Authorization Provider:
      i. Select **Services** > **OAuth2 Provider**.
      ii. On the **Advanced** tab, select **Encrypt Client-Side Tokens**.
2. Set up PingGateway:
   a. Set up PingGateway for HTTPS, as described in Configure PingGateway for TLS (server-side).
   b. Add the following route to PingGateway, replacing ig_keystore_directory:

Linux Windows

```
$HOME/.openig/config/routes/rs-stateless-encrypted.json
```

```
{
  "name": "rs-stateless-encrypted",
  "condition": "${find(request.uri.path, '/rs-stateless-encrypted')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "KeyStoreSecretStore-1",
      "type": "KeyStoreSecretStore",
      "config": {
        "file": "<ig_keystore_directory>/IG_keystore.p12",
        "storeType": "PKCS12",
        "storePasswordSecretId": "keystore.secret.id",
        "entryPasswordSecretId": "keystore.secret.id",
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "mappings": [
          {
            "secretId":
"stateless.access.token.decryption.key",
            "aliases": [ "decryption-key" ]
          }
        ]
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "capture": "all",
    "config": {
      "filters": [ {
        "name": "OAuth2ResourceServerFilter-1",
        "type": "OAuth2ResourceServerFilter",
        "config": {
          "scopes": [ "myscope" ],
          "requireHttps": false,
          "accessTokenResolver": {
            "type": "StatelessAccessTokenResolver",
            "config": {
              "secretsProvider": "KeyStoreSecretStore-1",
              "issuer":
"http://am.example.com:8088/openam/oauth2",
              "decryptionSecretId":
"stateless.access.token.decryption.key"
            }
          }
        }
      } ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": "<html><body><h2>Decoded access_token:
${contexts.oauth2.accessToken.info}</h2></body></html>"
        }
      }
    }
  }
}
```

Notice the following features of the route compared to `rs-stateless-signed.json` from Validate tokens.

- The route matches requests to `/rs-stateless-encrypted`.
- The OAuth2ResourceServerFilter and KeyStoreSecretStore refer to the configuration for a decryption key instead of a verification key.

3. Test the setup

   a. Get an access token for the demo user, using the scope `myscope`:

```
$ mytoken=$(curl -s \
--user "client-application:password" \
--data
"grant_type=password&username=demo&password=Ch4ng31t&scope
=myscope" \
http://am.example.com:8088/openam/oauth2/access_token | jq
-r ".access_token")
```

   b. Display the token:

```
$ echo ${mytoken}
```

   Note that the token is structured as an encrypted token.

   c. Access the route by providing the token returned in the previous step:

```
$ curl -v \
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--header "Authorization: Bearer ${mytoken}" \
https://ig.example.com:8443/rs-stateless-encrypted

...
Decoded access_token: {
sub=demo,
cts=OAUTH2_STATELESS_GRANT,
...
```

Mutual TLS

Clients can authenticate to AM through mutual TLS (mTLS) and X.509 certificates. Certificates must be self-signed or use public key infrastructure (PKI), as described in version 12 of the draft OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens.

For a client using mTLS, its OAuth 2.0 profile includes a means to access its certificate. When the client requests an access token from AM with mTLS, it authenticates with its certificate to establish the HTTPS connection. This is known as mutual authentication—because the server and client authenticate each other—or client certificate authentication. The client proves it has the private key for the certificate.

The authorization server (AS), such as AM, *binds* the access token to the client certificate with a *confirmation key*. The confirmation key is a certificate *thumbprint*, computed as `base64url-encode(sha256(der(certificate)))`. The access token is then *certificate-bound*, restricted to the client having the corresponding private key. The confirmation key is not necessarily visible in the access token, but the AS provides it on token introspection.

A resource server (RS), such as PingGateway, has the client use HTTPS mutual authentication when requesting a resource with the certificate-bound access token. The RS gets the certificate and proves the client has its private key when setting up HTTPS. The RS gets the confirmation key for the access token and verifies the certificate matches the key. The RS only grants access when the certificate matches the confirmation key.

This proof-of-possession interaction ensures only the client with the private key for the certificate can use the token to access protected resources.

Access to the client certificate

Service	Terminates HTTPS?	Do this
AM in an application container	Yes	Configure the container to allow mutual authentication for HTTPS. Get the client certificate from the HTTPS connection.
	No	Configure the service terminating HTTPS to allow mutual authentication and to forward the request with the client certificate in a trusted header. Get the client certificate from the trusted header.

Service	Terminates HTTPS?	Do this
PingGateway	Yes	Configure PingGateway to allow mutual authentication for HTTPS. Get the client certificate from the HTTPS connection.
	No	Configure the service terminating HTTPS to allow mutual authentication and to forward the request with the client certificate in a trusted header. Get the client certificate from the trusted header.

Next steps

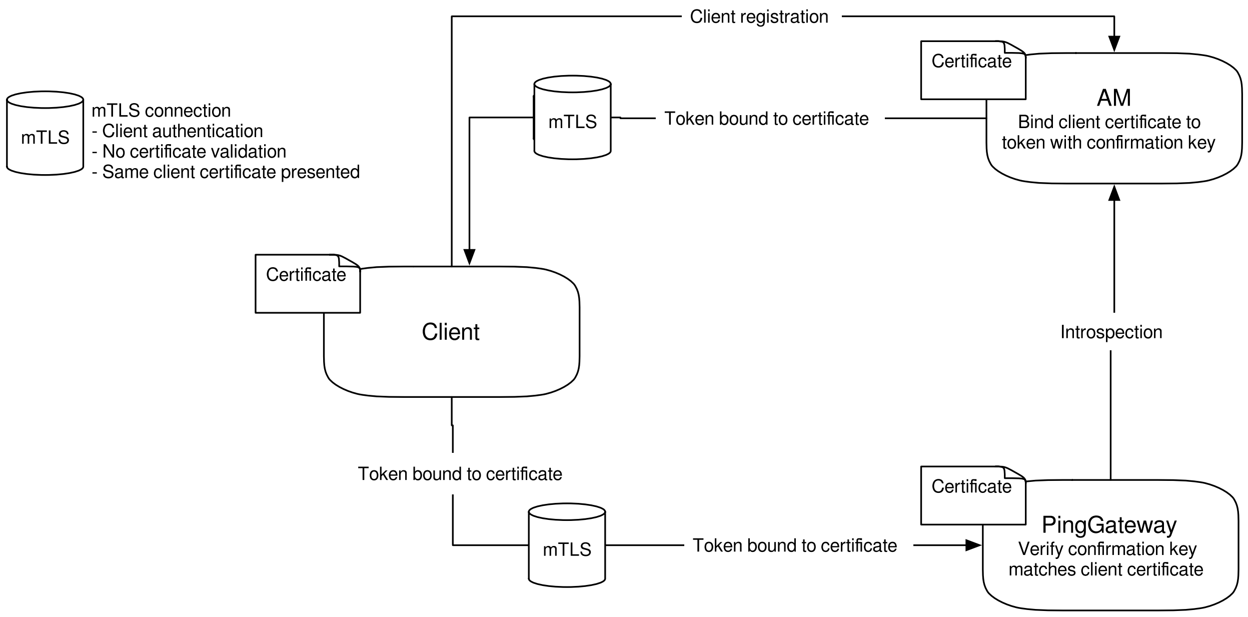
When implementing mTLS, make sure AM and PingGateway either support HTTPS mutual authentication or get the client certificate in a trusted header from the service that supports HTTPS mutual authentication.

- mTLS with client certificates shows how to configure PingGateway to get the client certificate from mutual authentication.
- mTLS with trusted headers shows how to configure PingGateway to get the client certificate from a trusted header.

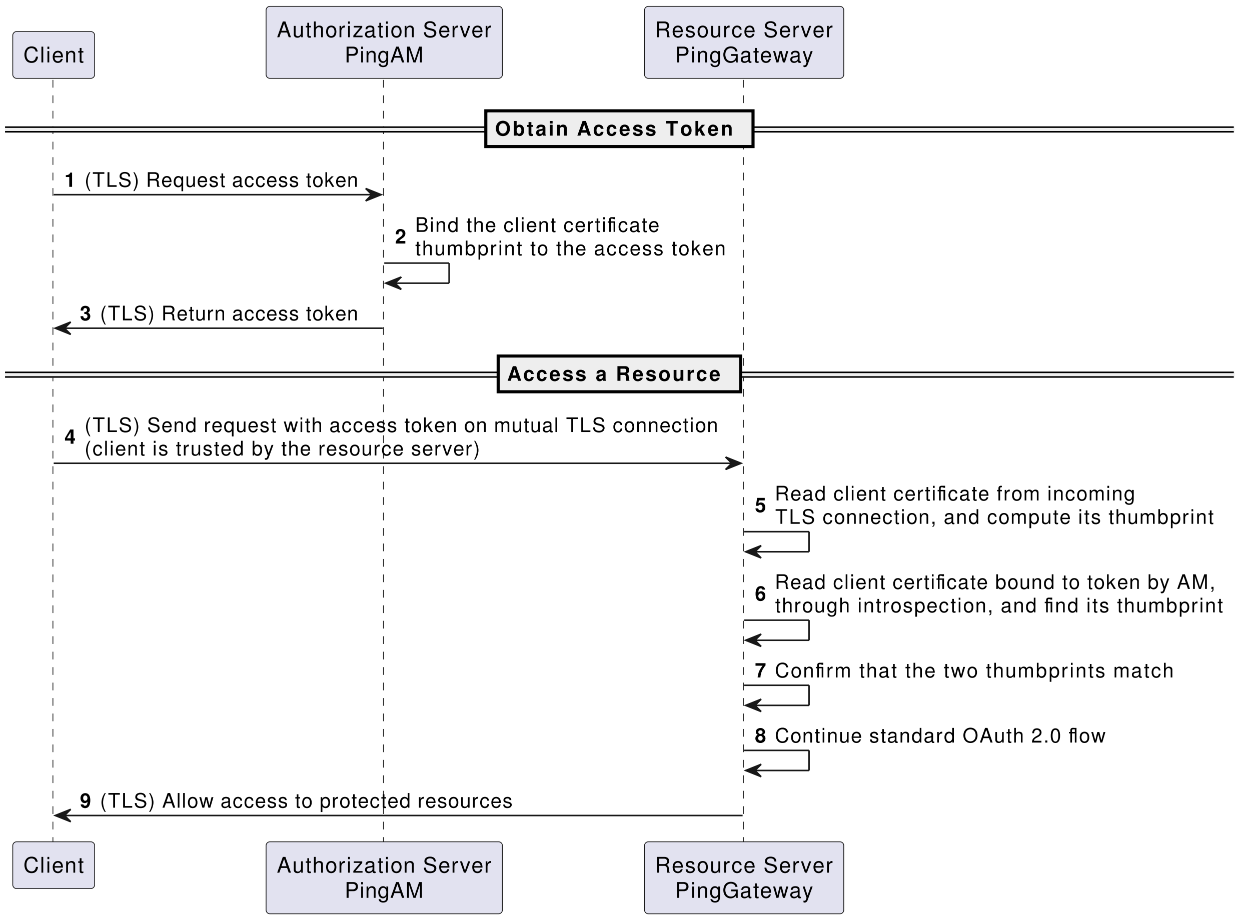
mTLS with client certificates

PingGateway can validate the thumbprint of certificate-bound access tokens by reading the client certificate from the TLS connection.

For this example, the client must be connected directly to PingGateway through a TLS connection, for which PingGateway is the TLS termination point, as shown in the following image. If TLS is terminated at a reverse proxy or load balancer before PingGateway, use the example in mTLS with trusted headers.



Client registration
Certificate
AM
Bind client certificate to token with confirmation key
mTLS
mTLS connection
- Client authentication
- No certificate validation
- Same client certificate presented
mTLS
Token bound to certificate
Certificate
Client
Introspection
Token bound to certificate
Certificate
PingGateway
Verify confirmation key matches client certificate
mTLS
Token bound to certificate



Client
Authorization Server PingAM
Resource Server PingGateway
Obtain Access Token
1 (TLS) Request access token
2 Bind the client certificate thumbprint to the access token
3 (TLS) Return access token
Access a Resource
4 (TLS) Send request with access token on mutual TLS connection (client is trusted by the resource server)
5 Read client certificate from incoming TLS connection, and compute its thumbprint
6 Read client certificate bound to token by AM, through introspection, and find its thumbprint
7 Confirm that the two thumbprints match
8 Continue standard OAuth 2.0 flow
9 (TLS) Allow access to protected resources
Client
Authorization Server PingAM
Resource Server PingGateway

Follow the steps in this example to try mTLS using standard TLS client certificate authentication.

Prepare the keys

1. To make it easy to identify and refer to secrets used in mTLS examples, create directories and environment variables:

```
$ export ig_keystore_directory=/path/to/ig/secrets
$ export am_keystore_directory=/path/to/am/secrets
$ export
oauth2_client_keystore_directory=/path/to/client/secrets
```

2. Create keys and certificates for the example:

   a. Create self-signed RSA key pairs for AM and the client:

   ```
   $ keytool -genkeypair \
   -alias openam-server \
   -keyalg RSA \
   -keysize 2048 \
   -keystore $am_keystore_directory/keystore.p12 \
   -storepass changeit \
   -storetype PKCS12 \
   -keypass changeit \
   -validity 360 \
   -dname CN=am.example.com,O=Example,C=FR
   ```

   ```
   $ keytool -genkeypair \
   -alias oauth2-client \
   -keyalg RSA \
   -keysize 2048 \
   -keystore $oauth2_client_keystore_directory/keystore.p12 \
   -storepass changeit \
   -storetype PKCS12 \
   -keypass changeit \
   -validity 360 \
   -dname CN=test
   ```

   b. Export the certificates to .pem so that the **curl** client can verify the identity of the AM and PingGateway servers:

   ```
   $ keytool -export \
   -rfc \
   -alias openam-server \
   -keystore $am_keystore_directory/keystore.p12 \
   -storepass changeit \
   -storetype PKCS12 \
   -file $am_keystore_directory/openam-server.cert.pem

   Certificate stored in file .../openam-server.cert.pem
   ```

c. Extract the certificate and client private key to .pem so that the `curl` command can identity itself as the client for the HTTPS connection:

```
$ keytool -export \
-rfc \
-alias oauth2-client \
-keystore $oauth2_client_keystore_directory/keystore.p12 \
-storepass changeit \
-storetype PKCS12 \
-file $oauth2_client_keystore_directory/client.cert.pem

Certificate stored in file .../client.cert.pem
```

```
$ openssl pkcs12 \
-in $oauth2_client_keystore_directory/keystore.p12 \
-nocerts \
-nodes \
-passin pass:changeit \
-out $oauth2_client_keystore_directory/client.key.pem

...verified OK
```

d. Create the CACerts truststore so that AM can validate the client identity:

```
$ keytool -import \
-noprompt \
-trustcacerts \
-file $oauth2_client_keystore_directory/client.cert.pem \
-keystore $oauth2_client_keystore_directory/cacerts.p12 \
-storepass changeit \
-storetype PKCS12 \
-alias client-cert

Certificate was added to keystore
```

e. In the *ig_keystore_directory*, add a file called `keystore.pass` containing the keystore password:

```
$ cd $ig_keystore_directory
$ echo -n 'changeit' > keystore.pass
```

Prepare AM

1. Configure AM for HTTPS connections using information in the AM documentation about Secure HTTP and LDAP connections.

   ▼ *Learn more*

   1. Add a connector configuration for port `8445` to AM's Tomcat `server.xml`, replacing the values for the keystore directories with your path. If the file already contains a connector for the port, edit that connector or replace it:

      ```
      <Connector port="8445" protocol="HTTP/1.1"
      SSLEnabled="true" scheme="https" secure="true">
        <SSLHostConfig protocols="+TLSv1.2,-TLSv1.1,-TLSv1,-
      SSLv2Hello,-SSLv3"
                       certificateVerification="optionalNoCA"

      truststoreFile="oauth2_client_keystore_directory/cacerts.
      p12"
                       truststorePassword="changeit"
                       truststoreType="PKCS12">
          <Certificate
      certificateKeystoreFile="am_keystore_directory/keystore.p
      12"
                       certificateKeystorePassword="changeit"
                       certificateKeystoreType="PKCS12"/>
        </SSLHostConfig>
      </Connector>
      ```

   2. In AM, export an environment variable for the base64-encoded value of the password (`changeit`) for the `cacerts.p12` truststore:

      ```
      $ export PASSWORDSECRETID='Y2hhbmdlaXQ='
      ```

   3. Restart AM, and make sure you can access it on the secure port `https://am.example.com:8445/openam`.

2. Configure AM for mutual TLS using information in the AM documentation about Mutual TLS.

   ▼ *Learn more*

   1. In the AM admin UI, select **Applications** > **Agents** > **Identity Gateway**, and register a PingGateway agent that can introspect access tokens:

      - **Agent ID**: `ig_agent`
      - **Password**: `password`
      - **Token Introspection**: `Realm Only`

IMPORTANT

IMPORTANT

Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

2. Configure an OAuth 2.0 Authorization Server with settings for mTLS:
   a. Select **Services** > **Add a Service** > **OAuth2 Provider**, and add a service with the default values.
   b. On the **Advanced** tab, select the following value:
      - **Support TLS Certificate-Bound Access Tokens**: enabled
3. Configure an OAuth 2.0 client to request access tokens:
   a. Select **Applications** > **OAuth 2.0** > **Clients**, and add a client with the following values:
      - **Client ID**: `client-application`
      - **Client secret**: `password`
      - **Scope(s)**: `test`
   b. (Optional) On the **Core** tab, switch to using a client secret associated with a secret label by setting a **Secret Label Identifier** and mapping the label to a secret.

      To learn more, read Create a client profile and Map and rotate secrets in the AM documentation.
   c. On the **Advanced** tab, select the following values:
      - **Grant Types**: `Client Credentials`

        The `password` is the only grant type used by the client in the example.
      - **Token Endpoint Authentication Method**: `self_signed_tls_client_auth`
   d. On the **Signing and Encryption** tab, set the following values:
      - **mTLS Self-Signed Certificate**: Enter the content of the X.509 certificate, `client.cert.pem`.
      - **mTLS Subject DN**: `CN=test`

        When this option is set, AM requires the subject DN in the client certificate to have the same value. This ensures that the certificate is from the client and not just a valid certificate trusted by the trust manager.
      - **Public key selector**: `x509`
      - **Use Certificate-Bound Access Tokens**: Enabled

Prepare PingGateway

1. Configure PingGateway for HTTPS using information from Configure PingGateway for TLS (server-side).

   This example uses a self-signed certificate stored in a PEM file.

2. Configure PingGateway for mutual TLS using information from Configure PingGateway for mTLS (server-side).

   This example uses the following `admin.json` with a SecretsTrustManager:

   Linux | Windows

   ```
   $HOME/.openig/config/admin.json
   ```

   ```
   {
     "mode": "DEVELOPMENT",
     "properties": {
       "ig_keystore_directory": "/path/to/ig/secrets",
       "oauth2_client_keystore_directory":
   "/path/to/client/secrets"
     },
     "connectors": [
       {
         "port": 8080
       },
       {
         "port": 8443,
         "tls": {
           "type": "ServerTlsOptions",
           "config": {
             "alpn": {
               "enabled": true
             },
             "clientAuth": "REQUEST",
             "keyManager": "SecretsKeyManager-1",
             "trustManager": "SecretsTrustManager-1"
           }
         }
       }
     ],
     "heap": [
       {
   ```

```
        "name": "SecretsPasswords",
        "type": "FileSystemSecretStore",
        "config": {
          "directory": "&{ig_keystore_directory}",
          "format": "PLAIN"
        }
      },
      {
        "name": "SecretsKeyManager-1",
        "type": "SecretsKeyManager",
        "config": {
          "signingSecretId": "key.manager.secret.id",
          "secretsProvider": "ServerIdentityStore"
        }
      },
      {
        "name": "SecretsTrustManager-1",
        "type": "SecretsTrustManager",
        "config": {
          "verificationSecretId": "trust.manager.secret.id",
          "secretsProvider": {
            "type": "KeyStoreSecretStore",
            "config": {
              "file": "&
{oauth2_client_keystore_directory}/cacerts.p12",
              "storePasswordSecretId": "keystore.pass",
              "secretsProvider": "SecretsPasswords",
              "mappings": [
                {
                  "secretId": "trust.manager.secret.id",
                  "aliases": ["client-cert"]
                }
              ]
            }
          }
        }
      },
      {
        "name": "ServerIdentityStore",
        "type": "FileSystemSecretStore",
        "config": {
          "format": "PLAIN",
          "directory": "&{ig_keystore_directory}",
          "suffix": ".pem",
          "mappings": [{
```

```
          "secretId": "key.manager.secret.id",
          "format": {
            "type": "PemPropertyFormat"
          }
        }]
      }
    }
  ]
}
```

Notice the `ServerTlsOptions`:

- The trust manager settings let PingGateway trust the self-signed client certificate.
- The `"clientAuth": "REQUEST"` setting permits optional HTTPS mutual authentication.

  Clients using mTLS authenticate with their certificates when setting up HTTPS. Other clients can connect over HTTPS without presenting a client certificate.

3. Replace the values of the secret directories with your directories, and then start PingGateway.

Make PingGateway an RS

1. Configure PingGateway as a resource server for mTLS:

   a. Set an environment variable for the PingGateway agent password, and then restart PingGateway:

   ```
   $ export AGENT_SECRET_ID='cGFzc3dvcmQ='
   ```

   The password is retrieved by a `SystemAndEnvSecretStore`, and must be base64-encoded.

   b. Add the following route to PingGateway:

   Linux | Windows

   ```
   $HOME/.openig/config/routes/mtls-certificate.json
   ```

   ```
   {
     "name": "mtls-certificate",
     "condition": "${find(request.uri.path, '/mtls-
   certificate')}",
   ```

```
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "capture": "all",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [
              "test"
            ],
            "requireHttps": false,
            "accessTokenResolver": {
              "type":
"ConfirmationKeyVerifierAccessTokenResolver",
              "config": {
                "delegate": {
                  "name": "token-resolver-1",
                  "type":
"TokenIntrospectionAccessTokenResolver",
                  "config": {
                    "amService": "AmService-1",
                    "providerHandler": {
                      "type": "Chain",
                      "config": {
                        "filters": [
                          {
                            "type":
"HttpBasicAuthenticationClientFilter",
                            "config": {
                              "username": "ig_agent",
                              "passwordSecretId":
"agent.secret.id",
                              "secretsProvider":
"SystemAndEnvSecretStore-1"
                            }
                          }
                        ],
                        "handler":
"ForgeRockClientHandler"
                      }
                    }
                  }
                }
              }
            }
          }
        }
      ],
      "handler": {
        "name": "StaticResponseHandler-1",
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/plain; charset=UTF-8"
]
          },
          "entity": "mTLS\n Valid token:
${contexts.oauth2.accessToken.token}\n Confirmation keys:
${contexts.oauth2}"
        }
      }
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/mtls-certificate`.

- The `OAuth2ResourceServerFilter` uses the `ConfirmationKeyVerifierAccessTokenResolver` to validate the certificate thumbprint against the thumbprint from the resolved access token provided by AM.

  The `ConfirmationKeyVerifierAccessTokenResolver` then delegates token resolution to the `TokenIntrospectionAccessTokenResolver`.
- The `providerHandler` adds an authorization header to the request, containing the username and password of the OAuth 2.0 client with the scope to examine (introspect) access tokens.
- The `OAuth2ResourceServerFilter` checks the resolved token has the required scopes and injects the token info into the context.
- The `StaticResponseHandler` returns the content of the access token from the context.

Try certificate-based mTLS

1. Get a certificate-bound access token from AM as the client application:

```
$ export ACCESS_TOKEN=$(curl \
--request POST \
--cacert $am_keystore_directory/openam-server.cert.pem \
--cert $oauth2_client_keystore_directory/client.cert.pem \
--key $oauth2_client_keystore_directory/client.key.pem \
--header 'cache-control: no-cache' \
--header 'content-type: application/x-www-form-urlencoded' \
--data 'client_id=client-application' \
--data 'grant_type=client_credentials' \
--data 'scope=test' \
https://am.example.com:8445/openam/oauth2/access_token | jq -r
.access_token)
```

   Notice the client gets an access token without using a client secret. It authenticates with its self-signed certificate.

2. Introspect the access token on AM using the PingGateway agent credentials:

```
$ curl \
--request POST \
--user ig_agent:password \
--header 'content-type: application/x-www-form-urlencoded' \
--data "token=$ACCESS_TOKEN" \
http://am.example.com:8088/openam/oauth2/realms/root/introspec
t | jq
```

```
{
  "active": true,
  "scope": "test",
  "realm": "/",
  "client_id": "client-application",
  "user_id": "client-application",
  "username": "client-application",
  "token_type": "Bearer",
  "exp": 1724250516,
  "sub": "(age!client-application)",
  "iss": "http://am.example.com:8088/openam/oauth2",
  "subname": "client-application",
  "cnf": {
    "x5t#S256": "TTXH27YoFFCgOAQ0189KMBKeqxU1ZfZ_2nYGxrsjHlM"
  },
  "authGrantId": "JqckJ_KhDLDeb4cKRkeBJcXZZUE",
  "auditTrackingId": "962fd5f6-fc2f-43c1-b044-ed1eb33d7aef-
429"
}
```

The `cnf` property indicates the value of the confirmation code:

- `x5` : X509 certificate
- `t` : thumbprint
- `#` : separator
- `S256` : algorithm used to hash the raw certificate bytes

3. Access the PingGateway route to validate the confirmation key with the ConfirmationKeyVerifierAccessTokenResolver:

```
$ curl \
--request POST \
--cacert $ig_keystore_directory/ig.example.com-certificate.pem
\
--cert $oauth2_client_keystore_directory/client.cert.pem \
--key $oauth2_client_keystore_directory/client.key.pem \
--header "Authorization: Bearer ${ACCESS_TOKEN}" \
https://ig.example.com:8443/mtls-certificate
mTLS
 Valid token: <ACCESS_TOKEN>
 Confirmation keys: { ... }
```

The command displays the validated token and confirmation keys.

mTLS with trusted headers

PingGateway can validate the thumbprint of certificate-bound access tokens by reading the client certificate from a configured, trusted HTTP header.

Use this method when TLS is terminated at a reverse proxy or load balancer before PingGateway. PingGateway cannot authenticate the client through the TLS connection's client certificate because:

- If the connection is over TLS, the connection presents the certificate of the TLS termination point before PingGateway.
- If the connection is not over TLS, the connection presents no client certificate.

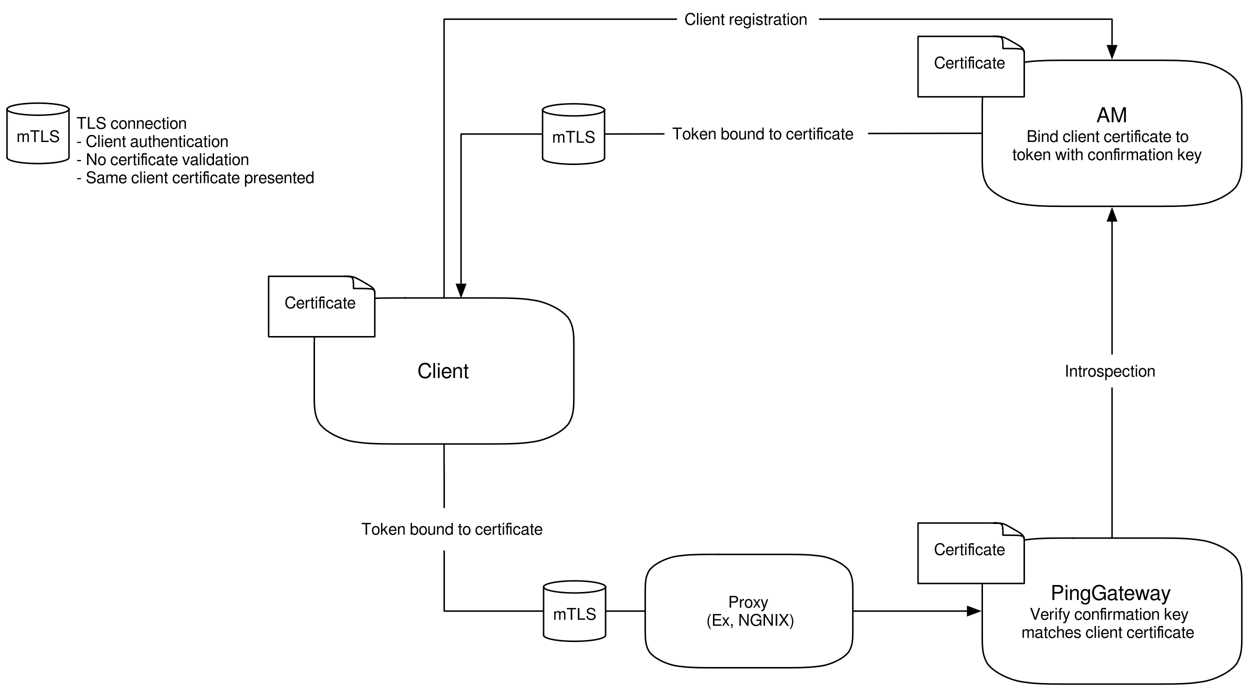
If the client is connected directly to PingGateway through a TLS connection, for which PingGateway is the TLS termination point, use the example in mTLS with client certificates.

Configure the proxy or load balancer to:

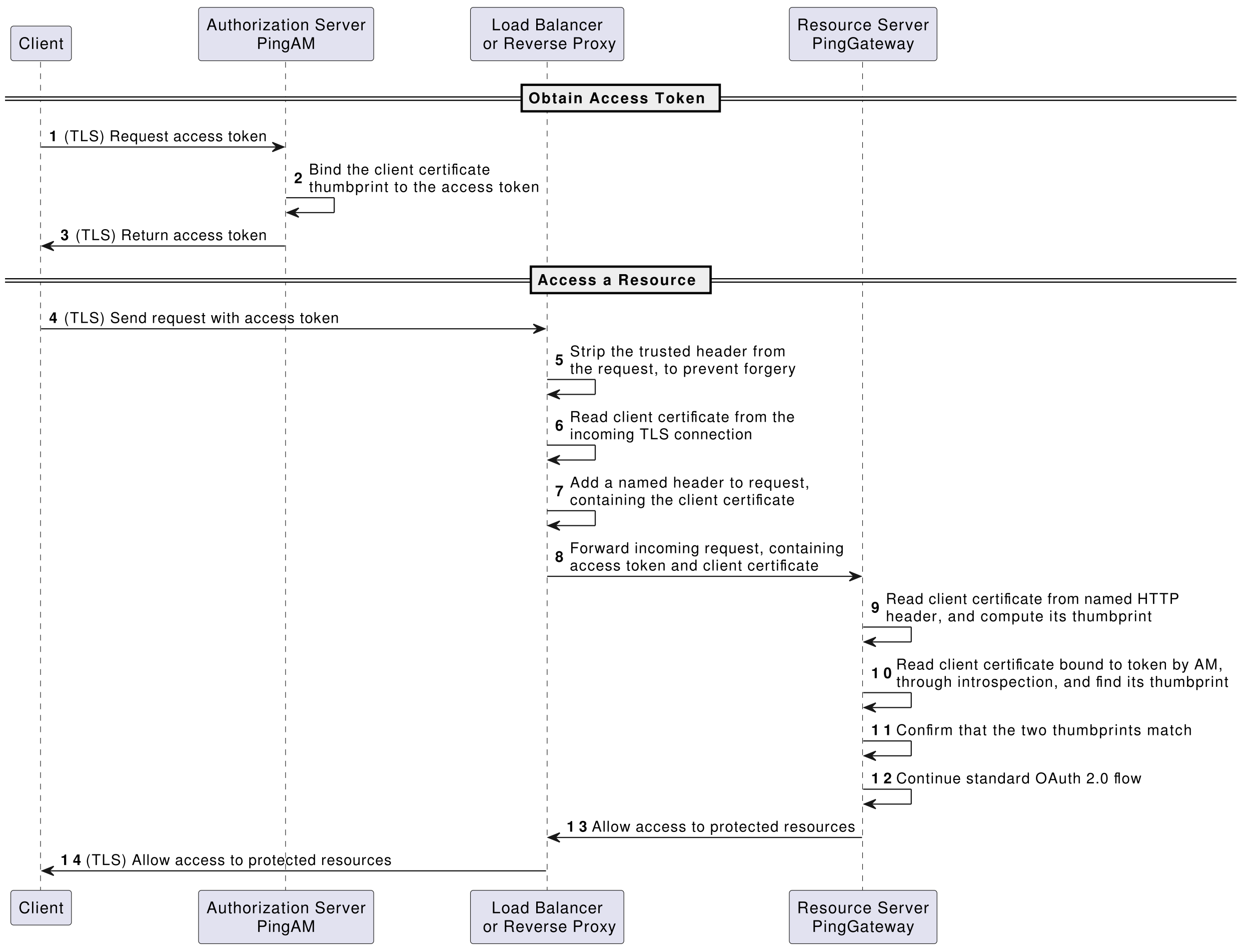
- Forward the encoded certificate to PingGateway in the trusted header. Encode the certificate in an HTTP-header compatible format that can convey a full certificate, so that PingGateway can rebuild the certificate.
- Strip the trusted header from incoming requests, and change the default header name to something an attacker can't guess.

Because there is a trust relationship between PingGateway and the TLS termination point, PingGateway doesn't authenticate the contents of the trusted header. PingGateway accepts any value in a header from a trusted TLS termination point.

The following image illustrates the connections and certificates required by the example:



Client registration
Certificate
AM
Bind client certificate to token with confirmation key
mTLS
TLS connection
- Client authentication
- No certificate validation
- Same client certificate presented
mTLS
Token bound to certificate
Certificate
Client
Introspection
Token bound to certificate
Certificate
PingGateway
Verify confirmation key matches client certificate
mTLS
Proxy
(Ex, NGNIX)



Client
Authorization Server PingAM
Load Balancer or Reverse Proxy
Resource Server PingGateway
Obtain Access Token
1 (TLS) Request access token
2 Bind the client certificate thumbprint to the access token
3 (TLS) Return access token
Access a Resource
4 (TLS) Send request with access token
5 Strip the trusted header from the request, to prevent forgery
6 Read client certificate from the incoming TLS connection
7 Add a named header to request, containing the client certificate
8 Forward incoming request, containing access token and client certificate
9 Read client certificate from named HTTP header, and compute its thumbprint
10 Read client certificate bound to token by AM, through introspection, and find its thumbprint
11 Confirm that the two thumbprints match
12 Continue standard OAuth 2.0 flow
13 Allow access to protected resources
14 (TLS) Allow access to protected resources
Client
Authorization Server PingAM
Load Balancer or Reverse Proxy
Resource Server PingGateway

Follow the steps in this example to try mTLS using trusted headers.

Before you start

1. Set up the keystores, truststores, AM, and PingGateway as described in mTLS with client certificates.

2. URL-encode the value of `$oauth2_client_keystore_directory/client.cert.pem`.

   PingGateway needs the certificate to validate the confirmation key.

Make PingGateway an RS

1. Add the following route to PingGateway:

   Linux | Windows

   ```
   $HOME/.openig/config/routes/mtls-header.json
   ```

   ```
   {
     "name": "mtls-header",
     "condition": "${find(request.uri.path, '/mtls-header')}",
     "heap": [
       {
         "name": "SystemAndEnvSecretStore-1",
         "type": "SystemAndEnvSecretStore"
       },
       {
         "name": "AmService-1",
         "type": "AmService",
         "config": {
           "agent": {
             "username": "ig_agent",
             "passwordSecretId": "agent.secret.id"
           },
           "secretsProvider": "SystemAndEnvSecretStore-1",
           "url": "http://am.example.com:8088/openam"
         }
       }
     ],
     "handler": {
       "type": "Chain",
       "capture": "all",
       "config": {
         "filters": [
           {
             "name": "CertificateThumbprintFilter-1",
             "type": "CertificateThumbprintFilter",
             "config": {
   ```

```
            "certificate":
"${pemCertificate(urlDecode(request.headers['x-ssl-cert']
[0]))}",
            "failureHandler": {
              "type": "ScriptableHandler",
              "config": {
                "type": "application/x-groovy",
                "source": [
                  "def response = new
Response(Status.TEAPOT);",
                  "response.entity = 'Failure in
CertificateThumbprintFilter'",
                  "return response"
                ]
              }
            }
          }
        },
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [
              "test"
            ],
            "requireHttps": false,
            "accessTokenResolver": {
              "type":
"ConfirmationKeyVerifierAccessTokenResolver",
              "config": {
                "delegate": {
                  "name": "token-resolver-1",
                  "type":
"TokenIntrospectionAccessTokenResolver",
                  "config": {
                    "amService": "AmService-1",
                    "providerHandler": {
                      "type": "Chain",
                      "config": {
                        "filters": [
                          {
                            "type":
"HttpBasicAuthenticationClientFilter",
                            "config": {
                              "username": "ig_agent",
                              "passwordSecretId":
"agent.secret.id",
                              "secretsProvider":
"SystemAndEnvSecretStore-1"
                            }
                          }
                        ],
                        "handler": "ForgeRockClientHandler"
                      }
                    }
                  }
                }
              }
            }
          }
        }
      ],
      "handler": {
        "name": "StaticResponseHandler-1",
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/plain; charset=UTF-8" ]
          },
          "entity": "mTLS\n Valid token:
${contexts.oauth2.accessToken.token}\n Confirmation keys:
${contexts.oauth2}"
        }
      }
    }
  }
}
```

Notice the following features of the route compared to `mtls-certificate.json`:

- The route matches requests to `/mtls-header`.
- The `CertificateThumbprintFilter` extracts the client certificate from the trusted header.

  In this example, the filter is configured as if NGINX were sending the trusted header. For additional examples, refer to the CertificateThumbprintFilter examples.

  The filter computes the certificate thumbprint and makes the thumbprint available to the `ConfirmationKeyVerifierAccessTokenResolver`.

Try mTLS with trusted headers

1. Get a certificate-bound access token from AM as the client application:

```
$ export ACCESS_TOKEN=$(curl \
--request POST \
--cacert $am_keystore_directory/openam-server.cert.pem \
--cert $oauth2_client_keystore_directory/client.cert.pem \
--key $oauth2_client_keystore_directory/client.key.pem \
--header 'cache-control: no-cache' \
--header 'content-type: application/x-www-form-urlencoded' \
--data 'client_id=client-application' \
--data 'grant_type=client_credentials' \
--data 'scope=test' \
https://am.example.com:8445/openam/oauth2/access_token | jq -r
.access_token)
```

   Notice the client gets an access token without using a client secret. It authenticates with its self-signed certificate.

2. Introspect the access token on AM using the PingGateway agent credentials:

```
$ curl \
--request POST \
--user ig_agent:password \
--header 'content-type: application/x-www-form-urlencoded' \
--data "token=$ACCESS_TOKEN" \
http://am.example.com:8088/openam/oauth2/realms/root/introspec
t | jq
{
  "active": true,
  "scope": "test",
  "realm": "/",
  "client_id": "client-application",
  "user_id": "client-application",
  "username": "client-application",
  "token_type": "Bearer",
  "exp": 1724249775,
  "sub": "(age!client-application)",
  "iss": "http://am.example.com:8088/openam/oauth2",
  "subname": "client-application",
  "cnf": {
    "x5t#S256": "TTXH27YoFFCgOAQ0189KMBKeqxU1ZfZ_2nYGxrsjHlM"
  },
  "authGrantId": "LMhPEqYaxMbrd2zXMAQjHcc8JYE",
  "auditTrackingId": "962fd5f6-fc2f-43c1-b044-ed1eb33d7aef-
403"
}
```

The `cnf` property indicates the value of the confirmation code:

- `x5`: X509 certificate
- `t`: thumbprint
- `#`: separator
- `S256`: algorithm used to hash the raw certificate bytes

3. Access the PingGateway route to validate the confirmation key.

   The <url-encoded-cert> is the URL-encoded value of `$oauth2_client_keystore_directory/client.cert.pem`:

```
$ curl \
--request POST \
--cacert $ig_keystore_directory/ig.example.com-certificate.pem
\
--header "Authorization: Bearer $ACCESS_TOKEN" \
--header 'x-ssl-cert: <url-encoded-cert>'
https://ig.example.com:8443/mtls-header
mTLS
 Valid token: UnUxGRuwXx_ugUCvNKFM3GJo3Cc
 Confirmation keys: { ... }
```

   The command displays the validated token and confirmation keys.

OAuth 2.0 context for authentication

This section contains an example route that retrieves scopes from a token introspection, assigns them as the PingGateway session username and password, and uses them to log the user directly in to the sample application.

For information about the context, refer to OAuth2Context.

Before you start, set up and test the example in Validate access tokens with introspection.

1. Set up AM:

   a. Select **Identities**, and change the email address of the demo user to `demo`.

   b. Select **Scripts** > **OAuth2 Access Token Modification Script**, and replace the default script as follows:

```
import org.forgerock.http.protocol.Request
import org.forgerock.http.protocol.Response
import com.iplanet.sso.SSOException
import groovy.json.JsonSlurper

def attributes = identity.getAttributes(["mail"].toSet())
accessToken.setField("mail", attributes["mail"][0])
accessToken.setField("password", "Ch4ng31t")
```

The AM script adds user profile information to the access token, and adds a `password` field with the value `Ch4ng31t`.

WARNING

Don't use this example in production. If the token is stateless and unencrypted, the password value is easily accessible when you have the token.

2. Set up PingGateway:

   a. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/rs-pwreplay.json
```

```
{
  "name" : "rs-pwreplay",
  "baseURI" : "http://app.example.com:8081",
  "condition" : "${find(request.uri.path, '^/rs-
pwreplay')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler" : {
    "type" : "Chain",
    "config" : {
      "filters" : [
        {
          "name" : "OAuth2ResourceServerFilter-1",
          "type" : "OAuth2ResourceServerFilter",
          "config" : {
            "scopes" : [ "mail", "employeenumber" ],
            "requireHttps" : false,
            "realm" : "OpenIG",
            "accessTokenResolver": {
              "name":
"TokenIntrospectionAccessTokenResolver-1",
              "type":
"TokenIntrospectionAccessTokenResolver",
              "config": {
                "amService": "AmService-1",
                "providerHandler": {
                  "type": "Chain",
                  "config": {
                    "filters": [
                      {
                        "type":
"HttpBasicAuthenticationClientFilter",
                        "config": {
                          "username": "ig_agent",
                          "passwordSecretId":
"agent.secret.id",
                          "secretsProvider":
"SystemAndEnvSecretStore-1"
                        }
                      }
                    ],
                    "handler": "ForgeRockClientHandler"
                  }
                }
              }
            }
          }
        },
        {
          "type": "AssignmentFilter",
          "config": {
            "onRequest": [{
              "target": "${session.username}",
              "value":
"${contexts.oauth2.accessToken.info.mail}"
            },
              {
                "target": "${session.password}",
                "value":
"${contexts.oauth2.accessToken.info.password}"
              }
            ]
          }
        },
        {
          "type": "StaticRequestFilter",
          "config": {
            "method": "POST",
            "uri": "http://app.example.com:8081/login",
            "form": {
              "username": [
                "${session.username}"
              ],
              "password": [
                "${session.password}"
              ]
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route compared to `rs-introspect.json`:

- The route matches requests to `/rs-pwreplay`.
- The AssignmentFilter accesses the context, and injects the username and password into the SessionContext, `${Session}`.
- The StaticRequestFilter retrieves the username and password from `session`, and replaces the original HTTP GET request with an HTTP POST

login request that contains the credentials to authenticate.

3. Test the setup:

   a. In a terminal window, use a **`curl`** command similar to the following to retrieve an access token:

   ```
   $ mytoken=$(curl -s \
   --user "client-application:password" \
   --data
   "grant_type=password&username=demo&password=Ch4ng31t&scope
   =mail%20employeenumber" \
   http://am.example.com:8088/openam/oauth2/access_token | jq
   -r ".access_token")
   ```

   b. Validate the access token returned in the previous step:

   ```
   $ curl -v \
   --cacert /path/to/secrets/ig.example.com-certificate.pem \
   --header "Authorization: Bearer ${mytoken}" \
   https://ig.example.com:8443/rs-pwreplay
   ```

   HTML for the sample application is displayed.

Cache access tokens

This section builds on the example in Validate access tokens with introspection to cache and then revoke access tokens.

When the access token **is not** cached, PingGateway calls AM to validate the access token. When the access token **is** cached, PingGateway doesn't validate the access token with AM.

When an access token is revoked on AM, the CacheAccessTokenResolver can delete the token from the cache when both of the following conditions are true:

- The `notification` property of AmService is enabled.
- The delegate AccessTokenResolver provides the token metadata required to update the cache.

When a refresh_token is revoked on AM, all associated access tokens are automatically and immediately revoked.

Before you start, set up and test the example in Validate access tokens with introspection.

1. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/rs-introspect-cache.json
```

```
{
  "name": "rs-introspect-cache",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/rs-introspect-
cache$')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "http://am.example.com:8088/openam",
        "realm": "/",
        "agent" : {
          "username" : "ig_agent",
          "passwordSecretId" : "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [
              "mail",
              "employeenumber"
            ],
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name": "CacheAccessTokenResolver-1",
              "type": "CacheAccessTokenResolver",
              "config": {
                "enabled": true,
                "defaultTimeout ": "1 hour",
                "maximumTimeToCache": "1 day",
                "amService":"AmService-1",
                "delegate": {
                  "name":
"TokenIntrospectionAccessTokenResolver-1",
                  "type":
"TokenIntrospectionAccessTokenResolver",
                  "config": {
                    "amService": "AmService-1",
                    "providerHandler": {
                      "type": "Chain",
                      "config": {
                        "filters": [
                          {
                            "type":
"HttpBasicAuthenticationClientFilter",
                            "config": {
                              "username": "ig_agent",
                              "passwordSecretId":
"agent.secret.id",
                              "secretsProvider":
"SystemAndEnvSecretStore-1"
                            }
                          }
                        ],
                        "handler": {
                          "type": "Delegate",
                          "capture": "all",
                          "config": {
                            "delegate":
"ForgeRockClientHandler"
                          }
                        }
                      }
                    }
                  }
                }
              }
            }
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": "<html><body><h2>Decoded access_token:
${contexts.oauth2.accessToken.info}</h2></body></html>"
        }
      }
    }
  }
}
```

Notice the following features of the route compared to `rs-introspect.json`, in Validate access tokens with introspection:

- The OAuth2ResourceServerFilter uses a CacheAccessTokenResolver to cache the access token, and then delegate token resolution to the TokenIntrospectionAccessTokenResolver.
- The `amService` property in CacheAccessTokenResolver enables WebSocket notifications from AM, for events such as token revocation.
- The TokenIntrospectionAccessTokenResolver uses a ForgeRockClientHandler and a capture decorator to capture PingGateway's interactions with AM.

2. Test token caching:

   a. In a terminal window, use a **`curl`** command similar to the following to retrieve an access token:

   ```
   $ mytoken=$(curl -s \
   --user "client-application:password" \
   --data
   "grant_type=password&username=demo&password=Ch4ng31t&scope
   =mail%20employeenumber" \
   http://am.example.com:8088/openam/oauth2/access_token | jq
   -r ".access_token")
   ```

   b. Access the route, using the access token returned in the previous step:

```
$ curl -v \
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--header "Authorization: Bearer ${mytoken}" \
https://ig.example.com:8443/rs-introspect-cache

{
 active = true,
 scope = employeenumber mail,
 client_id = client-application,
 user_id = demo,
 token_type = Bearer,
 exp = 158...907,
 ...
}
```

c. In the route log, note that PingGateway calls AM to introspect the access token:

```
POST
http://am.example.com:8088/openam/oauth2/realms/root/intro
spect HTTP/1.1
```

d. Access the route again. In the route log note that this time PingGateway doesn't call AM, because the token is cached.

e. Disable the cache and repeat the previous steps to cause PingGateway to call AM to validate the access token for each request.

3. Test token revocation:

a. In a terminal window, use a `curl` command similar to the following to revoke the access token obtained in the previous step:

```
$ curl --request POST \
--data "token=${mytoken}" \
--data "client_id=client-application" \
--data "client_secret=password" \
"http://am.example.com:8088/openam/oauth2/realms/root/toke
n/revoke"
```

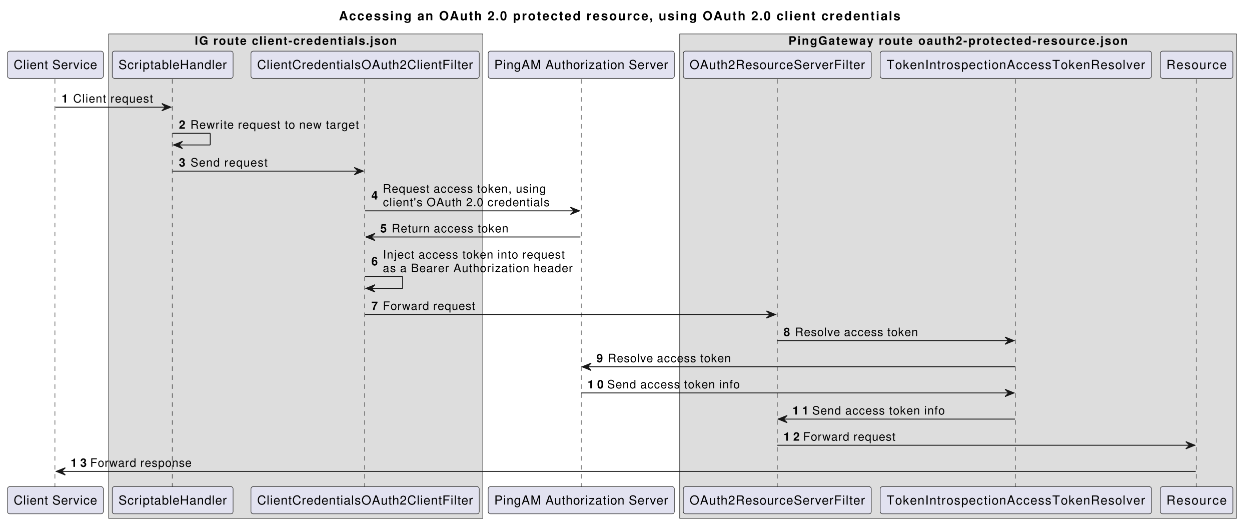
b. Access the route using the access token and and note that the request isn't authorized because the token is revoked:

```
$ curl -v \
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--header "Authorization: Bearer ${mytoken}" \
https://ig.example.com:8443/rs-introspect-cache
```

```
...
HTTP/1.1 401 Unauthorized
```

Client credentials grant

This example shows how a client service accesses an OAuth 2.0-protected resource by using its OAuth 2.0 client credentials.


Accessing an OAuth 2.0 protected resource, using OAuth 2.0 client credentials
IG route client-credentials.json
PingGateway route oauth2-protected-resource.json
Client Service
ScriptableHandler
ClientCredentialsOAuth2ClientFilter
PingAM Authorization Server
OAuth2ResourceServerFilter
TokenIntrospectionAccessTokenResolver
Resource
1 Client request
2 Rewrite request to new target
3 Send request
4 Request access token, using client's OAuth 2.0 credentials
5 Return access token
6 Inject access token into request as a Bearer Authorization header
7 Forward request
8 Resolve access token
9 Resolve access token
10 Send access token info
11 Send access token info
12 Forward request
13 Forward response

1. Set up the AM as an Authorization Server:
   a. Register a PingGateway agent with the following values, as described in Register a PingGateway agent in AM:
      - **Agent ID**: `ig_agent`
      - **Password**: `password`
      - **Token Introspection**: `Realm Only`

      IMPORTANT

      Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

   b. (Optional) Authenticate the agent to AM as described in Authenticate a PingGateway agent to AM.

      IMPORTANT

      PingGateway agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

   c. Create an OAuth 2.0 Authorization Server:

i. Select **Services** > **Add a Service** > **OAuth2 Provider**.

ii. Add a service with the default values.

d. Create an OAuth 2.0 client to request access tokens, using client credentials for authentication:

i. Select **Applications** > **OAuth 2.0** > **Clients**, and add a client with the following values:

- **Client ID** : `client-service`
- **Client secret** : `password`
- **Scope(s)** : `client-scope`

ii. (Optional) On the **Core** tab, switch to using a client secret associated with a secret label by setting a **Secret Label Identifier** and mapping the label to a secret.

To learn more, read Create a client profile and Map and rotate secrets in the AM documentation.

iii. On the **Advanced** tab, select the following value:

- **Grant Types** : `Client Credentials`

2. Set up PingGateway:

a. Set up PingGateway for HTTPS, as described in Configure PingGateway for TLS (server-side).

b. Set an environment variable for the PingGateway agent password, and then restart PingGateway:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

c. Add the following route to PingGateway:

Linux Windows

```
$HOME/.openig/config/routes/oauth2-protected-
resource.json
```

```
{
  "name": "oauth2-protected-resource",
  "condition": "${find(request.uri.path, '^/oauth2-
protected-resource')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [ "client-scope" ],
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name":
"TokenIntrospectionAccessTokenResolver-1",
              "type":
"TokenIntrospectionAccessTokenResolver",
              "config": {
                "amService": "AmService-1",
                "providerHandler": {
                  "type": "Chain",
                  "config": {
                    "filters": [
                      {
                        "type":
"HttpBasicAuthenticationClientFilter",
                        "config": {
                          "username": "ig_agent",
                          "passwordSecretId":
"agent.secret.id",
                          "secretsProvider":
"SystemAndEnvSecretStore-1"
                        }
                      }
                    ],
                    "handler": "ForgeRockClientHandler"
                  }
                }
              }
            }
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": "<html><body><h2>Access Granted</h2>
</body></html>"
        }
      }
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/oauth2-protected-resource` .
- The `OAuth2ResourceServerFilter` expects an OAuth 2.0 access token in the header of the incoming request, with the scope `client-scope` .
- The filter uses a TokenIntrospectionAccessTokenResolver to resolve the access token. The introspect endpoint is protected with HTTP Basic Authentication, and the `providerHandler` uses an HttpBasicAuthenticationClientFilter to provide the resource server credentials.
- For convenience in this test, `"requireHttps"` is false. In production environments, set it to true.
- After the filter successfully validates the access token, it creates a new context from the Authorization Server response, containing information

about the access token.

- The StaticResponseHandler returns a message that access is granted.

d. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/client-credentials.json
```

```
{
  "name": "client-credentials",
  "baseURI": "http://ig.example.com:8080",
  "condition" : "${find(request.uri.path, '^/client-
credentials')}",
  "heap" : [ {
    "name" : "clientSecretAccessTokenExchangeHandler",
    "type" : "Chain",
    "capture" : "all",
    "config" : {
      "filters" : [ {
        "type" : "ClientSecretBasicAuthenticationFilter",
        "config" : {
          "clientId" : "client-service",
          "clientSecretId" : "client.secret.id",
          "secretsProvider" : {
            "type" : "Base64EncodedSecretStore",
            "config" : {
              "secrets" : {
                "client.secret.id" : "cGFzc3dvcmQ="
              }
            }
          }
        }
      } ],
      "handler" : "ForgeRockClientHandler"
    }
  }, {
    "name" : "oauth2EnabledClientHandler",
    "type" : "Chain",
    "capture" : "all",
    "config" : {
      "filters" : [ {
        "type" : "ClientCredentialsOAuth2ClientFilter",
        "config" : {
          "tokenEndpoint" :
"http://am.example.com:8088/openam/oauth2/access_token",
          "endpointHandler":
"clientSecretAccessTokenExchangeHandler",
          "scopes" : [ "client-scope" ]
        }
      } ],
      "handler" : "ForgeRockClientHandler"
    }
  } ],
  "handler" : {
    "type" : "ScriptableHandler",
    "config" : {
      "type" : "application/x-groovy",
      "clientHandler" : "oauth2EnabledClientHandler",
      "source" : [ "request.uri.path = '/oauth2-protected-
resource'", "return http.send(context, request);" ]
    }
  }
}
```

Note the following features of the route:

- The route matches requests to `/client-credentials`.
- The ScriptableHandler rewrites the request to target it to `/oauth2-protected-resource`, and then calls the HTTP client, that has been redefined to use the oauth2EnabledClientHandler.
- The oauth2EnabledClientHandler calls the ClientCredentialsOAuth2ClientFilter to obtain an access token from AM.
- The ClientCredentialsOAuth2ClientFilter calls the clientSecretAccessTokenExchangeHandler to exchange tokens on the authorization endpoint.
- The clientSecretAccessTokenExchangeHandler calls a ClientSecretBasicAuthenticationFilter to authenticate the client through the HTTP basic access authentication scheme, and a ForgeRockClientHandler to propagate the request.
- The route `oauth2-protected-resource.json` uses the AM introspection endpoint to resolve the access token and display its contents.

3. Test the setup:

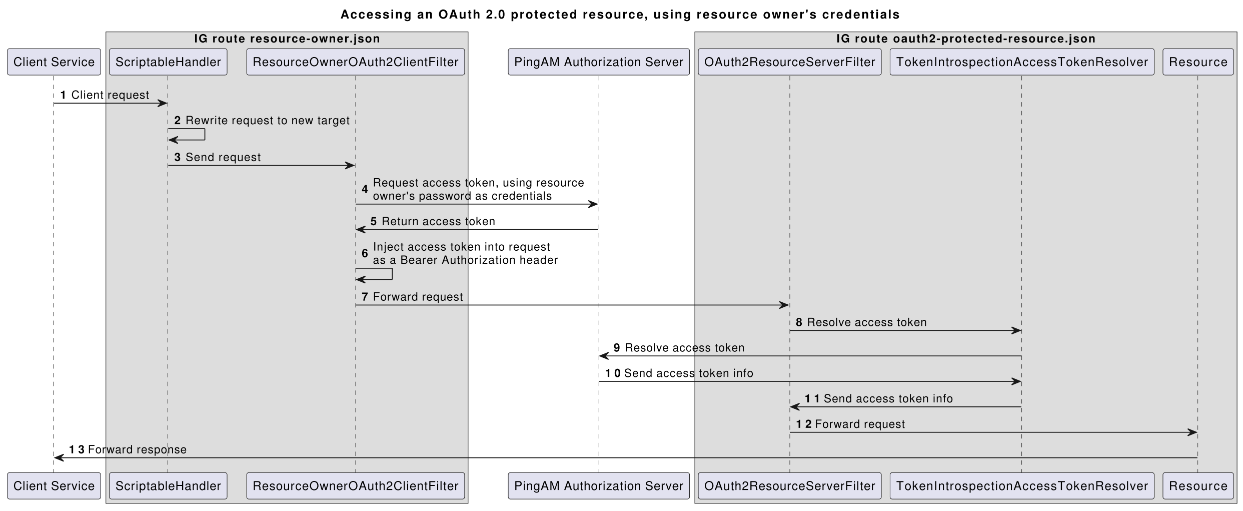
   a. In your browser's privacy or incognito mode, go to to https://ig.example.com:8443/client-credentials.

   b. If you see warnings that the site isn't secure, respond to the warnings to access the site.

A message shows that access is granted.

Resource owner password credentials grant

This example shows how a client service accesses an OAuth 2.0-protected resource by using resource owner password credentials.


Accessing an OAuth 2.0 protected resource, using resource owner's credentials
IG route resource-owner.json
IG route oauth2-protected-resource.json
Client Service
ScriptableHandler
ResourceOwnerOAuth2ClientFilter
PingAM Authorization Server
OAuth2ResourceServerFilter
TokenIntrospectionAccessTokenResolver
Resource
1 Client request
2 Rewrite request to new target
3 Send request
4 Request access token, using resource owner's password as credentials
5 Return access token
6 Inject access token into request as a Bearer Authorization header
7 Forward request
8 Resolve access token
9 Resolve access token
10 Send access token info
11 Send access token info
12 Forward request
13 Forward response

IMPORTANT

This procedure uses the *Resource Owner Password Credentials* grant type. As suggested in The OAuth 2.0 Authorization Framework, use other grant types whenever possible.

1. Set up the AM as an Authorization Server:
   a. Register a PingGateway agent with the following values, as described in Register a PingGateway agent in AM:
      - **Agent ID**: `ig_agent`
      - **Password**: `password`
      - **Token Introspection**: `Realm Only`

      IMPORTANT

      Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

   b. (Optional) Authenticate the agent to AM as described in Authenticate a PingGateway agent to AM.

IMPORTANT

IMPORTANT

PingGateway agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

c. Create an OAuth 2.0 Authorization Server:

   i. Select **Services** > **Add a Service** > **OAuth2 Provider**.

   ii. Add a service with the default values.

d. Create an OAuth 2.0 client to request access tokens, using the resource owner's password for authentication:

   i. Select **Applications** > **OAuth 2.0** > **Clients**, and add a client with the following values:

      - **Client ID** : `resource-owner-client`
      - **Client secret** : `password`
      - **Scope(s)** : `client-scope`

   ii. (Optional) On the **Core** tab, switch to using a client secret associated with a secret label by setting a **Secret Label Identifier** and mapping the label to a secret.

      To learn more, read Create a client profile and Map and rotate secrets in the AM documentation.

   iii. On the **Advanced** tab, select the following value:

      - **Grant Types** : `Resource Owner Password Credentials`

2. Set up PingGateway:

   a. Set up PingGateway for HTTPS, as described in Configure PingGateway for TLS (server-side).

   b. Set an environment variable for the PingGateway agent password, and then restart PingGateway:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

   The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

   c. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/oauth2-protected-
```

```
resource.json
```

```
{
  "name": "oauth2-protected-resource",
  "condition": "${find(request.uri.path, '^/oauth2-
protected-resource')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [ "client-scope" ],
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name":
"TokenIntrospectionAccessTokenResolver-1",
              "type":
"TokenIntrospectionAccessTokenResolver",
              "config": {
                "amService": "AmService-1",
                "providerHandler": {
                  "type": "Chain",
                  "config": {
                    "filters": [
                      {
                        "type":
"HttpBasicAuthenticationClientFilter",
                        "config": {
                          "username": "ig_agent",
                          "passwordSecretId":
"agent.secret.id",
                          "secretsProvider":
"SystemAndEnvSecretStore-1"
                        }
                      }
                    ],
                    "handler": "ForgeRockClientHandler"
                  }
                }
              }
            }
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": "<html><body><h2>Access Granted</h2>
</body></html>"
        }
      }
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/oauth2-protected-resource`.
- The `OAuth2ResourceServerFilter` expects an OAuth 2.0 access token in the header of the incoming request, with the scope `client-scope`.
- The filter uses a TokenIntrospectionAccessTokenResolver to resolve the access token. The introspect endpoint is protected with HTTP Basic

Authentication, and the `providerHandler` uses an HttpBasicAuthenticationClientFilter to provide the resource server credentials.

- For convenience in this test, `"requireHttps"` is false. In production environments, set it to true.
- After the filter successfully validates the access token, it creates a new context from the Authorization Server response, containing information about the access token.
- The StaticResponseHandler returns a message that access is granted.

d. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/resource-owner.json
```

```
{
  "name": "resource-owner",
  "baseURI": "http://ig.example.com:8080",
  "condition" : "${find(request.uri.path, '^/resource-
owner')}",
  "heap" : [ {
    "name" : "clientSecretAccessTokenExchangeHandler",
    "type" : "Chain",
    "capture" : "all",
    "config" : {
      "filters" : [ {
        "type" : "ClientSecretBasicAuthenticationFilter",
        "config" : {
          "clientId" : "resource-owner-client",
          "clientSecretId" : "client.secret.id",
          "secretsProvider" : {
            "type" : "Base64EncodedSecretStore",
            "config" : {
              "secrets" : {
                "client.secret.id" : "cGFzc3dvcmQ="
              }
            }
          }
        }
      } ],
      "handler" : "ForgeRockClientHandler"
    }
  }, {
    "name" : "oauth2EnabledClientHandler",
    "type" : "Chain",
    "capture" : "all",
    "config" : {
      "filters" : [ {
        "type" : "ResourceOwnerOAuth2ClientFilter",
        "config" : {
          "tokenEndpoint" :
"http://am.example.com:8088/openam/oauth2/access_token",
          "endpointHandler":
"clientSecretAccessTokenExchangeHandler",
          "scopes" : [ "client-scope" ],
          "username" : "demo",
          "passwordSecretId" : "user.password.secret.id",
          "secretsProvider" : {
            "type" : "Base64EncodedSecretStore",
            "config" : {
              "secrets" : {
                "user.password.secret.id" : "Q2g0bmczMXQ="
              }
            }
          }
        }
      } ],
      "handler" : "ForgeRockClientHandler"
    }
  } ],
  "handler" : {
    "type" : "ScriptableHandler",
    "config" : {
      "type" : "application/x-groovy",
      "clientHandler" : "oauth2EnabledClientHandler",
      "source" : [ "request.uri.path = '/oauth2-protected-
resource'", "return http.send(context, request);" ]
    }
  }
}
```

Note the following features of the route:

- The route matches requests to `/resource-owner`.
- The ScriptableHandler rewrites the request to target it to `/oauth2-protected-resource`, and then calls the HTTP client, that has been redefined to use the oauth2EnabledClientHandler.

        - The oauth2EnabledClientHandler calls the ResourceOwnerOAuth2ClientFilter to obtain an access token from AM.
        - The ResourceOwnerOAuth2ClientFilter calls the clientSecretAccessTokenExchangeHandler to exchange tokens on the authorization endpoint. The demo user authenticates with their username and password.
        - The clientSecretAccessTokenExchangeHandler calls a ClientSecretBasicAuthenticationFilter to authenticate the client through the HTTP basic access authentication scheme, and a ForgeRockClientHandler to propagate the request.
        - The route `oauth2-protected-resource.json` uses the AM introspection endpoint to resolve the access token and display its contents.

3. Test the setup:

    a. In your browser's privacy or incognito mode, go to to https://ig.example.com:8443/resource-owner.

    b. If you see warnings that the site isn't secure, respond to the warnings to access the site.

       A message shows that access is granted.

OpenID Connect

The following pages provide an overview of how PingGateway supports OpenID Connect 1.0 (OIDC), an authentication layer built on OAuth 2.0. The pages show how to set up PingGateway as an OIDC relying party in different deployment scenarios.

About PingGateway with OIDC

PingGateway supports OIDC deployments where the identity provider holds the protected resource third-party applications want to access.

OIDC specifications refer to the following entities:

- *End user*: An OAuth 2.0 resource owner whose user information the application needs to access.

  The end user wants to use an application through an existing identity provider account without signing up and creating credentials for another web service.

- *Relying Party* (RP): An OAuth 2.0 client that needs access to the end user's protected user information.

For example, an online mail application needs to know which end user is accessing the application to present the correct inbox.

As another example, an online shopping site needs to know which end user is accessing the site to present the right offerings, account, and shopping cart.

- *OpenID Provider* (OP): An OAuth 2.0 Authorization Server and resource server that holds the user information and grants access.

  The OP requires the end user to give the RP permission to access to some of its user information. Because OIDC defines unique identification for an account (subject identifier + issuer identifier), the RP can use that identification to bind its own user profile to a remote identity.

  For the online mail application, this key could be used to access the mailboxes and related account information. For the online shopping site, this key could be used to access the offerings, account, shopping cart and others. The key makes it possible to serve users as if they had local accounts.
- *UserInfo*: The protected resource that the third-party application wants to access. The information about the authenticated end user is expressed in a standard format. The user-info endpoint is hosted on the Authorization Server and is protected with OAuth 2.0.

When PingGateway acts as an RP, its role is to retrieve user information from the OP and to inject the information into the context for use by later filters and handlers.

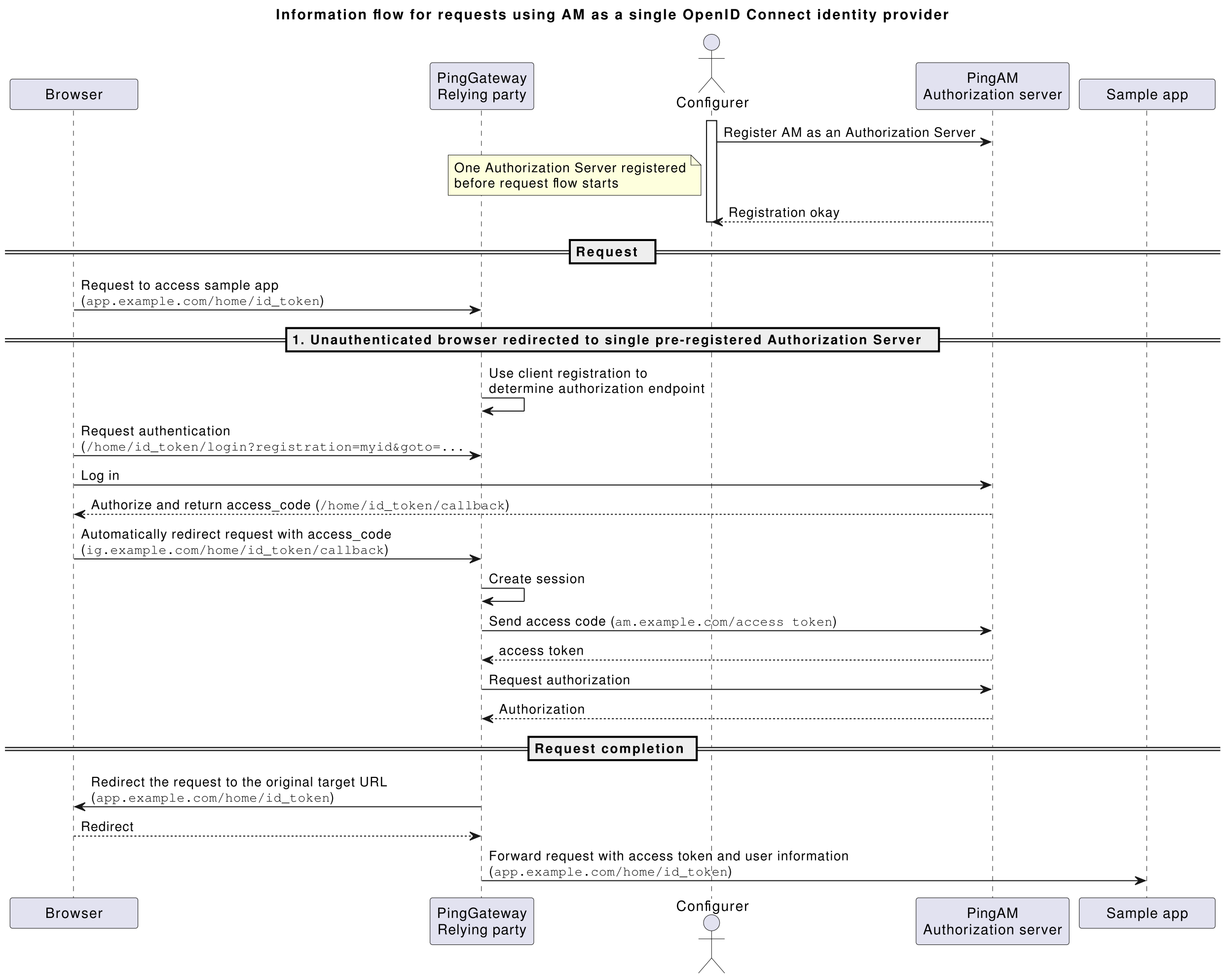
Next steps

- AM as OIDC provider
- PingOne Advanced Identity Cloud as OIDC provider
- PingOne as OIDC provider
- Multiple OIDC providers
- Discovery and dynamic registration
- ID token validation

AM as OIDC provider

This page gives an example of how to set up AM as an OIDC provider and PingGateway as a relying party for browser requests to the home page of the sample application.

The following sequence diagram shows the flow of information for a request to access the home page of the sample application. AM is the single, preregistered OIDC provider and PingGateway is the relying party:


Information flow for requests using AM as a single OpenID Connect identity provider
Browser
PingGateway Relying party
Configurer
PingAM Authorization server
Sample app
Register AM as an Authorization Server
One Authorization Server registered before request flow starts
Registration okay
Request
Request to access sample app (app.example.com/home/id_token)
1. Unauthenticated browser redirected to single pre-registered Authorization Server
Use client registration to determine authorization endpoint
Request authentication (/home/id_token/login?registration=myid&goto=...
Log in
Authorize and return access_code (/home/id_token/callback)
Automatically redirect request with access_code (ig.example.com/home/id_token/callback)
Create session
Send access code (am.example.com/access token)
access token
Request authorization
Authorization
Request completion
Redirect the request to the original target URL (app.example.com/home/id_token)
Redirect
Forward request with access token and user information (app.example.com/home/id_token)

Before you begin, prepare AM, PingGateway, and the sample application. Learn more in the example installation for this guide.

1. Set up AM as an OIDC provider:
    a. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:
        - `https://ig.example.com:8443/*`
        - `https://ig.example.com:8443/*?*`
    b. Create an OAuth 2.0 Authorization Server:
        i. Select **Services** > **Add a Service** > **OAuth2 Provider**.
        ii. Add a service with the default values.
    c. Create an OAuth 2.0 Client to request OAuth 2.0 access tokens:
        i. Select **Applications** > **OAuth 2.0** > **Clients**.
        ii. Add a client with the following values:
            - **Client ID**: `oidc_client`
            - **Client secret**: `password`
            - **Redirection URIs**: `https://ig.example.com:8443/home/id_token/callback`

- **Scope(s)**: `openid`, `profile`, and `email`

iii. (Optional) On the **Core** tab, switch to using a client secret associated with a secret label by setting a **Secret Label Identifier** and mapping the label to a secret.

To learn more, read Create a client profile and Map and rotate secrets in the AM documentation.

iv. On the **Advanced** tab, select the following values:

- **Grant Types**: `Authorization Code`

2. Set up PingGateway:

a. Set up PingGateway for HTTPS, as described in Configure PingGateway for TLS (server-side).

b. Set an environment variable for `oidc_client`, and then restart PingGateway:

```
$ export OIDC_SECRET_ID='cGFzc3dvcmQ='
```

c. Add the following route to PingGateway to serve the sample application .css and other static resources:

Linux | Windows

```
$HOME/.openig/config/routes/00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or
matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

d. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/07-openid.json
```

```
{
  "name": "07-openid",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path,
'^/home/id_token')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AuthenticatedRegistrationHandler-1",
      "type": "Chain",
      "config": {
        "filters": [
          {
            "name":
"ClientSecretBasicAuthenticationFilter-1",
            "type":
"ClientSecretBasicAuthenticationFilter",
            "config": {
              "clientId": "oidc_client",
              "clientSecretId": "oidc.secret.id",
              "secretsProvider": "SystemAndEnvSecretStore-
1"
            }
          }
        ],
        "handler": "ForgeRockClientHandler"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "AuthorizationCodeOAuth2ClientFilter-1",
          "type": "AuthorizationCodeOAuth2ClientFilter",
          "config": {
            "clientEndpoint": "/home/id_token",
            "failureHandler": {
              "type": "StaticResponseHandler",
              "config": {
                "status": 500,
                "headers": {
                  "Content-Type": [
                    "text/plain"
                  ]
                },
                "entity": "Error in OAuth 2.0 setup."
              }
            },
            "registrations": [
              {
                "name": "oidc-user-info-client",
                "type": "ClientRegistration",
                "config": {
                  "clientId": "oidc_client",
                  "issuer": {
                    "name": "Issuer",
                    "type": "Issuer",
                    "config": {
                      "wellKnownEndpoint":
"http://am.example.com:8088/openam/oauth2/.well-
known/openid-configuration"
                    }
                  },
                  "scopes": [
                    "openid",
                    "profile",
                    "email"
                  ],
                  "authenticatedRegistrationHandler":
"AuthenticatedRegistrationHandler-1"
                }
              }
            ],
            "requireHttps": false,
            "cacheExpiration": "disabled"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

For information about how to set up the PingGateway route in Studio, see OpenID Connect in Structured Editor.

Notice the following features about the route:

- The route matches requests to `/home/id_token` .
- The `AuthorizationCodeOAuth2ClientFilter` enables PingGateway to act as a relying party. It uses a single client registration defined inline.
- The filter has a base client endpoint of `/home/id_token` , which creates the following service URIs:
  - Requests to `/home/id_token/login` start the delegated authorization process.
  - Requests to `/home/id_token/callback` are expected as redirects from the OAuth 2.0 Authorization Server (OIDC provider). This is why the redirect URI in the client profile in AM is set to `https://ig.example.com:8443/home/id_token/callback` .
  - Requests to `/home/id_token/logout` remove the authorization state for the end user, and redirect to the specified URL if a `goto` parameter is provided.

    These endpoints are implicitly reserved. Attempts to access them directly can cause undefined errors.
- For convenience in this test, `"requireHttps"` is false. In production environments, set it to true. So that you see the delegated authorization process when you make a request, `"requireLogin"` has the default value `true` .
- The target for storing authorization state information is `${attributes.openid}` . This is where subsequent filters and handlers can find access tokens and user information.

3. Test the setup:

   a. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/id_token.

      The AM login page is displayed.

   b. Log in to AM as user `demo` , password `Ch4ng31t` , and then allow the application to access user information.

      The home page of the sample application is displayed.

Authenticate automatically to the sample application

To authenticate automatically to the sample application, change the last name of the user `demo` to match the password `Ch4ng31t` , and add a StaticRequestFilter like the following to the end of the chain in `07-openid.json` :

```
{
  "type": "StaticRequestFilter",
  "config": {
    "method": "POST",
    "uri": "http://app.example.com:8081/login",
    "form": {
      "username": [
        "${attributes.openid.user_info.sub}"
      ],
      "password": [
        "${attributes.openid.user_info.family_name}"
      ]
    }
  }
}
```

The StaticRequestFilter retrieves the username and password from the context, and replaces the original HTTP GET request with an HTTP POST login request containing credentials.

Multiple OIDC providers

This page shows OIDC with two identity providers.

Client registrations for an AM identity provider and PingOne Advanced Identity Cloud identity provider are declared in the heap. The Nascar page helps the user to choose an identity provider.

1. Set up AM as the first identity provider, as described in AM as OIDC provider.
2. Set up PingOne Advanced Identity Cloud as a second identity provider, as described in PingOne Advanced Identity Cloud as an OpenID Connect provider.
3. Add the following route to PingGateway, replacing the value for the property `amInstanceUrl`:

   Linux | Windows

   ```
   $HOME/.openig/config/routes/07-openid-nascar.json
   ```

   ```
   {
     "heap": [
       {
   ```

```
        "name": "SystemAndEnvSecretStore-1",
        "type": "SystemAndEnvSecretStore"
      },
      {
        "name": "AuthenticatedRegistrationHandler-1",
        "type": "Chain",
        "config": {
          "filters": [
            {
              "name": "ClientSecretBasicAuthenticationFilter-1",
              "type": "ClientSecretBasicAuthenticationFilter",
              "config": {
                "clientId": "oidc_client",
                "clientSecretId": "oidc.secret.id",
                "secretsProvider": "SystemAndEnvSecretStore-1"
              }
            }
          ],
          "handler": "ForgeRockClientHandler"
        }
      },
      {
        "name": "openam",
        "type": "ClientRegistration",
        "config": {
          "clientId": "oidc_client",
          "issuer": {
            "name": "am_issuer",
            "type": "Issuer",
            "config": {
              "wellKnownEndpoint":
"http://am.example.com:8088/openam/oauth2/.well-known/openid-
configuration"
            }
          },
          "scopes": [
            "openid",
            "profile",
            "email"
          ],
          "authenticatedRegistrationHandler":
"AuthenticatedRegistrationHandler-1"
        }
      },
      {
```

```
      "name": "idcloud",
      "type": "ClientRegistration",
      "config": {
        "clientId": "oidc_client",
        "issuer": {
          "name": "idc_issuer",
          "type": "Issuer",
          "config": {
            "wellKnownEndpoint": "&
{amInstanceUrl}/oauth2/realms/alpha/.well-known/openid-
configuration"
          }
        },
        "scopes": [
          "openid",
          "profile",
          "email"
        ],
        "authenticatedRegistrationHandler":
"AuthenticatedRegistrationHandler-1"
      }
    },
    {
      "name": "NascarPage",
      "type": "StaticResponseHandler",
      "config": {
        "status": 200,
        "headers": {
          "Content-Type": [ "text/html; charset=UTF-8" ]
        },
        "entity": [
          "<html>",
          "  <body>",
          "    <p><a href='/home/id_token/login?
registration=oidc_client&issuer=am_issuer&goto=${urlEncodeQuer
yParameterNameOrValue('https://ig.example.com:8443/home/id_tok
en')}'>Access Management login</a></p>",
          "    <p><a href='/home/id_token/login?
registration=oidc_client&issuer=idc_issuer&goto=${urlEncodeQue
ryParameterNameOrValue('https://ig.example.com:8443/home/id_to
ken')}'>Identity Cloud login</a></p>",
          "  </body>",
          "</html>"
        ]
      }
    }
  ],
  "name": "07-openid-nascar",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/id_token')}",
  "properties": {
    "amInstanceUrl": "https://myTenant.forgeblocks.com/am"
  },
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "AuthorizationCodeOAuth2ClientFilter",
          "config": {
            "clientEndpoint": "/home/id_token",
            "failureHandler": {
              "type": "StaticResponseHandler",
              "config": {
                "comment": "Trivial failure handler for
debugging only",
                "status": 500,
                "headers": {
                  "Content-Type": [ "text/plain; charset=UTF-
8" ]
                },
                "entity": "${contexts.oauth2Failure.error}:
${contexts.oauth2Failure.description}"
              }
            },
            "loginHandler": "NascarPage",
            "registrations": [ "openam", "idcloud" ],
            "requireHttps": false,
            "cacheExpiration": "disabled"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Consider the differences with `07-openid.json`:

- The heap objects `openam` and `idcloud` define client registrations.

   - The StaticResponseHandler provides links to the client registrations.
   - The AuthorizationCodeOAuth2ClientFilter uses a `loginHandler` to allow users to choose a client registration and therefore an identity provider.

4. Test the setup:

   a. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/id_token.

      The Nascar page offers the choice of identity provider.

   b. Using the following credentials, select a provider, log in, and allow the application to access user information:

      - AM: user `demo`, password `Ch4ng31t`.
      - PingOne Advanced Identity Cloud: user `demo`, password `Ch4ng3!t`

        The home page of the sample application is displayed.

Discovery and dynamic registration

OIDC defines mechanisms for discovering and dynamically registering with an identity provider that isn't known in advance, as specified in the following publications: OpenID Connect Discovery, OpenID Connect Dynamic Client Registration, and OAuth 2.0 Dynamic Client Registration Protocol.

In dynamic registration, issuer and client registrations are generated dynamically. They are held in memory and can be reused, but don't persist when PingGateway is restarted.

This section builds on the example in AM as OIDC provider to give an example of discovering and dynamically registering with an identity provider that isn't known in advance. In this example, the client sends a signed JWT to the Authorization Server.

To facilitate the example, a WebFinger service is embedded in the sample application. In a normal deployment, the WebFinger server is likely to be a service on the issuer's domain.

1. Set up a key

   a. Locate a directory for secrets, and go to it:

```
$ cd /path/to/secrets
```

   b. Create a key:

```
$ keytool -genkey \
  -alias myprivatekeyalias \
  -keyalg RSA \
  -keysize 2048 \
  -keystore keystore.p12 \
  -storepass keystore \
  -storetype PKCS12 \
  -keypass keystore \
  -validity 360 \
  -dname "CN=ig.example.com, OU=example, O=com, L=fr,
ST=fr, C=fr"
```

2. Set up AM:

   a. Set up AM as described in AM as OIDC provider.

   b. Select the user `demo`, and change the last name to `Ch4ng31t`. For this example, the last name must be the same as the password.

   c. Configure the OAuth 2.0 Authorization Server for dynamic registration:

      i. Select **Services** > **OAuth2 Provider**.

      ii. On the **Advanced** tab, add the following scopes to **Client Registration Scope Allowlist**: `openid`, `profile`, `email`.

      iii. On the **Client Dynamic Registration** tab, select these settings:

         - **Allow Open Dynamic Client Registration**: Enabled
         - **Generate Registration Access Tokens**: Disabled

   d. Configure the authentication method for the OAuth 2.0 Client:

      i. Select **Applications** > **OAuth 2.0** > **Clients**.

      ii. Select `oidc_client`, and on the **Advanced** tab, select **Token Endpoint Authentication Method**: `private_key_jwt`.

3. Set up PingGateway:

   a. In the PingGateway configuration, set an environment variable for the keystore password, and then restart PingGateway:

```
$ export KEYSTORE_SECRET_ID='a2V5c3RvcmU='
```

   The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

   b. Add the following route to PingGateway to serve the sample application .css and other static resources:

Linux | Windows

```
$HOME/.openig/config/routes/00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or
matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

c. Add the following script to PingGateway:

Linux | Windows

```
$HOME/.openig/scripts/groovy/discovery.groovy
```

```
/*
 * OIDC discovery with the sample application
 */
response = new Response(Status.OK)
response.getHeaders().put(ContentTypeHeader.NAME,
"text/html");
response.entity = """
<!doctype html>
<html>
  <head>
    <title>OpenID Connect Discovery</title>
    <meta charset='UTF-8'>
  </head>
  <body>
    <form id='form' action='/discovery/login?'>
      Enter your user ID or email address:
        <input type='text' id='discovery' name='discovery'
          placeholder='demo or demo@example.com' />
        <input type='hidden' name='goto'

value='${contexts.idpSelectionLogin.originalUri}' />
    </form>
    <script>
      // Make sure sampleAppUrl is correct for your sample
app.
      window.onload = function() {
      document.getElementById('form').onsubmit =
function() {
      // Fix the URL if not using the default settings.
      var sampleAppUrl = 'http://app.example.com:8081/';
      var discovery =
document.getElementById('discovery');
      discovery.value = sampleAppUrl +
discovery.value.split('@', 1)[0];
      };
    };
    </script>
  </body>
</html>""" as String
return response
```

The script transforms the input into a `discovery` value for PingGateway. This is not a requirement for deployment, only a convenience for the purposes of this example. Alternatives are described in the discovery protocol specification.

d. Add the following route to PingGateway, replacing `/path/to/secrets/keystore.p12` with your path:

Linux | Windows

```
$HOME/.openig/config/routes/07-discovery.json
```

```
{
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "SecretsProvider-1",
      "type": "SecretsProvider",
      "config": {
        "stores": [
          {
            "type": "KeyStoreSecretStore",
            "config": {
              "file": "/path/to/secrets/keystore.p12",
              "mappings": [
                {
                  "aliases": [ "myprivatekeyalias" ],
                  "secretId":
"private.key.jwt.signing.key"
```

```
                }
              ],
              "storePasswordSecretId":
"keystore.secret.id",
              "storeType": "PKCS12",
              "secretsProvider": "SystemAndEnvSecretStore-
1"
            }
          }
        ]
      }
    },
    {
      "name": "DiscoveryPage",
      "type": "ScriptableHandler",
      "config": {
        "type": "application/x-groovy",
        "file": "discovery.groovy"
      }
    }
  ],
  "name": "07-discovery",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/discovery')}",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "DynamicallyRegisteredClient",
          "type": "AuthorizationCodeOAuth2ClientFilter",
          "config": {
            "clientEndpoint": "/discovery",
            "requireHttps": false,
            "requireLogin": true,
            "target": "${attributes.openid}",
            "failureHandler": {
              "type": "StaticResponseHandler",
              "config": {
                "comment": "Trivial failure handler for
debugging only",
                "status": 500,
                "headers": {
                  "Content-Type": [ "text/plain;
charset=UTF-8" ]
```

```
                },
                "entity":
"${contexts.oauth2Failure.error}:
${contexts.oauth2Failure.description}"
              }
            },
            "loginHandler": "DiscoveryPage",
            "discoverySecretId":
"private.key.jwt.signing.key",
            "tokenEndpointAuthMethod": "private_key_jwt",
            "secretsProvider": "SecretsProvider-1",
            "metadata": {
              "client_name": "My Dynamically Registered
Client",
              "redirect_uris": [
"http://ig.example.com:8080/discovery/callback" ],
              "scopes": [ "openid", "profile", "email" ]
            }
          }
        },
        {
          "type": "StaticRequestFilter",
          "config": {
            "method": "POST",
            "uri": "http://app.example.com:8081/login",
            "form": {
              "username": [
                "${attributes.openid.user_info.name}"
              ],
              "password": [

"${attributes.openid.user_info.family_name}"
              ]
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Consider the differences with `07-openid.json`:

- The route matches requests to `/discovery`.

- The AuthorizationCodeOAuth2ClientFilter uses `DiscoveryPage` as the login handler, and specifies metadata to prepare the dynamic registration request.
- `DiscoveryPage` uses a ScriptableHandler and script to provide the `discovery` parameter and `goto` parameter.

  If there is a match, then it can use the issuer's registration endpoint and avoid an additional request to look up the user's issuer using the WebFinger protocol.

  If there is no match, PingGateway uses the `discovery` value as the `resource` for a WebFinger request using the OIDC discovery protocol.
- PingGateway uses the `discovery` parameter to find an identity provider. PingGateway extracts the domain host and port from the value, and attempts to find a match in the `supportedDomains` lists for issuers configured for the route.
- When `discoverySecretId` is set, the `tokenEndpointAuthMethod` is always `private_key_jwt`. Clients send a signed JWT to the Authorization Server.

  Redirects PingGateway to the end user's browser, using the `goto` parameter, after the process is complete and PingGateway has injected the OIDC user information into the context.

4. Test the setup:

   a. Log out of AM, and clear any cookies.

   b. Go to http://ig.example.com:8080/discovery.

   c. Enter the following email address: `demo@example.com`. The AM login page is displayed.

   d. Log in as user `demo`, password `Ch4ng31t`, and then allow the application to access user information. The sample application returns the user's page.

ID token validation

This page uses an IdTokenValidationFilter to validate an ID token.

1. Set up AM:

   a. Set up AM as described in Validate access tokens with introspection.

   b. Select **Applications** > **OAuth 2.0** > **Clients** and add the additional scope `openid to client-application`.

2. Set up PingGateway:

   a. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/idtokenvalidation.json
```

```
{
  "name": "idtokenvalidation",
  "condition": "${find(request.uri.path,
'^/idtokenvalidation')}",
  "capture": "all",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [{
        "type": "IdTokenValidationFilter",
        "config": {
          "idToken": "<id_token_value>",
          "audience": "client-application",
          "issuer":
"http://am.example.com:8088/openam/oauth2",
          "failureHandler": {
            "type": "ScriptableHandler",
            "config": {
              "type": "application/x-groovy",
              "source": [
                "def response = new
Response(Status.FORBIDDEN)",
                "response.headers['Content-Type'] =
'text/html; charset=utf-8'",
                "def errors =
contexts.jwtValidationError.violations.collect{it.descript
ion}",
                "def display = \"<html>Can't validate
id_token:<br> ${contexts.jwtValidationError.jwt} \"",
                "display <<=\"<br><br>For the following
errors:<br> ${errors.join(\"<br>\")}</html>\"",
                "response.entity=display as String",
                "return response"
              ]
            }
          },
          "verificationSecretId": "verify",
          "secretsProvider": {
            "type": "JwkSetSecretStore",
            "config": {
              "jwkUrl":
"http://am.example.com:8088/openam/oauth2/connect/jwk_uri"
            }
          }
        }
      }],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": "<html><body>Validated id_token:<br>
${contexts.jwtValidation.value}</body></html>"
        }
      }
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/idtokenvalidation`.
- A SecretsProvider declares a JwkSetSecretStore to validate secrets for signed JWTs. The JwkSetSecretStore specifies a URL to a JWK set on AM that contains the signing keys.
- The property `verificationSecretId` is configured with an arbitrary value. If this property isn't configured, the filter doesn't verify the signature of tokens.
- The JwkSetSecretStore specifies the URL to a JWK set on AM, that contains verification keys identified by a `kid`. PingGateway validates the token signature as follows:
  - If the value of a `kid` in the JWK set matches a `kid` in the the signed JWT, the JwkSetSecretStore verifies the signature.
  - If the JWT doesn't have a `kid`, or if the JWK set doesn't contain a key with the same value, the JwkSetSecretStore looks for valid secrets with the same purpose as the value of `verificationSecretId`.
- If the filter validates the token, the StaticResponseHandler displays the token value from the context `${contexts.jwtValidation.value}`. Otherwise, the ScriptableHandler displays the token value and a list of

violations from the context `${contexts.jwtValidationError.violations}`

3. Test the setup:

   a. In a terminal window, use a **`curl`** command similar to the following to retrieve an id_token:

   ```
   $ curl -s \
   --user "client-application:password" \
   --data
   "grant_type=password&username=demo&password=Ch4ng31t&scope
   =openid" \
   http://am.example.com:8088/openam/oauth2/access_token

   {
    "access_token":"...",
    "scope":"openid",
    "id_token":"...",
    "token_type":"Bearer",
    "expires_in":3599
   }
   ```

   b. In the route, replace `<id_token_value>` with the value of the `id_token` returned in the previous step.

   c. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/idtokenvalidation.

      The validated token is displayed.

   d. In the route, invalidate the token by changing the value of the audience or issuer, and then access the route again.

      The value of the token, and the reasons that the token is invalid, are displayed.

Pass data along the chain

Pass profile data downstream

Retrieve user profile attributes of an AM user, and provide them in the UserProfileContext to downstream filters and handlers. Profile attributes that are enabled in AM can be retrieved, except the `roles` attribute.

The `userProfile` property of AmService is configured to retrieve `employeeNumber` and `mail`. When the property is not configured, all available attributes in `rawInfo` or `asJsonValue()` are displayed.

Retrieve profile attributes for a user authenticated with an SSO token

In this example, the user is authenticated with AM through the SingleSignOnFilter, which stores the SSO token and its validation information in the `SsoTokenContext`. The UserProfileFilter retrieves the user's mail and employee number, as well as the `username`, `_id`, and `_rev`, from that context.

1. Set up AM:
   a. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:
      - `https://ig.example.com:8443/*`
      - `https://ig.example.com:8443/*?*`

   b. Register a PingGateway agent with the following values, as described in Register a PingGateway agent in AM:
      - **Agent ID**: `ig_agent`
      - **Password**: `password`

      IMPORTANT

      Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

   c. (Optional) Authenticate the agent to AM as described in Authenticate a PingGateway agent to AM.

      IMPORTANT

      PingGateway agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

   d. Select **Services** > **Add a Service**, and add a **Validation Service** with the following **Valid goto URL Resources**:
      - `http://ig.example.com:8080/*`
      - `http://ig.example.com:8080/?`

2. Set up PingGateway:
   a. Set up PingGateway for HTTPS, as described in Configure PingGateway for TLS (server-side).

   b. Set an environment variable for the PingGateway agent password, and then restart PingGateway:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

c. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/user-profile-sso.json
```

```
{
  "name": "user-profile-sso",
  "condition": "${find(request.uri.path, '^/user-profile-
sso')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "http://am.example.com:8088/openam",
        "realm": "/",
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "amHandler": "ForgeRockClientHandler"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SingleSignOnFilter",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        },
        {
          "name": "UserProfileFilter-1",
          "type": "UserProfileFilter",
          "config": {
            "username": "${contexts.ssoToken.info.uid}",
            "userProfileService": {
              "type": "UserProfileService",
              "config": {
                "amService": "AmService-1",
                "profileAttributes": [ "employeeNumber",
"mail" ]
              }
            }
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": "<html><body>username:
${contexts.userProfile.username}<br><br>rawInfo:
<pre>${contexts.userProfile.rawInfo}</pre></body></html>"
        }
      }
    }
  }
}
```

3. Test the setup:

   a. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/user-profile-sso.

   b. Log in to AM with username `demo` and password `Ch4ng31t`.

   The UserProfileFilter retrieves the user's profile data and stores it in the UserProfileContext. The StaticResponseHandler displays the username and the profile data available in `rawInfo`:

```
username: demo

rawInfo:

{_id=demo, _rev=-1, mail=[demo@example.com],
username=demo}
```

Retrieve a username from the sessionInfo context

In this example, the UserProfileFilter retrieves AM profile information for the user identified by the SessionInfoContext, at `${contexts.amSession.username}`. The SessionInfoFilter validates an SSO token without redirecting the request to an authentication page.

1. Set up AM:

   a. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:

      - `https://ig.example.com:8443/*`
      - `https://ig.example.com:8443/*?*`

   b. Register a PingGateway agent with the following values, as described in Register a PingGateway agent in AM:

      - **Agent ID**: `ig_agent`
      - **Password**: `password`

        IMPORTANT

        Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

   c. (Optional) Authenticate the agent to AM as described in Authenticate a PingGateway agent to AM.

      IMPORTANT

      PingGateway agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

2. Set up PingGateway:

   a. Set up PingGateway for HTTPS, as described in Configure PingGateway for TLS (server-side).

b. Set an environment variable for the PingGateway agent password, and then restart PingGateway:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

c. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/user-profile-ses-info.json
```

```
{
  "name": "user-profile-ses-info",
  "condition": "${find(request.uri.path, '^/user-profile-
ses-info')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "http://am.example.com:8088/openam",
        "realm": "/",
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "amHandler": "ForgeRockClientHandler"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "capture": "all",
    "config": {
      "filters": [
        {
          "name": "SessionInfoFilter-1",
          "type": "SessionInfoFilter",
          "config": {
            "amService": "AmService-1"
          }
        },
        {
          "name": "UserProfileFilter-1",
          "type": "UserProfileFilter",
          "config": {
            "username": "${contexts.amSession.username}",
            "userProfileService": {
              "type": "UserProfileService",
              "config": {
                "amService": "AmService-1",
                "profileAttributes": [ "employeeNumber",
"mail" ]
              }
            }
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "application/json" ]
          },
          "entity": "{ \"username\":
\"${contexts.userProfile.username}\", \"user_profile\":
${contexts.userProfile.asJsonValue()} }"
        }
      }
    }
  }
}
```

3. Test the setup:

   a. In a terminal window, use a `curl` command similar to the following to retrieve an access token:

```
$ curl --request POST \
--url
```

```
http://am.example.com:8088/openam/json/realms/root/authent
icate \
--header 'accept-api-version: resource=2.0' \
--header 'content-type: application/json' \
--header 'x-openam-username: demo' \
--header 'x-openam-password: Ch4ng31t' \
--data '{}'

{"tokenId":"AQI...AA*","successUrl":"/openam/console"}
```

b. Access the route, providing the path to the certificate and token ID retrieved in the previous step:

```
$ curl -v \
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--cookie 'iPlanetDirectoryPro=tokenID'  \
https://ig.example.com:8443/user-profile-ses-info | jq .

{
  "username": "demo",
  "user_profile": {
    "_id": "demo",
    "_rev": "123...456",
    "employeeNumber": ["123"],
    "mail": ["demo@example.com"],
    "username": "demo"
  }
}
```

iPlanetDirectoryPro is the name of the AM session cookie. For more information, refer to Find the AM session cookie name.

The UserProfileFilter retrieves the user’s profile data and stores it in the UserProfileContext. The StaticResponseHandler displays the username and the profile data available in `asJsonValue()` .

Retrieve a username from the OAuth2Context

In this example, the OAuth2ResourceServerFilter validates a request containing an OAuth 2.0 access token, using the introspection endpoint, and injects the token into the OAuth2Context context. The UserProfileFilter retrieves AM profile information for the user identified by this context.

Before you start, set up and test the example in Validate access tokens with introspection.

1. Add the following route to PingGateway:

Linux Windows

```
$HOME/.openig/config/routes/user-profile-oauth.json
```

```
{
  "name": "user-profile-oauth",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/user-profile-
oauth')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "http://am.example.com:8088/openam",
        "realm": "/",
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "amHandler": "ForgeRockClientHandler"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [
              "mail",
              "employeenumber"
            ],
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name": "token-resolver-1",
              "type": "TokenIntrospectionAccessTokenResolver",
              "config": {
                "amService": "AmService-1",
                "providerHandler": {
                  "type": "Chain",
                  "config": {
                    "filters": [
                      {
                        "type":
"HttpBasicAuthenticationClientFilter",
                        "config": {
                          "username": "ig_agent",
                          "passwordSecretId":
"agent.secret.id",
                          "secretsProvider":
"SystemAndEnvSecretStore-1"
                        }
                      }
                    ],
                    "handler": "ForgeRockClientHandler"
                  }
                }
              }
            }
          }
        },
        {
          "name": "UserProfileFilter-1",
          "type": "UserProfileFilter",
          "config": {
            "username":
"${contexts.oauth2.accessToken.info.sub}",
            "userProfileService": {
              "type": "UserProfileService",
              "config": {
                "amService": "AmService-1",
                "profileAttributes": [ "employeeNumber",
"mail" ]
              }
            }
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "application/json" ]
          },
          "entity": "{ \"username\":
\"${contexts.userProfile.username}\", \"user_profile\":
${contexts.userProfile.asJsonValue()} }"
        }
      }
    }
  }
}
```

2. Test the setup:

   a. In a terminal window, use a **curl** command similar to the following to retrieve an access token:

   ```
   $ mytoken=$(curl -s \
   --user "client-application:password" \
   --data
   "grant_type=password&username=demo&password=Ch4ng31t&scope
   =mail%20employeenumber" \
   http://am.example.com:8088/openam/oauth2/access_token | jq
   -r ".access_token")
   ```

   b. Validate the access token returned in the previous step:

   ```
   $ curl -v \
   --cacert /path/to/secrets/ig.example.com-certificate.pem \
   --header "Authorization: Bearer ${mytoken}" \
   https://ig.example.com:8443/user-profile-oauth | jq .

   {
     "username": "demo",
     "user_profile": {
       "_id": "demo",
       "_rev": "123...456",
       "employeeNumber": ["123"],
       "mail": ["demo@example.com"],
       "username": "demo"
   ```

```
    }
  }
```

The UserProfileFilter retrieves the user's profile data and stores it in the UserProfileContext. The StaticResponseHandler displays the username and the profile data that is available in `asJsonValue()`.

Passing runtime data downstream

The following sections describe how to pass identity or other runtime information in a JWT, downstream to a protected application:

The examples in this section use the following objects:

- JwtBuilderFilter to collect runtime information and pack it into a JWT
- HeaderFilter to add the information to the forwarded request

To help with development, the sample application includes a `/jwt` endpoint to display the JWT, verify its signature, and decrypt the JWT.

Pass runtime data in a JWT signed with a PEM

1. Set up secrets
   a. Locate a directory for secrets, and go to it:

   ```
   $ cd /path/to/secrets
   ```

   b. Generate PEM files to sign and verify the JWT:

   ```
   $ openssl req \
   -newkey rsa:2048 \
   -new \
   -nodes \
   -x509 \
   -days 3650 \
   -subj
   "/CN=ig.example.com/OU=example/O=com/L=fr/ST=fr/C=fr" \
   -keyout id.key.for.signing.jwt.pem \
   -out id.key.for.verifying.jwt.pem
   ```

2. Set up AM:
   a. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:
      - `https://ig.example.com:8443/*`

- `https://ig.example.com:8443/*?*`

b. Register a PingGateway agent with the following values, as described in Register a PingGateway agent in AM:

- **Agent ID**: `ig_agent`
- **Password**: `password`

> IMPORTANT
>
> Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

c. (Optional) Authenticate the agent to AM as described in Authenticate a PingGateway agent to AM.

> IMPORTANT
>
> PingGateway agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

3. Set up PingGateway:

a. Set up PingGateway for HTTPS, as described in Configure PingGateway for TLS (server-side).

b. Set an environment variable for the PingGateway agent password, and then restart PingGateway:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

c. Add the following route to PingGateway to serve the sample application .css and other static resources:

Linux | Windows

```
$HOME/.openig/config/routes/00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or
matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

d. Add the following route to PingGateway, replacing value of the property `secretsDir` with the directory for the PEM file:

Linux Windows

```
$HOME/.openig/config/routes/jwt-builder-sign-pem.json
```

```
{
  "name": "jwt-builder-sign-pem",
  "condition": "${find(request.uri.path, '/jwt-builder-
sign-pem')}",
  "baseURI": "http://app.example.com:8081",
  "properties": {
    "secretsDir": "/path/to/secrets"
  },
  "capture": "all",
  "heap": [
    {
      "name": "pemPropertyFormat",
      "type": "PemPropertyFormat"
    },
    {
      "name": "FileSystemSecretStore-1",
      "type": "FileSystemSecretStore",
      "config": {
        "format": "PLAIN",
        "directory": "&{secretsDir}",
        "suffix": ".pem",
        "mappings": [{
          "secretId": "id.key.for.signing.jwt",
          "format": "pemPropertyFormat"
        }]
      }
    },
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [{
        "name": "SingleSignOnFilter",
        "type": "SingleSignOnFilter",
        "config": {
          "amService": "AmService-1"
        }
      }, {
        "name": "UserProfileFilter",
        "type": "UserProfileFilter",
        "config": {
          "username": "${contexts.ssoToken.info.uid}",
          "userProfileService": {
            "type": "UserProfileService",
            "config": {
              "amService": "AmService-1"
            }
          }
        }
      }, {
        "name": "JwtBuilderFilter-1",
        "type": "JwtBuilderFilter",
        "config": {
          "template": {
            "name": "${contexts.userProfile.commonName}",
            "email":
"${contexts.userProfile.rawInfo.mail[0]}"
          },
          "secretsProvider": "FileSystemSecretStore-1",
          "signature": {
            "secretId": "id.key.for.signing.jwt",
            "algorithm": "RS512"
          }
        }
      }, {
        "name": "HeaderFilter-1",
        "type": "HeaderFilter",
        "config": {
          "messageType": "REQUEST",
          "add": {
            "x-openig-user":
["${contexts.jwtBuilder.value}"]
          }
        }
      }],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/jwt-builder-sign-pem`.
- The agent password for AmService is provided by a SystemAndEnvSecretStore.
- If the request doesn't have a valid AM session cookie, the SingleSignOnFilter redirects the request to authenticate with AM. If the request already has a valid AM session cookie, the SingleSignOnFilter passes the request to the next filter, and stores the cookie value in an SsoTokenContext.
- The UserProfileFilter reads the username from the SsoTokenContext, uses it to retrieve the user's profile info from AM, and places the data into the UserProfileContext.
- The JwtBuilderFilter refers to the secret ID of the PEM, and uses the FileSystemSecretStore to manage the secret.
- The FileSystemSecretStore mapping refers to the secret ID of the PEM, and uses the PemPropertyFormat to define the format.
- The HeaderFilter retrieves the JWT from the JwtBuilderContext, and adds it to the header field `x-openig-user` in the request, so that the sample app can display the JWT.
- The ClientHandler passes the request to the sample app, which displays the JWT.

4. Test the setup:

a. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/jwt-builder-sign-pem.

b. Sign on to AM as user `demo`, password `Ch4ng31t`. The sample application displays the signed JWT along with its header and payload.

c. In `USE PEM FILE` in the sample app, enter the path to `id.key.for.verifying.jwt.pem` to verify the JWT signature.

Pass runtime data in a JWT signed with PEM then encrypted with a symmetric key

This example passes runtime data in a JWT that is signed with a PEM, and then encrypted with a symmetric key.

1. Set up secrets

   a. Locate a directory for secrets, and go to it:

   ```
   $ cd /path/to/secrets
   ```

   b. From the secrets directory, generate PEM files to sign and verify the JWT:

   ```
   $ openssl req \
   -newkey rsa:2048 \
   -new \
   -nodes \
   -x509 \
   -days 3650 \
   -subj
   "/CN=ig.example.com/OU=example/O=com/L=fr/ST=fr/C=fr" \
   -keyout id.key.for.signing.jwt.pem \
   -out id.key.for.verifying.jwt.pem
   ```

   c. Encrypt the PEM file used to sign the JWT:

   ```
   $ openssl pkcs8 \
   -topk8 \
   -inform PEM \
   -outform PEM \
   -in id.key.for.signing.jwt.pem \
   -out id.encrypted.key.for.signing.jwt.pem \
   -passout pass:encryptedpassword \
   -v1 PBE-SHA1-3DES
   ```

The encrypted PEM file used for signatures is `id.encrypted.key.for.signing.jwt.pem`. The password to decode the file is `encryptedpassword`.

> TIP
>
> If encryption fails, make sure your encryption methods and ciphers are supported by the Java Cryptography Extension.

d. Generate a symmetric key to encrypt the JWT:

```
$ openssl rand -base64 32 >
symmetric.key.for.encrypting.jwt
```

e. Make sure you have the following keys in your secrets directory:

- `id.encrypted.key.for.signing.jwt.pem`
- `id.key.for.signing.jwt.pem`
- `id.key.for.verifying.jwt.pem`
- `symmetric.key.for.encrypting.jwt`

2. Set up AM:

a. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:

- `https://ig.example.com:8443/*`
- `https://ig.example.com:8443/*?*`

b. Register a PingGateway agent with the following values, as described in Register a PingGateway agent in AM:

- **Agent ID**: `ig_agent`
- **Password**: `password`

> IMPORTANT
>
> Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

c. (Optional) Authenticate the agent to AM as described in Authenticate a PingGateway agent to AM.

> IMPORTANT
>
> PingGateway agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

3. Set up PingGateway:

   a. Set up PingGateway for HTTPS, as described in Configure PingGateway for TLS (server-side).

   b. Set an environment variable for the PingGateway agent password, and then restart PingGateway:

   ```
   $ export AGENT_SECRET_ID='cGFzc3dvcmQ='
   ```

   The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

   c. Add the following route to PingGateway to serve the sample application .css and other static resources:

   Linux | Windows

   ```
   $HOME/.openig/config/routes/00-static-resources.json
   ```

   ```
   {
     "name" : "00-static-resources",
     "baseURI" : "http://app.example.com:8081",
     "condition": "${find(request.uri.path,'^/css') or
   matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
   matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
     "handler": "ReverseProxyHandler"
   }
   ```

   d. In PingGateway, create an environment variable for the base64-encoded password to decrypt the PEM file used to sign the JWT:

   ```
   $ export
   ID_DECRYPTED_KEY_FOR_SIGNING_JWT='ZW5jcnlwdGVkcGFzc3dvcmQ=
   '
   ```

   e. Add the following route to PingGateway, replacing the value of `secretsDir` with your secrets directory:

   Linux | Windows

   ```
   $HOME/.openig/config/routes/jwtbuilder-sign-then-
   encrypt.json
   ```

```
{
  "name": "jwtbuilder-sign-then-encrypt",
  "condition": "${find(request.uri.path, '/jwtbuilder-
sign-then-encrypt')}",
  "baseURI": "http://app.example.com:8081",
  "properties": {
    "secretsDir": "/path/to/secrets"
  },
  "capture": "all",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore",
      "type": "SystemAndEnvSecretStore",
      "config": {
        "mappings": [{
          "secretId": "id.decrypted.key.for.signing.jwt",
          "format": "BASE64"
        }]
      }
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore",
        "url": "http://am.example.com:8088/openam"
      }
    },
    {
      "name": "pemPropertyFormat",
      "type": "PemPropertyFormat",
      "config": {
        "decryptionSecretId":
"id.decrypted.key.for.signing.jwt",
        "secretsProvider": "SystemAndEnvSecretStore"
      }
    },
    {
      "name": "FileSystemSecretStore-1",
      "type": "FileSystemSecretStore",
      "config": {
        "format": "PLAIN",
        "directory": "&{secretsDir}",
        "mappings": [{
          "secretId":
"id.encrypted.key.for.signing.jwt.pem",
          "format": "pemPropertyFormat"
        }, {
          "secretId": "symmetric.key.for.encrypting.jwt",
          "format": {
            "type": "SecretKeyPropertyFormat",
            "config": {
              "format": "BASE64",
              "algorithm": "AES"
            }
          }
        }]
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [{
        "name": "SingleSignOnFilter",
        "type": "SingleSignOnFilter",
        "config": {
          "amService": "AmService-1"
        }
      }, {
        "name": "UserProfileFilter",
        "type": "UserProfileFilter",
        "config": {
          "username": "${contexts.ssoToken.info.uid}",
          "userProfileService": {
            "type": "UserProfileService",
            "config": {
              "amService": "AmService-1"
            }
          }
        }
      }, {
        "name": "JwtBuilderFilter-1",
        "type": "JwtBuilderFilter",
        "config": {
          "template": {
            "name": "${contexts.userProfile.commonName}",
            "email":
"${contexts.userProfile.rawInfo.mail[0]}"
          },
          "secretsProvider": "FileSystemSecretStore-1",
          "signature": {
            "secretId":
"id.encrypted.key.for.signing.jwt.pem",
            "algorithm": "RS512",
            "encryption": {
              "secretId":
"symmetric.key.for.encrypting.jwt",
              "algorithm": "dir",
              "method": "A128CBC-HS256"
            }
          }
        }
      }, {
        "name": "AddBuiltJwtToHeader",
        "type": "HeaderFilter",
        "config": {
          "messageType": "REQUEST",
          "add": {
            "x-openig-user":
["${contexts.jwtBuilder.value}"]
          }
        }
      },
        {
          "name": "AddBuiltJwtAsCookie",
          "type": "HeaderFilter",
          "config": {
            "messageType": "RESPONSE",
            "add": {
              "set-cookie": ["my-
jwt=${contexts.jwtBuilder.value};PATH=/"]
            }
          }
        }],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/jwtbuilder-sign-then-encrypt`.
- The SystemAndEnvSecretStore provides the PingGateway agent password and the password to decode the PEM file for the signing keys.
- The FileSystemSecretStore maps the secret IDs of the encrypted PEM file used to sign the JWT, and the symmetric key used to encrypt the JWT.
- After authentication, the UserProfileFilter reads the username from the SsoTokenContext, uses it to retrieve the user's profile info from AM, and places the data into the UserProfileContext.
- The JwtBuilderFilter takes the username and email from the UserProfileContext, and stores them in a JWT in the JwtBuilderContext. It uses the secrets mapped in the FileSystemSecretStore to sign then encrypt the JWT.
- The `AddBuiltJwtToHeader` HeaderFilter retrieves the JWT from the JwtBuilderContext, and adds it to the header field `x-openig-user` in the request so that the sample app can display the JWT.
- The `AddBuiltJwtAsCookie` HeaderFilter adds the JWT to a cookie called `my-jwt` so that it can be retrieved by the JwtValidationFilter in JWT validation. The cookie is ignored in this example.
- The ClientHandler passes the request to the sample app.

4. Test the setup:
   a. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/jwtbuilder-sign-then-encrypt.
   b. Log in to AM as user `demo`, password `Ch4ng31t`. The sample app displays the encrypted JWT. The payload is concealed because the JWT is encrypted.
   c. In the `ENTER SECRET` box, enter the value of `symmetric.key.for.encrypting.jwt` to decrypt the JWT. The signed JWT and its payload are now displayed.
   d. In the `USE PEM FILE` box, enter the path to `id.key.for.verifying.jwt.pem` to verify the JWT signature.

Pass runtime data in JWT encrypted with a symmetric key

1. Set up secrets:
   a. Locate a directory for secrets, and go to it:

```
$ cd /path/to/secrets
```

   b. In the secrets folder, generate an AES 256-bit key:

```
$ openssl rand -base64 32

loH...UFQ=
```

c. In the secrets folder, create a file called `symmetric.key.for.encrypting.jwt` containing the AES key:

```
$ echo -n 'loH...UFQ=' > symmetric.key.for.encrypting.jwt
```

Make sure the password file contains only the password, with no trailing spaces or carriage returns.

2. Set up AM:

a. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:

- `https://ig.example.com:8443/*`
- `https://ig.example.com:8443/*?*`

b. Register a PingGateway agent with the following values, as described in Register a PingGateway agent in AM:

- **Agent ID**: `ig_agent`
- **Password**: `password`

IMPORTANT

Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

c. (Optional) Authenticate the agent to AM as described in Authenticate a PingGateway agent to AM.

IMPORTANT

PingGateway agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

3. Set up PingGateway:

a. Set up PingGateway for HTTPS, as described in Configure PingGateway for TLS (server-side).

b. Set an environment variable for the PingGateway agent password, and then restart PingGateway:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

c. Add the following route to PingGateway to serve the sample application .css and other static resources:

Linux | Windows

```
$HOME/.openig/config/routes/00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or
matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

d. Add the following route to PingGateway, replacing the value of the property `secretsDir` with your value:

Linux | Windows

```
$HOME/.openig/config/routes/jwtbuilder-encrypt-
symmetric.json
```

```
{
  "name": "jwtbuilder-encrypt-symmetric",
  "condition": "${find(request.uri.path, '/jwtbuilder-
encrypt-symmetric')}",
  "baseURI": "http://app.example.com:8081",
  "properties": {
    "secretsDir": "/path/to/secrets"
  },
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam"
      }
    },
    {
      "name": "FileSystemSecretStore-1",
      "type": "FileSystemSecretStore",
      "config": {
        "format": "PLAIN",
        "directory": "&{secretsDir}",
        "mappings": [{
          "secretId": "symmetric.key.for.encrypting.jwt",
          "format": {
            "type": "SecretKeyPropertyFormat",
            "config": {
              "format": "BASE64",
              "algorithm": "AES"
            }
          }
        }]
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [{
        "name": "SingleSignOnFilter-1",
        "type": "SingleSignOnFilter",
        "config": {
          "amService": "AmService-1"
        }
      }, {
        "name": "UserProfileFilter-1",
        "type": "UserProfileFilter",
        "config": {
          "username": "${contexts.ssoToken.info.uid}",
          "userProfileService": {
            "type": "UserProfileService",
            "config": {
              "amService": "AmService-1"
            }
          }
        }
      }, {
        "name": "JwtBuilderFilter-1",
        "type": "JwtBuilderFilter",
        "config": {
          "template": {
            "name": "${contexts.userProfile.commonName}",
            "email":
"${contexts.userProfile.rawInfo.mail[0]}"
          },
          "secretsProvider": "FileSystemSecretStore-1",
          "encryption": {
            "secretId":
"symmetric.key.for.encrypting.jwt",
            "algorithm": "dir",
            "method": "A128CBC-HS256"
          }
        }
      }, {
        "name": "HeaderFilter-1",
        "type": "HeaderFilter",
        "config": {
          "messageType": "REQUEST",
          "add": {
            "x-openig-user":
["${contexts.jwtBuilder.value}"]
          }
        }
      }],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/jwtbuilder-encrypt-symmetric`.

- The JWT encryption key is managed by the FileSystemSecretStore in the heap, which defines the SecretKeyPropertyFormat.
- The JwtBuilderFilter `encryption` property refers to key in the FileSystemSecretStore.
- The HeaderFilter retrieves the JWT from the JwtBuilderContext, and adds it to the header field `x-openig-user` in the request, so that the sample app can display the JWT.

4. Test the setup:
   a. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/jwtbuilder-encrypt-symmetric.
   b. Log in to AM as user `demo`, password `Ch4ng31t`, or as another user. The JWT is displayed in the sample app.
   c. In the `ENTER SECRET` field, enter the value of the AES 256-bit key to decrypt the JWT and display its payload.

Pass runtime data in JWT encrypted with an asymmetric key

The asymmetric key in this example is a PEM, but you can equally use a keystore.

1. Set up secrets:
   a. Locate a directory for secrets, and go to it:

```
$ cd /path/to/secrets
```

   b. Generate an encrypted PEM file:

```
$ openssl req \
-newkey rsa:2048 \
-new \
-nodes \
-x509 \
-days 3650 \
-subj
"/CN=ig.example.com/OU=example/O=com/L=fr/ST=fr/C=fr" \
-keyout id.key.for.encrypting.jwt.pem \
-out id.key.for.decrypting.jwt.pem
```

2. Set up AM:
   a. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:
      - `https://ig.example.com:8443/*`

- `https://ig.example.com:8443/*?*`

b. Register a PingGateway agent with the following values, as described in Register a PingGateway agent in AM:

- **Agent ID**: `ig_agent`
- **Password**: `password`

> IMPORTANT
>
> Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

c. (Optional) Authenticate the agent to AM as described in Authenticate a PingGateway agent to AM.

> IMPORTANT
>
> PingGateway agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

3. Set up PingGateway:

a. Set up PingGateway for HTTPS, as described in Configure PingGateway for TLS (server-side).

b. Set an environment variable for the PingGateway agent password, and then restart PingGateway:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

c. Add the following route to PingGateway to serve the sample application .css and other static resources:

Linux | Windows

```
$HOME/.openig/config/routes/00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or
matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

d. Add the following route to PingGateway, replacing value of the property `secretsDir` with the directory for the PEM file:

Linux | Windows

```
$HOME/.openig/config/routes/jwtbuilder-encrypt-
asymmetric.json
```

```
{
  "name": "jwtbuilder-encrypt-asymmetric",
  "condition": "${find(request.uri.path, '/jwtbuilder-
encrypt-asymmetric')}",
  "baseURI": "http://app.example.com:8081",
  "properties": {
    "secretsDir": "/path/to/secrets"
  },
  "capture": "all",
  "heap": [
    {
      "name": "pemPropertyFormat",
      "type": "PemPropertyFormat"
    },
    {
      "name": "FileSystemSecretStore-1",
      "type": "FileSystemSecretStore",
      "config": {
        "format": "PLAIN",
        "directory": "&{secretsDir}",
        "suffix": ".pem",
        "mappings": [{
          "secretId": "id.key.for.decrypting.jwt",
          "format": "pemPropertyFormat"
        }]
      }
    },
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [{
        "name": "SingleSignOnFilter",
        "type": "SingleSignOnFilter",
        "config": {
          "amService": "AmService-1"
        }
      }, {
        "name": "UserProfileFilter",
        "type": "UserProfileFilter",
        "config": {
          "username": "${contexts.ssoToken.info.uid}",
          "userProfileService": {
            "type": "UserProfileService",
            "config": {
              "amService": "AmService-1"
            }
          }
        }
      }, {
        "name": "JwtBuilderFilter-1",
        "type": "JwtBuilderFilter",
        "config": {
          "template": {
            "name": "${contexts.userProfile.commonName}",
            "email":
"${contexts.userProfile.rawInfo.mail[0]}"
          },
          "secretsProvider": "FileSystemSecretStore-1",
          "encryption": {
            "secretId": "id.key.for.decrypting.jwt",
            "algorithm": "RSA-OAEP-256",
            "method": "A128CBC-HS256"
          }
        }
      }, {
        "name": "HeaderFilter-1",
        "type": "HeaderFilter",
        "config": {
          "messageType": "REQUEST",
          "add": {
            "x-openig-user":
["${contexts.jwtBuilder.value}"]
          }
        }
      }],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/jwtbuilder-encrypt-asymmetric`.
- The JwtBuilderFilter refers to the secret ID of the PEM, and uses the FileSystemSecretStore to manage the secret.
- The FileSystemSecretStore mapping refers to the secret ID of the PEM, and uses the default PemPropertyFormat.
- The HeaderFilter retrieves the JWT from the JwtBuilderContext, and adds it to the header field `x-openig-user` in the request, so that the sample app can display the JWT.

4. Test the setup:

   a. In your browser’s privacy or incognito mode, go to https://ig.example.com:8443/jwtbuilder-encrypt-asymmetric.

   b. Log in to AM as user `demo`, password `Ch4ng31t`, or as another user. The JWT is displayed in the sample app.

   c. In the `USE PEM FILE` field, enter the path to `id.key.for.encrypting.jwt.pem` to decrypt the JWT and display its payload.

SAML

PingGateway implements SAML 2.0 to validate users and log them in to protected applications.

For more information about the SAML 2.0 standard, refer to RFC 7522. The following terms are used:

- *Identity Provider* (IDP): The service that manages the user identity, for example PingOne Advanced Identity Cloud or AM.
- *Service Provider* (SP): The service that a user wants to access. PingGateway acts as a SAML 2.0 SP for SSO, providing an interface to applications that don't support SAML 2.0.
- *Circle of trust* (CoT): An IDP and SP that participate in federation.
- *Fedlet*: SAML configuration files.

SAML assertions

SAML assertions can be signed and encrypted. Use SHA-256 variants (rsa-sha256 or ecdsa-sha256).

SAML assertions can contain configurable attribute values, such as user meta-information or anything else provided by the IDP. The attributes of a SAML assertion can contain one or more values, made available as a list of strings. Even if an attribute contains a single value, it is made available as a list of strings.

SAML configuration

PingGateway scans SAML configuration files once, the first time that a request accesses the SamlFederationFilter or SamlFederationHandler (deprecated) after startup. Restart PingGateway after any change to the SAML configuration files.

SAML in deployments with multiple instances of PingGateway

When PingGateway acts as a SAML service provider, session information is stored in the fedlet not the session cookie. In deployments with multiple instances of PingGateway as a SAML service provider, it is necessary to set up sticky sessions so that requests always hit the instance where the SAML interaction was started.

For information, refer to Session state considerations in AM's *SAML v2.0 guide*.

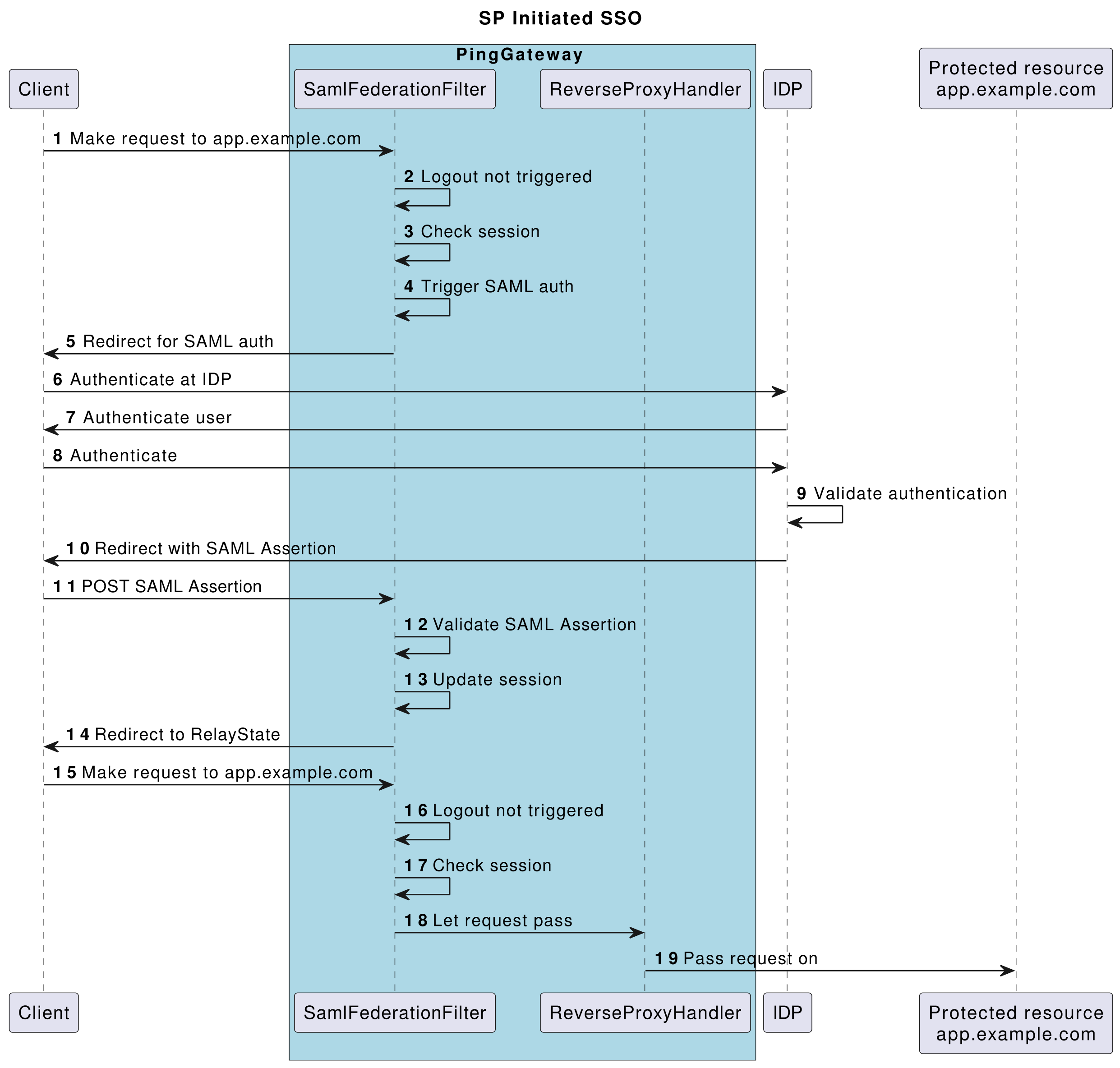
About SP-initiated SSO

SP-initiated single sign-on (SSO) occurs when a user attempts to access a protected application directly through the service provider (SP). Because the user's federated identity is managed by the identity provider (IdP), the SP sends a SAML authentication request to the IdP. After the IdP authenticates the user, it provides the SP with a SAML assertion for the user.

For the SamlFederationFilter, prefer SP-initiated SSO to IdP-initiated SSO:

- A dedicated SAML URI is not required to start SP-initiated authentication.
- The HTTP session tracks the state of the user session.

The following sequence diagram shows the flow of information in SP-initiated SSO when PingGateway acts as a SAML 2.0 SP:



AM as IdP

- Unsigned/unencrypted assertions
- Signed/encrypted assertions

PingOne as IdP

- PingOne as SAML IDP

Federation using the SamlFederationHandler (deprecated)

IMPORTANT

The SamlFederationHandler is deprecated. Use the SamlFederationFilter instead.

- About SP-initiated SSO with the SamlFederationHandler
- About IDP-initiated SSO
- Unsigned/unencrypted assertions
- Signed/encrypted assertions
- SAML 2.0 and multiple applications

Additional topics

- Non-transient NameID format
- Example fedlet files

AM as IDP

Unsigned/unencrypted assertions

This example sets up federation using AM as the identity provider with unsigned/unencrypted assertions.

1. Set up the network:

   Add `sp.example.com` to your `/etc/hosts` file:

   ```
   127.0.0.1 localhost am.example.com ig.example.com
   app.example.com sp.example.com
   ```

   Traffic to the application is proxied through PingGateway, using the host name `sp.example.com`.

2. Configure a Java Fedlet:

NOTE

NOTE

The SAML library component validates the SP's AssertionConsumerService Location against the incoming IDP SAML Assertion, based on the request information, including the port. In `sp.xml`, always specify the port in the Location value of `AssertionConsumerService`, even when using defaults of 443 or 80, as follows:

```
<AssertionConsumerService isDefault="true"
                          index="0"

Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"

Location="https://sp.example.com:443/fedletapplication" />
```

For more information about Java Fedlets, refer to Creating and configuring the Fedlet in AM's *SAML v2.0 guide*.

a. Copy and unzip the fedlet zip file, `Fedlet-7.5.0.zip`, delivered with the AM installation, into a local directory.

```
$ unzip $HOME/openam/Fedlet-7.5.0.zip

Archive:  Fedlet-7.5.0.zip
creating: conf/
inflating: README
inflating: conf/FederationConfig.properties
inflating: conf/fedlet.cot-template
inflating: conf/idp-extended.xml-template
inflating: conf/sp-extended.xml-template
inflating: conf/sp.xml-template
inflating: fedlet.war
```

b. In each file, search and replace the following properties:

Replace this	With this
`IDP_ENTITY_ID`	`openam`
`FEDLET_ENTITY_ID`	`sp`
`FEDLET_PROTOCOL://FEDLET_HOST:FEDLET_PORT/FEDLET_DEPLOY_URI`	`https://sp.example.com:8443/home/saml`
`fedletcot` and `FEDLET_COT`	`Circle of Trust`

Replace this	With this
`sp.example.com:8443/home/saml/fedletapplication`	`sp.example.com:8443/home/saml/fedletapplication/metaAlias/sp`

c. Save the files as .xml, without the `-template` extension, so that the directory looks like this:

```
conf
├── FederationConfig.properties
├── fedlet.cot
├── idp-extended.xml
├── sp-extended.xml
└── sp.xml
```

By default, AM as an IDP uses the NameID format `urn:oasis:names:tc:SAML:2.0:nameid-format:transient` to communicate about a user. For information about using a different NameID format, refer to Non-transient NameID format.

3. Set up AM:

   a. In the AM admin UI, select **Identities**, select the user `demo` , and change the last name to `Ch4ng31t` . Note that, for this example, the last name must be the same as the password.

   b. Select **Applications** > **Federation** > **Circles of Trust**, and add a circle of trust called `Circle of Trust` , with the default settings.

   c. Set up a remote service provider:

      i. Select **Applications** > **Federation** > **Entity Providers**, and add a remote entity provider.

      ii. Drag in or import `sp.xml` created in the previous step.

      iii. Select **Circles of Trust**: `Circle of Trust` .

   d. Set up a hosted identity provider:

      i. Select **Applications** > **Federation** > **Entity Providers**, and add a hosted entity provider with the following values:

         - **Entity ID**: `openam`
         - **Entity Provider Base URL**: `http://am.example.com:8088/openam`
         - **Identity Provider Meta Alias**: `idp`
         - **Circles of Trust**: `Circle of Trust`

      ii. Select **Assertion Processing** > **Attribute Mapper**, map the following SAML attribute keys and values, and then save your changes:

- **SAML Attribute**: `cn` , **Local Attribute**: `cn`
- **SAML Attribute**: `sn` , **Local Attribute**: `sn`

iii. In a terminal, export the XML-based metadata for the IDP:

```
$ curl -v \
--output idp.xml \
"http://am.example.com:8088/openam/saml2/jsp/exportmeta
data.jsp?entityid=openam"
```

The `idp.xml` file is created locally.

4. Set up PingGateway:

   a. Set up PingGateway for HTTPS, as described in Configure PingGateway for TLS (server-side).

   b. Copy the edited fedlet files, and the exported `idp.xml` file into the PingGateway configuration, at `$HOME/.openig/SAML` .

```
$ ls -l $HOME/.openig/SAML

FederationConfig.properties
fedlet.cot
idp-extended.xml
idp.xml
sp-extended.xml
sp.xml
```

   c. Add the following route to PingGateway to serve the sample application .css and other static resources:

Linux | Windows

```
$HOME/.openig/config/routes/00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or
matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

d. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/saml-filter.json
```

```
{
  "name": "saml-filter",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home')}",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SamlFilter",
          "type": "SamlFederationFilter",
          "config": {
            "assertionMapping": {
              "name": "cn",
              "surname": "sn"
            },
            "subjectMapping": "sp-subject-name",
            "redirectURI": "/home/saml-filter"
          }
        },
        {
          "name": "SetSamlHeaders",
          "type": "HeaderFilter",
          "config": {
            "messageType": "REQUEST",
            "add": {
              "x-saml-cn": [ "${toString(session.name)}"
],
              "x-saml-sn": [
"${toString(session.surname)}" ]
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/home`.
- The SamlFederationFilter extracts `cn` and `sn` from the SAML assertion, and maps them to the SessionContext, at `session.name[0]` and `session.surname[0]`.
- The HeaderFilter adds the session name and surname as headers to the request so that they are displayed by the sample application.

e. Restart PingGateway.

5. Test the setup:

a. In your browser's privacy or incognito mode, go to https://sp.example.com:8443/home.

b. Log in to AM as user `demo`, password `Ch4ng31t`. The request is redirected to the sample application.

TIP

If a request returns an HTTP 414 URI Too Long error, consider the information in URI Too Long error.

Signed/encrypted assertions

This example set up federation using AM as the identity provider with signed/encrypted assertions.

Before you start, set up and test the example in Unsigned/unencrypted assertions.

1. Set up the SAML keystore:

a. Find the values of AM's default SAML keypass and storepass:

```
$ more /path/to/am/secrets/default/.keypass
$ more /path/to/am/secrets/default/.storepass
```

b. Copy the SAML keystore from the AM configuration to PingGateway:

```
$ cp /path/to/am/secrets/keystores/keystore.jceks
/path/to/ig/secrets/keystore.jceks
```

WARNING

WARNING

Legacy keystore types such as JKS and JCEKS are supported but aren’t secure. Consider using the PKCS#12 keystore type.

2. Configure the Fedlet in PingGateway:
   a. In `FederationConfig.properties`, make the following changes:
      i. Delete the following lines:
         - `com.sun.identity.saml.xmlsig.keystore=%BASE_DIR%/security/keystores/keystore.jks`
         - `com.sun.identity.saml.xmlsig.storepass=%BASE_DIR%/.storepass`
         - `com.sun.identity.saml.xmlsig.keypass=%BASE_DIR%/.keypass`
         - `com.sun.identity.saml.xmlsig.certalias=test`
         - `com.sun.identity.saml.xmlsig.storetype=JKS`
         - `am.encryption.pwd=@AM_ENC_PWD@`

      ii. Add the following line:

      ```
      org.forgerock.openam.saml2.credential.resolver.class=org.forgerock.openig.handler.saml.SecretsSaml2CredentialResolver
      ```

      This class is responsible for resolving secrets and supplying credentials.

      TIP

      Be sure to leave no space at the end of the line.

   b. In `sp.xml`, make the following changes:
      i. Change `AuthnRequestsSigned="false"` to `AuthnRequestsSigned="true"`.
      ii. Add the following KeyDescriptor just before `</SPSSODescriptor>`

      ```
              <KeyDescriptor use="signing">
                  <ds:KeyInfo
      xmlns:ds="http://www.w3.org/2000/09/xmldsig#" >
                      <ds:X509Data>
                          <ds:X509Certificate>

                          </ds:X509Certificate>
                      </ds:X509Data>
                  </ds:KeyInfo>
              </KeyDescriptor>
          </SPSSODescriptor>
      ```

iii. Copy the value of the signing certificate from `idp.xml` to this file:

```
<KeyDescriptor use="signing">
  <ds:KeyInfo>
    <ds:X509Data>
      <ds:X509Certificate>

        MII...zA6

      </ds:X509Certificate>
```

This is the public key used for signing so that the IDP can verify request signatures.

3. Replace the remote service provider in AM:

   a. Select **Applications** > **Federation** > **Entity Providers**, and remove the `sp` entity provider.

   b. Drag in or import the new `sp.xml` updated in the previous step.

   c. Select **Circles of Trust**: `Circle of Trust`.

4. Set up PingGateway

   a. In the PingGateway configuration, set environment variables for the following secrets, and then restart PingGateway:

```
$ export KEYSTORE_SECRET_ID='a2V5c3RvcmU='
$ export SAML_KEYSTORE_STOREPASS_SECRET_ID='base64-encoded
value of the SAML storepass'
$ export SAML_KEYSTORE_KEYPASS_SECRET_ID='base64-encoded
value of the SAML keypass'
```

   The passwords are retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

   b. Remove `saml-filter.json` from the configuration, and add the following route, replacing the path to `keystore.jceks` with your path:

Linux Windows

```
$HOME/.openig/config/routes/saml-filter-secure.json
```

```
{
  "name": "saml-filter-secure",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "KeyStoreSecretStore-1",
      "type" : "KeyStoreSecretStore",
      "config" : {
        "file" : "/path/to/ig/keystore.jceks",
        "storeType" : "jceks",
        "storePasswordSecretId" :
"saml.keystore.storepass.secret.id",
        "entryPasswordSecretId" :
"saml.keystore.keypass.secret.id",
        "secretsProvider" : "SystemAndEnvSecretStore-1",
        "mappings" : [ {
          "secretId" : "sp.signing.sp",
          "aliases" : [ "rsajwtsigningkey" ]
        }, {
          "secretId" : "sp.decryption.sp",
          "aliases" : [ "test" ]
        } ]
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SamlFilter",
          "type": "SamlFederationFilter",
          "config": {
            "assertionMapping": {
              "name": "cn",
              "surname": "sn"
            },
            "subjectMapping": "sp-subject-name",
            "redirectURI": "/home/saml-filter",
            "secretsProvider" : "KeyStoreSecretStore-1"
          }
        },
        {
          "name": "SetSamlHeaders",
          "type": "HeaderFilter",
          "config": {
            "messageType": "REQUEST",
            "add": {
              "x-saml-cn": [  "${toString(session.name)}"
],
              "x-saml-sn": [
"${toString(session.surname)}" ]
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route compared to `saml-filter.json`:

- The SamlFederationHandler refers to the KeyStoreSecretStore to provide the keys for the signed and encrypted SAML assertions.
- The secret IDs, `sp.signing.sp` and `sp.decryption.sp`, follow a naming convention based on the name of the service provider, `sp`.
- The alias for the signing key corresponds to the PEM in `keystore.jceks`.

c. Restart PingGateway.

5. Test the setup:

   a. In your browser's privacy or incognito mode, go to https://sp.example.com:8443/home.

   b. Log in to AM as user `demo`, password `Ch4ng31t`. The request is redirected to the sample application.

TIP

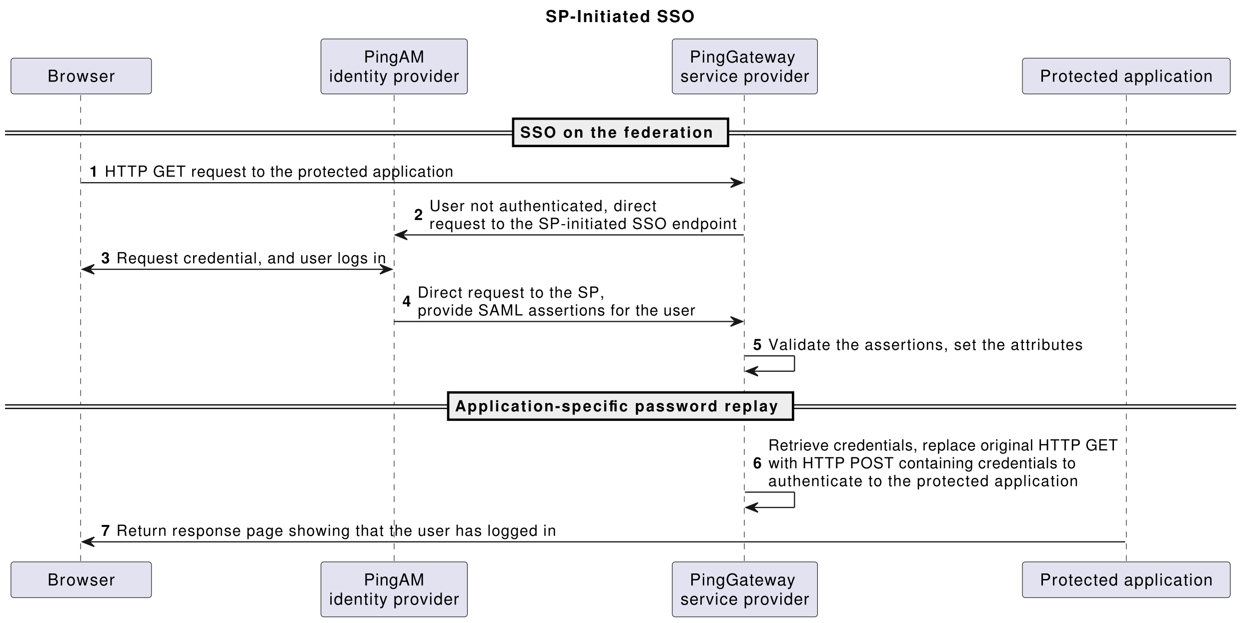
If a request returns an HTTP 414 URI Too Long error, consider the information in URI Too Long error.

SamlFederationHandler (deprecated)

About SP-initiated SSO with the SamlFederationHandler

SP-initiated SSO occurs when a user attempts to access a protected application directly through the SP. Because the user's federated identity is managed by the IDP, the SP sends a SAML authentication request to the IDP. After the IDP authenticates the user, it provides the SP with a SAML assertion for the user.

The following sequence diagram shows the flow of information in SP-initiated SSO, when PingGateway acts as a SAML 2.0 SP:


SP-Initiated SSO
Browser
PingAM identity provider
PingGateway service provider
Protected application
SSO on the federation
1 HTTP GET request to the protected application
2 User not authenticated, direct request to the SP-initiated SSO endpoint
3 Request credential, and user logs in
4 Direct request to the SP, provide SAML assertions for the user
5 Validate the assertions, set the attributes
Application-specific password replay
6 Retrieve credentials, replace original HTTP GET with HTTP POST containing credentials to authenticate to the protected application
7 Return response page showing that the user has logged in

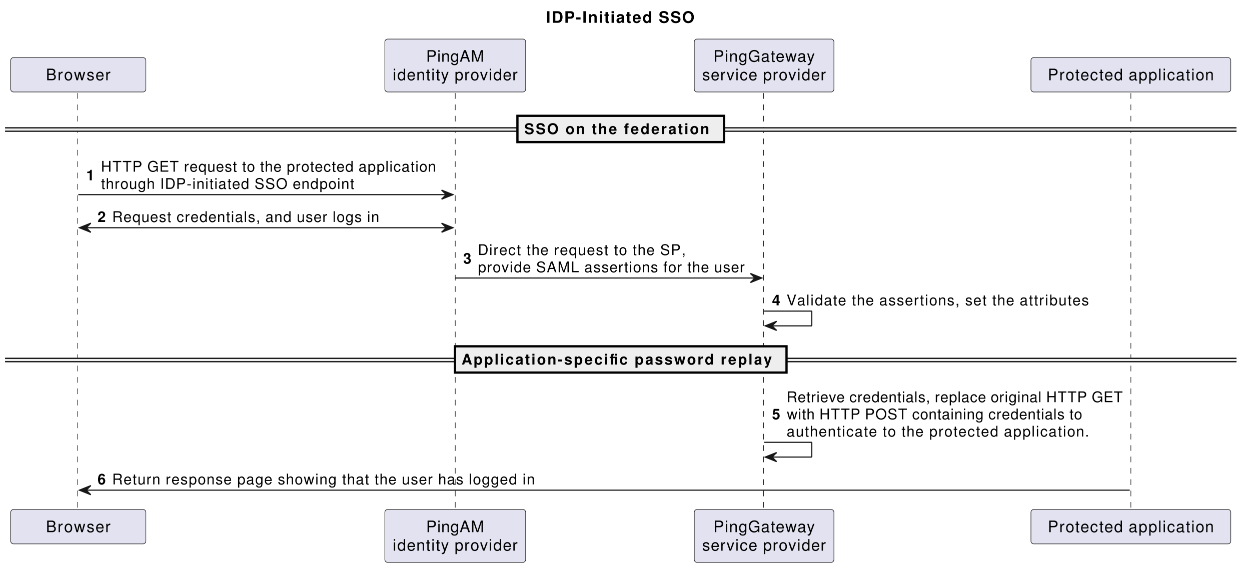
About IDP-initiated SSO

IDP-initiated SSO occurs when a user attempts to access a protected application, using the IDP for authentication. The IDP sends an unsolicited authentication statement to the SP.

Before IDP-initiated SSO can occur:

- The user must access a link on the IDP that refers to the remote SP.
- The user must authenticate to the IDP.
- The IDP must be configured with links that refer to the SP.

The following sequence diagram shows the flow of information in IDP-initiated SSO when PingGateway acts as a SAML 2.0 SP:


IDP-Initiated SSO
Browser
PingAM identity provider
PingGateway service provider
Protected application
SSO on the federation
1 HTTP GET request to the protected application through IDP-initiated SSO endpoint
2 Request credentials, and user logs in
3 Direct the request to the SP, provide SAML assertions for the user
4 Validate the assertions, set the attributes
Application-specific password replay
5 Retrieve credentials, replace original HTTP GET with HTTP POST containing credentials to authenticate to the protected application.
6 Return response page showing that the user has logged in
Browser
PingAM identity provider
PingGateway service provider
Protected application

Unsigned/unencrypted assertions

For examples of the federation configuration files, refer to Example fedlet files. To set up multiple SPs, work through this page and SAML 2.0 and multiple applications.

1. Set up the network:

   Add `sp.example.com` to your `/etc/hosts` file:

   ```
   127.0.0.1 localhost am.example.com ig.example.com
   app.example.com sp.example.com
   ```

   Traffic to the application is proxied through PingGateway, using the host name `sp.example.com`.

2. Configure a Java Fedlet:

   NOTE

   The SAML library component validates the SP's AssertionConsumerService Location against the incoming IDP SAML Assertion, based on the request information, including the port. In `sp.xml`, always specify the port in the Location value of `AssertionConsumerService`, even when using defaults of 443 or 80, as follows:

   ```
   <AssertionConsumerService isDefault="true"
                             index="0"

   Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"

   Location="https://sp.example.com:443/fedletapplication" />
   ```

For more information about Java Fedlets, refer to Creating and configuring the Fedlet in AM’s *SAML v2.0 guide*.

a. Copy and unzip the fedlet zip file, `Fedlet-7.5.0.zip`, delivered with the AM installation, into a local directory.

```
$ unzip $HOME/openam/Fedlet-7.5.0.zip

Archive:  Fedlet-7.5.0.zip
creating: conf/
inflating: README
inflating: conf/FederationConfig.properties
inflating: conf/fedlet.cot-template
inflating: conf/idp-extended.xml-template
inflating: conf/sp-extended.xml-template
inflating: conf/sp.xml-template
inflating: fedlet.war
```

b. In each file, search and replace the following properties:

Replace this	With this
`IDP_ENTITY_ID`	`openam`
`FEDLET_ENTITY_ID`	`sp`
`FEDLET_PROTOCOL://FEDLET_HOST:FEDLET_PORT/FEDLET_DEPLOY_URI`	`http://sp.example.com:8080/saml`
`fedletcot and FEDLET_COT`	`Circle of Trust`
`sp.example.com:8080/saml/fedletapplication`	`sp.example.com:8080/saml/fedletapplication/metaAlias/sp`

c. Save the files as .xml, without the `-template` extension, so that the directory looks like this:

```
conf
├── FederationConfig.properties
├── fedlet.cot
├── idp-extended.xml
├── sp-extended.xml
└── sp.xml
```

By default, AM as an IDP uses the NameID format `urn:oasis:names:tc:SAML:2.0:nameid-format:transient` to communicate about a user. For information about using a different NameID format, refer to Non-transient NameID format.

3. Set up AM:

   a. In the AM admin UI, select **Identities**, select the user `demo`, and change the last name to `Ch4ng31t`. Note that, for this example, the last name must be the same as the password.

   b. Select **Applications** > **Federation** > **Circles of Trust**, and add a circle of trust called `Circle of Trust`, with the default settings.

   c. Set up a remote service provider:

      i. Select **Applications** > **Federation** > **Entity Providers**, and add a remote entity provider.

      ii. Drag in or import `sp.xml` created in the previous step.

      iii. Select **Circles of Trust**: `Circle of Trust`.

   d. Set up a hosted identity provider:

      i. Select **Applications** > **Federation** > **Entity Providers**, and add a hosted entity provider with the following values:

         - **Entity ID**: `openam`
         - **Entity Provider Base URL**: `http://am.example.com:8088/openam`
         - **Identity Provider Meta Alias**: `idp`
         - **Circles of Trust**: `Circle of Trust`

      ii. Select **Assertion Processing** > **Attribute Mapper**, map the following SAML attribute keys and values, and then save your changes:

         - **SAML Attribute**: `cn`, **Local Attribute**: `cn`
         - **SAML Attribute**: `sn`, **Local Attribute**: `sn`

      iii. In a terminal, export the XML-based metadata for the IDP:

```
$ curl -v \
--output idp.xml \
"http://am.example.com:8088/openam/saml2/jsp/exportmetadata.jsp?entityid=openam"
```

      The `idp.xml` file is created locally.

4. Set up PingGateway:

   a. Copy the edited fedlet files, and the exported `idp.xml` file into the PingGateway configuration, at `$HOME/.openig/SAML`.

```
$ ls -l $HOME/.openig/SAML

FederationConfig.properties
fedlet.cot
idp-extended.xml
idp.xml
sp-extended.xml
sp.xml
```

b. In `config.json`, comment out or remove the `baseURI`:

```
{
  "handler": {
    "_baseURI": "http://app.example.com:8081",
    ...
  }
}
```

Requests to the SamlFederationHandler must not be rebased, because the request URI must match the endpoint in the SAML metadata.

c. Add the following route to PingGateway to serve the sample application .css and other static resources:

Linux Windows

```
$HOME/.openig/config/routes/00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or
matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

d. Add the following route to PingGateway:

Linux Windows

```
$HOME/.openig/config/routes/saml-handler.json
```

```
{
  "name": "saml-handler",
  "condition": "${find(request.uri.path, '^/saml')}",
  "handler": {
    "type": "SamlFederationHandler",
    "config": {
      "useOriginalUri": true,
      "assertionMapping": {
        "username": "cn",
        "password": "sn"
      },
      "subjectMapping": "sp-subject-name",
      "redirectURI": "/home/federate"
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/saml`.
- After authentication, the SamlFederationHandler extracts `cn` and `sn` from the SAML assertion, and maps them to the SessionContext, at `session.username` and `session.password`.
- The handler stores the subject name as a string in the session field `session.sp-subject-name`, which is named by the `subjectMapping` property. By default, the subject name is stored in the session field `session.subjectName`.
- The handler redirects the request to the `/federate` route.

e. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/federate-handler.json
```

```
{
  "name": "federate-handler",
  "condition": "${find(request.uri.path,
'^/home/federate')}",
  "baseURI": "http://app.example.com:8081",
  "handler": {
    "type": "DispatchHandler",
    "config": {
      "bindings": [
        {
          "condition": "${empty session.username}",
          "handler": {
            "type": "StaticResponseHandler",
            "config": {
              "status": 302,
              "headers": {
                "Location": [

"http://sp.example.com:8080/saml/SPInitiatedSSO?
metaAlias=/sp"
                ]
              }
            }
          }
        },
        {
          "handler": {
            "type": "Chain",
            "config": {
              "filters": [
                {
                  "type": "HeaderFilter",
                  "config": {
                    "messageType": "REQUEST",
                    "add": {
                      "x-username":
["${session.username[0]}"],
                      "x-password":
["${session.password[0]}"]
                    }
                  }
                }
              ],
              "handler": "ReverseProxyHandler"
            }
          }
        }
      ]
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/home/federate`.
- If the user is not authenticated with AM, the username is not populated in the context. The DispatchHandler then dispatches the request to the StaticResponseHandler, which redirects it to the SP-initiated SSO endpoint.

  If the credentials are in the context, or after successful authentication, the DispatchHandler dispatches the request to the Chain.
- The HeaderFilter adds headers for the first value for the `username` and `password` attributes of the SAML assertion.

f. Restart PingGateway.

5. Test the setup:

   a. Log out of AM, and test the setup with the following links:

   - IDP-initiated SSO
   - SP-initiated SSO

   b. Log in to AM with username `demo` and password `Ch4ng31t`.

   PingGateway returns the response page showing that the the demo user has logged in.

TIP

For more control over the URL where the user agent is redirected, use the `RelayState` query string parameter in the URL of the redirect `Location` header. `RelayState` specifies where to redirect the user when the SAML 2.0 web browser SSO process is complete. It overrides the `redirectURI` set in the SamlFederationHandler.

The `RelayState` value must be URL-encoded. When using an expression, use a function to encode the value. For example, use `${urlEncodeQueryParameterNameOrValue(contexts.router.originalUri)}`.

In the following example, the user is finally redirected to the original URI from the request:

```
"headers": {
  "Location": [
    "http://ig.example.com:8080/saml/SPInitiatedSSO?
RelayState=${urlEncodeQueryParameterNameOrValue(contexts.router.originalUr
i)}"
  ]
}
```

Signed/encrypted assertions

1. Set up the example in Unsigned/unencrypted assertions.
2. Set up the SAML keystore:
   a. Find the values of AM's default SAML keypass and storepass:

   ```
   $ more /path/to/am/secrets/default/.keypass
   $ more /path/to/am/secrets/default/.storepass
   ```

   b. Copy the SAML keystore from the AM configuration to PingGateway:

   ```
   $ cp /path/to/am/secrets/keystores/keystore.jceks
   /path/to/ig/secrets/keystore.jceks
   ```

   WARNING

   Legacy keystore types such as JKS and JCEKS are supported but aren't secure. Consider using the PKCS#12 keystore type.
3. Configure the Fedlet in PingGateway:
   a. In `FederationConfig.properties` , make the following changes:
      i. Delete the following lines:
         - `com.sun.identity.saml.xmlsig.keystore=%BASE_DIR%/security/keystores/keystore.jks`
         - `com.sun.identity.saml.xmlsig.storepass=%BASE_DIR%/.storepass`
         - `com.sun.identity.saml.xmlsig.keypass=%BASE_DIR%/.keypass`
         - `com.sun.identity.saml.xmlsig.certalias=test`
         - `com.sun.identity.saml.xmlsig.storetype=JKS`
         - `am.encryption.pwd=@AM_ENC_PWD@`

      ii. Add the following line:

      ```
      org.forgerock.openam.saml2.credential.resolver.class=org.forgerock.openig.handler.saml.SecretsSaml2CredentialResolver
      ```

      This class is responsible for resolving secrets and supplying credentials.

      TIP

      Be sure to leave no space at the end of the line.
   b. In `sp.xml` , make the following changes:
      i. Change `AuthnRequestsSigned="false"` to `AuthnRequestsSigned="true"` .

ii. Add the following KeyDescriptor just before `</SPSSODescriptor>`

```
        <KeyDescriptor use="signing">
            <ds:KeyInfo
xmlns:ds="http://www.w3.org/2000/09/xmldsig#" >
                <ds:X509Data>
                    <ds:X509Certificate>

                    </ds:X509Certificate>
                </ds:X509Data>
            </ds:KeyInfo>
        </KeyDescriptor>
    </SPSSODescriptor>
```

iii. Copy the value of the signing certificate from `idp.xml` to this file:

```
<KeyDescriptor use="signing">
  <ds:KeyInfo>
    <ds:X509Data>
      <ds:X509Certificate>

        MII...zA6

      </ds:X509Certificate>
```

This is the public key used for signing so that the IDP can verify request signatures.

4. Replace the remote service provider in AM:

   a. Select **Applications** > **Federation** > **Entity Providers**, and remove the `sp` entity provider.

   b. Drag in or import the new `sp.xml` updated in the previous step.

   c. Select **Circles of Trust**: `Circle of Trust`.

5. Set up PingGateway:

   a. In the PingGateway configuration, set environment variables for the following secrets, and then restart PingGateway:

```
$ export KEYSTORE_SECRET_ID='a2V5c3RvcmU='
$ export SAML_KEYSTORE_STOREPASS_SECRET_ID='base64-encoded
value of the SAML storepass'
$ export SAML_KEYSTORE_KEYPASS_SECRET_ID='base64-encoded
value of the SAML keypass'
```

The passwords are retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

b. Remove `saml-handler.json` from the configuration, and add the following route, replacing the path to `keystore.jceks` with your path:

Linux | Windows

```
$HOME/.openig/config/routes/saml-handler-secure.json
```

```
{
  "name": "saml-handler-secure",
  "condition": "${find(request.uri.path, '^/saml')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "KeyStoreSecretStore-1",
      "type" : "KeyStoreSecretStore",
      "config" : {
        "file" : "/path/to/ig/keystore.jceks",
        "storeType" : "jceks",
        "storePasswordSecretId" :
"saml.keystore.storepass.secret.id",
        "entryPasswordSecretId" :
"saml.keystore.keypass.secret.id",
        "secretsProvider" : "SystemAndEnvSecretStore-1",
        "mappings" : [ {
          "secretId" : "sp.signing.sp",
          "aliases" : [ "rsajwtsigningkey" ]
        }, {
          "secretId" : "sp.decryption.sp",
          "aliases" : [ "test" ]
        } ]
      }
    }
  ],
  "handler": {
    "type": "SamlFederationHandler",
    "config": {
      "useOriginalUri": true,
      "assertionMapping": {
        "username": "cn",
        "password": "sn"
      },
      "subjectMapping": "sp-subject-name",
      "redirectURI": "/home/federate",
      "secretsProvider" : "KeyStoreSecretStore-1"
    }
  }
}
```

Notice the following features of the route compared to `saml-handler.json`:

- The SamlFederationHandler refers to the KeyStoreSecretStore to provide the keys for the signed and encrypted SAML assertions.
- The secret IDs, `sp.signing.sp` and `sp.decryption.sp`, follow a naming convention based on the name of the service provider, `sp`.
- The alias for the signing key corresponds to the PEM in `keystore.jceks`.

c. Restart PingGateway.

6. Test the setup:

   a. Log out of AM, and test the setup with the following links:

   - IDP-initiated SSO
   - SP-initiated SSO

   b. Log in to AM with username `demo` and password `Ch4ng31t`.

   PingGateway returns the response page showing that the the demo user has logged in.

SAML 2.0 and multiple applications

This page extends the previous example to add a second service provider.

The new service provider has entity ID `sp2` and runs on the host `sp2.example.com`. To prevent unwanted behavior, the service providers must have different values.

1. Add `sp2.example.com` to your `/etc/hosts` file:

```
127.0.0.1 localhost am.example.com ig.example.com
app.example.com sp.example.com sp2.example.com
```

2. In PingGateway, configure the service provider files for `sp2`, using the files you created to configure the Fedlet.

   a. In `fedlet.cot`, add `sp2` to the list of sun-fm-trusted-providers:

```
cot-name=Circle of Trust
sun-fm-cot-status=Active
sun-fm-trusted-providers=openam, sp, sp2
sun-fm-saml2-readerservice-url=
sun-fm-saml2-writerservice-url=
```

b. Copy `sp.xml` to `sp2.xml`, and copy `sp-extended.xml` to `sp2-extended.xml`.

c. In both files, search and replace the following strings:

- `entityID=sp`: replace with `entityID=sp2`
- `sp.example.com`: replace with `sp2.example.com`
- `metaAlias=/sp`: replace with `metaAlias=/sp2`
- `/metaAlias/sp`: replace with `/metaAlias/sp2`

d. Restart PingGateway.

3. In AM, set up a remote service provider for `sp2`:

a. Select **Applications** > **Federation** > **Entity Providers**.

b. Drag in or import `sp2.xml` created in the previous step.

c. Select **Circles of Trust**: `Circle of Trust`.

4. Add the following routes to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/saml-handler-sp2.json
```

```
{
  "name": "saml-handler-sp2",
  "condition": "${find(request.uri.host, 'sp2.example.com')
and find(request.uri.path, '^/saml')}",
  "handler": {
    "type": "SamlFederationHandler",
    "config": {
      "comment": "Use unique session properties for this SP.",
      "useOriginalUri": true,
      "assertionMapping": {
        "sp2Username": "cn",
        "sp2Password": "sn"
      },
      "authnContext": "sp2AuthnContext",
      "sessionIndexMapping": "sp2SessionIndex",
      "subjectMapping": "sp2SubjectName",
      "redirectURI": "/sp2"
    }
  }
}
```

Linux Windows

```
$HOME/.openig/config/routes/federate-handler-sp2.json
```

```
{
  "name": "federate-handler-sp2",
  "condition": "${find(request.uri.host, 'sp2.example.com')
and not find(request.uri.path, '^/saml')}",
  "baseURI": "http://app.example.com:8081",
  "handler": {
    "type": "DispatchHandler",
    "config": {
      "bindings": [
        {
          "condition": "${empty session.sp2Username}",
          "handler": {
            "type": "StaticResponseHandler",
            "config": {
              "status": 302,
              "headers": {
                "Location": [

"http://sp2.example.com:8080/saml/SPInitiatedSSO?
metaAlias=/sp2"
                ]
              }
            }
          }
        },
        {
          "handler": {
            "type": "Chain",
            "config": {
              "filters": [
                {
                  "type": "HeaderFilter",
                  "config": {
                    "messageType": "REQUEST",
                    "add": {
                      "x-username":
["${session.sp2Username[0]}"],
                      "x-password":
["${session.sp2Password[0]}"]
                    }
                  }
                }
              ],
              "handler": "ReverseProxyHandler"
            }
          }
        }
      ]
    }
  }
}
```

5. Test the setup:
   a. Log out of AM, and test the setup with the following links:
      - IDP-initiated SSO
      - SP-initiated SSO
   b. Log in to AM with username `demo` and password `Ch4ng31t`.

      PingGateway returns the response page showing that the user has logged in.

Additional topics

Non-transient NameID format

By default, AM as an IdP uses the NameID format `urn:oasis:names:tc:SAML:2.0:nameid-format:transient`. Learn more in the AM documentation on Hosted identity provider configuration properties.

When the IdP uses another NameID format, configure PingGateway to use that NameID format by editing the Fedlet configuration file `sp-extended.xml`:

- To use the NameID value provided by the IdP, add the following attribute:

```
<Attribute name="useNameIDAsSPUserID">
  <Value>true</Value>
```

```
</Attribute>
```

- To use an attribute from the assertion, add the following attribute:

  ```
  <Attribute name="autofedEnabled">
    <Value>true</Value>
  </Attribute>
  <Attribute name="autofedAttribute">
    <Value>sn</Value>
  </Attribute>
  ```

  This example uses the value in `SN` to identify the subject.

Although PingGateway supports the `persistent` NameID format, PingGateway doesn't store the mapping. To configure this behavior, edit the file `sp-extended.xml`:

- To disable attempts to persist the user mapping, add the following attribute:

  ```
  <Attribute name="spDoNotWriteFederationInfo">
    <Value>true</Value>
  </Attribute>
  ```

- To enable attempts to persist the user mapping, add the following attribute:

  ```
  <Attribute name="spDoNotWriteFederationInfo">
    <Value>false</Value>
  </Attribute>
  ```

If a sign on request doesn't contain a NameID format query parameter, the value is defined by the presence and content of the NameID format list for the SP and IdP. For example, an SP-initiated login can be constructed with the binding and `NameIDFormat` as a parameter, as follows:

```
http://fedlet.example.org:7070/fedlet/SPInitiatedSSO?
binding=urn:oasis:names:tc:SAML:2.0:bindings:HTTP-
POST&NameIDFormat=urn:oasis:names:tc:SAML:1.1:nameid-
format:unspecified
```

When the NameID format is provided in a list, it is resolved as follows:

- If both the IdP and SP have a list, the first matching NameID format in the lists.
- If either the IdP or SP list is empty, the first NameID format in the other list.
- If neither the IdP nor SP has a list, AM defaults to `transient` and PingGateway defaults to `persistent`.

Example fedlet files

File	Description
`FederationConfig.properties`	Fedlet properties
`fedlet.cot`	Circle of trust for PingGateway and the IDP
`idp.xml`	Standard metadata for the IDP
`idp-extended.xml`	Metadata extensions for the IDP
`sp.xml`	Standard metadata for the PingGateway SP
`sp-extended.xml`	Metadata extensions for the PingGateway SP

AM as IDP

▼ FederationConfig.properties

The following example of `$HOME/.openig/SAML/FederationConfig.properties` defines the fedlet properties:

```
#
# DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS HEADER.
#
# Copyright (c) 2006 Sun Microsystems Inc. All Rights Reserved
#
# The contents of this file are subject to the terms
# of the Common Development and Distribution License
# (the License). You may not use this file except in
# compliance with the License.
#
# You can obtain a copy of the License at
# https://opensso.dev.java.net/public/CDDLv1.0.html or
# opensso/legal/CDDLv1.0.txt
# See the License for the specific language governing
# permission and limitations under the License.
#
# When distributing Covered Code, include this CDDL
# Header Notice in each file and include the License file
# at opensso/legal/CDDLv1.0.txt.
# If applicable, add the following below the CDDL Header,
# with the fields enclosed by brackets [] replaced by
# your own identifying information:
# "Portions Copyrighted [year] [name of copyright owner]"
#
# $Id: FederationConfig.properties,v 1.21 2010/01/08 22:41:28
exu Exp $
#
# Portions Copyright 2016-2024 Ping Identity Corporation.

# If a component wants to use a different datastore provider
than the
# default one defined above, it can define a property like
follows:
# com.sun.identity.plugin.datastore.class.<componentName>=
<provider class>

# com.sun.identity.plugin.configuration.class specifies
implementation for
# com.sun.identity.plugin.configuration.ConfigurationInstance
interface.
com.sun.identity.plugin.configuration.class=com.sun.identity.plu
gin.configuration.impl.FedletConfigurationImpl

# Specifies implementation for
# com.sun.identity.plugin.datastore.DataStoreProvider interface.
# This property defines the default datastore provider.
com.sun.identity.plugin.datastore.class.default=com.sun.identity
.plugin.datastore.impl.FedletDataStoreProvider

# Specifies implementation for
# org.forgerock.openam.federation.plugin.rooturl.RootUrlProvider
interface.
# This property defines the default base url provider.
com.sun.identity.plugin.root.url.class.default=org.forgerock.ope
nam.federation.plugin.rooturl.impl.FedletRootUrlProvider

# com.sun.identity.plugin.log.class specifies implementation for
# com.sun.identity.plugin.log.Logger interface.
com.sun.identity.plugin.log.class=com.sun.identity.plugin.log.im
pl.FedletLogger

# com.sun.identity.plugin.session.class specifies implementation
for
# com.sun.identity.plugin.session.SessionProvider interface.
```

```
com.sun.identity.plugin.session.class=com.sun.identity.plugin.se
ssion.impl.FedletSessionProvider

# com.sun.identity.plugin.monitoring.agent.class specifies
implementation for
# com.sun.identity.plugin.monitoring.FedMonAgent interface.
com.sun.identity.plugin.monitoring.agent.class=com.sun.identity.
plugin.monitoring.impl.FedletAgentProvider

# com.sun.identity.plugin.monitoring.saml2.class specifies
implementation for
# com.sun.identity.plugin.monitoring.FedMonSAML2Svc interface.
com.sun.identity.plugin.monitoring.saml2.class=com.sun.identity.
plugin.monitoring.impl.FedletMonSAML2SvcProvider

# com.sun.identity.saml.xmlsig.keyprovider.class specified the
implementation
# class for com.sun.identity.saml.xmlsig.KeyProvider interface
com.sun.identity.saml.xmlsig.keyprovider.class=com.sun.identity.
saml.xmlsig.JKSKeyProvider

# com.sun.identity.saml.xmlsig.signatureprovider.class specified
the
# implementation class for
com.sun.identity.saml.xmlsig.SignatureProvider
# interface
com.sun.identity.saml.xmlsig.signatureprovider.class=com.sun.ide
ntity.saml.xmlsig.AMSignatureProvider

com.iplanet.am.server.protocol=http
com.iplanet.am.server.host=am.example.com
com.iplanet.am.server.port=8080
com.iplanet.am.services.deploymentDescriptor=/openam
com.iplanet.am.logstatus=ACTIVE

# Name of the webcontainer.
# Even though the servlet/JSP are web container independent,
# Access/Federation Manager uses servlet 2.3 API
request.setCharacterEncoding()
# to decode incoming non English characters. These APIs will not
work if
# Access/Federation Manager is deployed on Sun Java System Web
Server 6.1.
# We use gx_charset mechanism to correctly decode incoming data
in
```

```
# Sun Java System Web Server 6.1 and S1AS7.0. Possible values
# are BEA6.1, BEA 8.1, IBM5.1 or IAS7.0.
# If the web container is Sun Java System Webserver, the tag is
not replaced.
com.sun.identity.webcontainer=WEB_CONTAINER

# Identify saml xml signature keystore file, keystore password
file
# key password file
com.sun.identity.saml.xmlsig.keystore=%BASE_DIR%/security/keysto
res/keystore.jks
com.sun.identity.saml.xmlsig.storepass=%BASE_DIR%/.storepass
com.sun.identity.saml.xmlsig.keypass=%BASE_DIR%/.keypass
com.sun.identity.saml.xmlsig.certalias=test

# Type of keystore used for saml xml signature. Default is JKS.
#
# com.sun.identity.saml.xmlsig.storetype=JKS

# Specifies the implementation class for
# com.sun.identity.saml.xmlsig.PasswordDecoder interface.
com.sun.identity.saml.xmlsig.passwordDecoder=com.sun.identity.fe
dlet.FedletEncodeDecode

# The following key is used to specify the maximum content-
length
# for an HttpRequest that will be accepted by the OpenSSO
# The default value is 16384 which is 16k
com.iplanet.services.comm.server.pllrequest.maxContentLength=163
84

# The following keys are used to configure the Debug service.
# Possible values for the key 'level' are: off | error | warning
| message.
# The key 'directory' specifies the output directory where the
debug files
# will be created.
# Trailing spaces are significant.
# Windows: Use forward slashes "/" separate directories, not
backslash "\".
# Windows: Spaces in the file name are allowed for Windows.
#
com.iplanet.services.debug.level=message
com.iplanet.services.debug.directory=%BASE_DIR%%SERVER_URI%/debu
g
```

```
# The following keys are used to configure the Stats service.
# Possible values for the key 'level' are: off | file | console
# Stats state 'file' will write to a file under the specified
directory,
# and 'console' will write into  webserver log files
# The key 'directory' specifies the output directory where the
debug files
# will be created.
# Trailing spaces are significant.
# Windows: Use forward slashes "/" separate directories, not
backslash "\".
# Windows: Spaces in the file name are allowed for Windows.
# Stats interval should be atleast 5 secs to avoid CPU
saturation,
# the product would assume any thing less than 5 secs is 5 secs.
com.iplanet.am.stats.interval=60
com.iplanet.services.stats.state=file
com.iplanet.services.stats.directory=%BASE_DIR%/var/stats

# The key that will be used to encrypt and decrypt passwords.
am.encryption.pwd=@AM_ENC_PWD@

# SecureRandom Properties: The key
# "com.iplanet.security.SecureRandomFactoryImpl"
# specifies the factory class name for SecureRandomFactory
# Available impl classes are:
#   com.iplanet.am.util.JSSSecureRandomFactoryImpl (uses JSS)
#   com.iplanet.am.util.SecureRandomFactoryImpl (pure Java)
com.iplanet.security.SecureRandomFactoryImpl=com.iplanet.am.util
.SecureRandomFactoryImpl

# SocketFactory properties: The key
"com.iplanet.security.SSLSocketFactoryImpl"
# specifies the factory class name for LDAPSocketFactory
# Available classes are:
#    com.iplanet.services.ldap.JSSSocketFactory (uses JSS)
#    com.sun.identity.shared.ldap.factory.JSSESocketFactory
(pure Java)
com.iplanet.security.SSLSocketFactoryImpl=com.sun.identity.share
d.ldap.factory.JSSESocketFactory

# Encryption: The key "com.iplanet.security.encryptor" specifies
# the encrypting class implementation.
# Available classes are:
```

```
#    com.iplanet.services.util.JCEEncryption
#    com.iplanet.services.util.JSSEncryption
com.iplanet.security.encryptor=com.iplanet.services.util.JCEEncr
yption

# Determines if JSS will be added with highest priority to JCE
# Set this to "true" if other JCE providers should be used for
# digial signatures and encryptions.
com.sun.identity.jss.donotInstallAtHighestPriority=true

# Configuration File (serverconfig.xml) Location
com.iplanet.services.configpath=@BASE_DIR@
```

▼ fedlet.cot

The following example of `$HOME/.openig/SAML/fedlet.cot` defines a circle of trust between AM as the IDP and PingGateway as the SP:

```
cot-name=Circle of Trust
sun-fm-cot-status=Active
sun-fm-trusted-providers=openam, sp
sun-fm-saml2-readerservice-url=
sun-fm-saml2-writerservice-url=
```

▼ idp.xml

The following example of `$HOME/.openig/SAML/idp.xml` defines a SAML configuration file for the AM IDP, `idp`:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<EntityDescriptor entityID="openam"
xmlns="urn:oasis:names:tc:SAML:2.0:metadata"
xmlns:query="urn:oasis:names:tc:SAML:metadata:ext:query"
xmlns:mdattr="urn:oasis:names:tc:SAML:metadata:attribute"
xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
xmlns:xenc11="http://www.w3.org/2009/xmlenc11#"
xmlns:alg="urn:oasis:names:tc:SAML:metadata:algsupport"
xmlns:x509qry="urn:oasis:names:tc:SAML:metadata:X509:query"
xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <IDPSSODescriptor
protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol
">
        <KeyDescriptor use="signing">
            <ds:KeyInfo>
```

```
                <ds:X509Data>
                    <ds:X509Certificate>
...
                    </ds:X509Certificate>
                </ds:X509Data>
            </ds:KeyInfo>
        </KeyDescriptor>
        <KeyDescriptor use="encryption">
            <ds:KeyInfo>
                <ds:X509Data>
                    <ds:X509Certificate>
...
                    </ds:X509Certificate>
                </ds:X509Data>
            </ds:KeyInfo>
            <EncryptionMethod
Algorithm="http://www.w3.org/2009/xmlenc11#rsa-oaep">
                <ds:DigestMethod
Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
                <xenc11:MGF
Algorithm="http://www.w3.org/2009/xmlenc11#mgf1sha256"/>
            </EncryptionMethod>
            <EncryptionMethod
Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc">
                <xenc:KeySize>128</xenc:KeySize>
            </EncryptionMethod>
        </KeyDescriptor>
        <ArtifactResolutionService index="0"
Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP"
Location="http://am.example.com:8088/openam/ArtifactResolver/met
aAlias/idp"/>
        <SingleLogoutService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect"
Location="http://am.example.com:8088/openam/IDPSloRedirect/metaA
lias/idp"
ResponseLocation="http://am.example.com:8088/openam/IDPSloRedire
ct/metaAlias/idp"/>
        <SingleLogoutService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
Location="http://am.example.com:8088/openam/IDPSloPOST/metaAlias
/idp"
ResponseLocation="http://am.example.com:8088/openam/IDPSloPOST/m
etaAlias/idp"/>
        <SingleLogoutService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP"
```

```
Location="http://am.example.com:8088/openam/IDPSloSoap/metaAlias
/idp"/>
        <ManageNameIDService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect"
Location="http://am.example.com:8088/openam/IDPMniRedirect/metaA
lias/idp"
ResponseLocation="http://am.example.com:8088/openam/IDPMniRedire
ct/metaAlias/idp"/>
        <ManageNameIDService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
Location="http://am.example.com:8088/openam/IDPMniPOST/metaAlias
/idp"
ResponseLocation="http://am.example.com:8088/openam/IDPMniPOST/m
etaAlias/idp"/>
        <ManageNameIDService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP"
Location="http://am.example.com:8088/openam/IDPMniSoap/metaAlias
/idp"/>
        <NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-
format:persistent</NameIDFormat>
        <NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-
format:transient</NameIDFormat>
        <NameIDFormat>urn:oasis:names:tc:SAML:1.1:nameid-
format:emailAddress</NameIDFormat>
        <NameIDFormat>urn:oasis:names:tc:SAML:1.1:nameid-
format:unspecified</NameIDFormat>
        <NameIDFormat>urn:oasis:names:tc:SAML:1.1:nameid-
format:WindowsDomainQualifiedName</NameIDFormat>
        <NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-
format:kerberos</NameIDFormat>
        <NameIDFormat>urn:oasis:names:tc:SAML:1.1:nameid-
format:X509SubjectName</NameIDFormat>
        <SingleSignOnService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect"
Location="http://am.example.com:8088/openam/SSORedirect/metaAlia
s/idp"/>
        <SingleSignOnService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
Location="http://am.example.com:8088/openam/SSOPOST/metaAlias/id
p"/>
        <SingleSignOnService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP"
Location="http://am.example.com:8088/openam/SSOSoap/metaAlias/id
p"/>
        <NameIDMappingService
```

```
Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP"
Location="http://am.example.com:8088/openam/NIMSoap/metaAlias/id
p"/>
        <AssertionIDRequestService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP"
Location="http://am.example.com:8088/openam/AIDReqSoap/IDPRole/m
etaAlias/idp"/>
        <AssertionIDRequestService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:URI"
Location="http://am.example.com:8088/openam/AIDReqUri/IDPRole/me
taAlias/idp"/>
    </IDPSSODescriptor>
</EntityDescriptor>
```

▼ idp-extended.xml

The following example of `$HOME/.openig/SAML/idp-extended.xml` defines an AM SAML descriptor file for the IDP:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!--
   DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS HEADER.

   Copyright (c) 2002-2010 Sun Microsystems Inc. All Rights
Reserved

   The contents of this file are subject to the terms
   of the Common Development and Distribution License
   (the License). You may not use this file except in
   compliance with the License.

   You can obtain a copy of the License at
   https://opensso.dev.java.net/public/CDDLv1.0.html or
   opensso/legal/CDDLv1.0.txt
   See the License for the specific language governing
   permission and limitations under the License.

   When distributing Covered Code, include this CDDL
   Header Notice in each file and include the License file
   at opensso/legal/CDDLv1.0.txt.
   If applicable, add the following below the CDDL Header,
   with the fields enclosed by brackets [] replaced by
   your own identifying information:
   "Portions Copyrighted [year] [name of copyright owner]"
```

```
    Portions Copyrighted 2010-2017 Ping Identity Corporation.
-->
<EntityConfig entityID="openam" hosted="0"
xmlns="urn:sun:fm:SAML:2.0:entityconfig">
    <IDPSSOConfig>
        <Attribute name="description">
            <Value/>
        </Attribute>
        <Attribute name="cotlist">
            <Value>Circle of Trust</Value>
        </Attribute>
    </IDPSSOConfig>
    <AttributeAuthorityConfig>
        <Attribute name="cotlist">
            <Value>Circle of Trust</Value>
        </Attribute>
    </AttributeAuthorityConfig>
    <XACMLPDPConfig>
        <Attribute name="wantXACMLAuthzDecisionQuerySigned">
            <Value></Value>
        </Attribute>
        <Attribute name="cotlist">
            <Value>Circle of Trust</Value>
        </Attribute>
    </XACMLPDPConfig>
</EntityConfig>
```

▼ sp.xml

NOTE

The SAML library component validates the SP's AssertionConsumerService Location against the incoming IDP SAML Assertion, based on the request information including the port. Always specify the port in the Location value of `AssertionConsumerService` even when using defaults of 443 or 80:

```
<AssertionConsumerService isDefault="true"
                          index="0"

Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"

Location="https://sp.example.com:443/fedletapplication" />
```

The following example of `$HOME/.openig/SAML/sp.xml` defines a SAML configuration file for the PingGateway SP, `sp`.

```
<!--
   DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS HEADER.

   Copyright (c) 2002-2010 Sun Microsystems Inc. All Rights
Reserved

   The contents of this file are subject to the terms
   of the Common Development and Distribution License
   (the License). You may not use this file except in
   compliance with the License.

   You can obtain a copy of the License at
   https://opensso.dev.java.net/public/CDDLv1.0.html or
   opensso/legal/CDDLv1.0.txt
   See the License for the specific language governing
   permission and limitations under the License.

   When distributing Covered Code, include this CDDL
   Header Notice in each file and include the License file
   at opensso/legal/CDDLv1.0.txt.
   If applicable, add the following below the CDDL Header,
   with the fields enclosed by brackets [] replaced by
   your own identifying information:
   "Portions Copyrighted [year] [name of copyright owner]"

   Portions Copyrighted 2010-2017 Ping Identity Corporation.
-->
<EntityDescriptor entityID="sp"
xmlns="urn:oasis:names:tc:SAML:2.0:metadata">
    <SPSSODescriptor AuthnRequestsSigned="false"
WantAssertionsSigned="false"
protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol
">
        <SingleLogoutService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect"
Location="http://sp.example.com:8080/saml/fedletSloRedirect"
ResponseLocation="http://sp.example.com:8080/saml/fedletSloRedir
ect"/>
        <SingleLogoutService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
Location="http://sp.example.com:8080/saml/fedletSloPOST"
ResponseLocation="http://sp.example.com:8080/saml/fedletSloPOST"
/>
        <SingleLogoutService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP"
```

```
Location="http://sp.example.com:8080/saml/fedletSloSoap"/>
        <NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-
format:transient</NameIDFormat>
        <AssertionConsumerService isDefault="true" index="0"
Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
Location="http://sp.example.com:8080/saml/fedletapplication/meta
Alias/sp"/>
        <AssertionConsumerService index="1"
Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Artifact"
Location="http://sp.example.com:8080/saml/fedletapplication/meta
Alias/sp"/>
    </SPSSODescriptor>
    <RoleDescriptor xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns:query="urn:oasis:names:tc:SAML:metadata:ext:query"
xsi:type="query:AttributeQueryDescriptorType"
protocolSupportEnumeration=
"urn:oasis:names:tc:SAML:2.0:protocol">
    </RoleDescriptor>
    <XACMLAuthzDecisionQueryDescriptor
WantAssertionsSigned="false"
protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol
">
    </XACMLAuthzDecisionQueryDescriptor>
</EntityDescriptor>
```

▼ sp-extended.xml

The following example of `$HOME/.openig/SAML/sp-extended.xml` defines an AM SAML descriptor file for the SP:

```
<!--
   DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS HEADER.

   Copyright (c) 2002-2010 Sun Microsystems Inc. All Rights
Reserved

   The contents of this file are subject to the terms
   of the Common Development and Distribution License
   (the License). You may not use this file except in
   compliance with the License.

   You can obtain a copy of the License at
   https://opensso.dev.java.net/public/CDDLv1.0.html or
   opensso/legal/CDDLv1.0.txt
```

```
    See the License for the specific language governing
    permission and limitations under the License.

    When distributing Covered Code, include this CDDL
    Header Notice in each file and include the License file
    at opensso/legal/CDDLv1.0.txt.
    If applicable, add the following below the CDDL Header,
    with the fields enclosed by brackets [] replaced by
    your own identifying information:
    "Portions Copyrighted [year] [name of copyright owner]"

    Portions Copyrighted 2010-2017 Ping Identity Corporation.
-->
<EntityConfig xmlns="urn:sun:fm:SAML:2.0:entityconfig"
xmlns:fm="urn:sun:fm:SAML:2.0:entityconfig" hosted="1"
entityID="sp">
    <SPSSOConfig metaAlias="/sp">
        <Attribute name="description">
            <Value></Value>
        </Attribute>
        <Attribute name="signingCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="encryptionCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="basicAuthOn">
            <Value>false</Value>
        </Attribute>
        <Attribute name="basicAuthUser">
            <Value></Value>
        </Attribute>
        <Attribute name="basicAuthPassword">
            <Value></Value>
        </Attribute>
        <Attribute name="autofedEnabled">
            <Value>false</Value>
        </Attribute>
        <Attribute name="autofedAttribute">
            <Value></Value>
        </Attribute>
        <Attribute name="transientUser">
            <Value>anonymous</Value>
        </Attribute>
        <Attribute name="spAdapter">
```

```
            <Value></Value>
        </Attribute>
        <Attribute name="spAdapterEnv">
            <Value></Value>
        </Attribute>
        <Attribute name="fedletAdapter">
            <Value></Value>
        </Attribute>
        <Attribute name="fedletAdapterEnv">
            <Value></Value>
        </Attribute>
        <Attribute name="spAccountMapper">

<Value>com.sun.identity.saml2.plugins.DefaultLibrarySPAccountMap
per</Value>
        </Attribute>
        <Attribute name="spAttributeMapper">

<Value>com.sun.identity.saml2.plugins.DefaultSPAttributeMapper</
Value>
        </Attribute>
        <Attribute name="spAuthncontextMapper">

<Value>com.sun.identity.saml2.plugins.DefaultSPAuthnContextMappe
r</Value>
        </Attribute>
        <Attribute name="spAuthncontextClassrefMapping">

<Value>urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedT
ransport|0|default</Value>
        </Attribute>
        <Attribute name="spAuthncontextComparisonType">
           <Value>exact</Value>
        </Attribute>
        <Attribute name="attributeMap">
           <Value>*=*</Value>
        </Attribute>
        <Attribute name="saml2AuthModuleName">
           <Value></Value>
       </Attribute>
       <Attribute name="localAuthURL">
           <Value></Value>
       </Attribute>
       <Attribute name="intermediateUrl">
           <Value></Value>
```

```xml
        </Attribute>
        <Attribute name="defaultRelayState">
            <Value></Value>
        </Attribute>
        <Attribute name="appLogoutUrl">
            <Value>http://sp.example.com:8080/saml/logout</Value>
        </Attribute>
        <Attribute name="assertionTimeSkew">
            <Value>300</Value>
        </Attribute>
        <Attribute name="wantAttributeEncrypted">
            <Value></Value>
        </Attribute>
        <Attribute name="wantAssertionEncrypted">
            <Value></Value>
        </Attribute>
        <Attribute name="wantNameIDEncrypted">
            <Value></Value>
        </Attribute>
        <Attribute name="wantPOSTResponseSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="wantArtifactResponseSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="wantLogoutRequestSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="wantLogoutResponseSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="wantMNIRequestSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="wantMNIResponseSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="cotlist">
            <Value>Circle of Trust</Value></Attribute>
        <Attribute name="saeAppSecretList">
        </Attribute>
        <Attribute name="saeSPUrl">
            <Value></Value>
        </Attribute>
        <Attribute name="saeSPLogoutUrl">
        </Attribute>
        <Attribute name="ECPRequestIDPListFinderImpl">

<Value>com.sun.identity.saml2.plugins.ECPIDPFinder</Value>
        </Attribute>
        <Attribute name="ECPRequestIDPList">
            <Value></Value>
        </Attribute>
        <Attribute name="enableIDPProxy">
            <Value>false</Value>
        </Attribute>
        <Attribute name="idpProxyList">
            <Value></Value>
        </Attribute>
        <Attribute name="idpProxyCount">
            <Value>0</Value>
        </Attribute>
        <Attribute name="useIntroductionForIDPProxy">
            <Value>false</Value>
        </Attribute>
     </SPSSOConfig>
     <AttributeQueryConfig metaAlias="/attrQuery">
         <Attribute name="signingCertAlias">
             <Value></Value>
         </Attribute>
         <Attribute name="encryptionCertAlias">
             <Value></Value>
         </Attribute>
         <Attribute name="wantNameIDEncrypted">
            <Value></Value>
         </Attribute>
         <Attribute name="cotlist">
            <Value>Circle of Trust</Value>
         </Attribute>
     </AttributeQueryConfig>
     <XACMLAuthzDecisionQueryConfig metaAlias="/pep">
         <Attribute name="signingCertAlias">
             <Value></Value>
         </Attribute>
         <Attribute name="encryptionCertAlias">
             <Value></Value>
         </Attribute>
         <Attribute name="basicAuthOn">
             <Value>false</Value>
         </Attribute>
         <Attribute name="basicAuthUser">
             <Value></Value>
         </Attribute>
         <Attribute name="basicAuthPassword">
             <Value></Value>
         </Attribute>
         <Attribute name="wantXACMLAuthzDecisionResponseSigned">
             <Value></Value>
         </Attribute>
         <Attribute name="wantAssertionEncrypted">
             <Value></Value>
         </Attribute>
         <Attribute name="cotlist">
             <Value>Circle of Trust</Value>
         </Attribute>
    </XACMLAuthzDecisionQueryConfig>
 </EntityConfig>
```

PingOne as IDP

▼ FederationConfig.properties

```
# If a component wants to use a different datastore provider
than the
# default one defined above, it can define a property like
follows:
# com.sun.identity.plugin.datastore.class.<componentName>=
<provider class>

# com.sun.identity.plugin.configuration.class specifies
implementation for
# com.sun.identity.plugin.configuration.ConfigurationInstance
interface.
com.sun.identity.plugin.configuration.class=com.sun.identity.plu
gin.configuration.impl.FedletConfigurationImpl

# Specifies implementation for
# com.sun.identity.plugin.datastore.DataStoreProvider interface.
# This property defines the default datastore provider.
com.sun.identity.plugin.datastore.class.default=com.sun.identity
.plugin.datastore.impl.FedletDataStoreProvider

# Specifies implementation for
# org.forgerock.openam.federation.plugin.rooturl.RootUrlProvider
```

```
interface.
# This property defines the default base url provider.
com.sun.identity.plugin.root.url.class.default=org.forgerock.ope
nam.federation.plugin.rooturl.impl.FedletRootUrlProvider

# com.sun.identity.plugin.log.class specifies implementation for
# com.sun.identity.plugin.log.Logger interface.
com.sun.identity.plugin.log.class=com.sun.identity.plugin.log.im
pl.FedletLogger

# com.sun.identity.plugin.session.class specifies implementation
for
# com.sun.identity.plugin.session.SessionProvider interface.
com.sun.identity.plugin.session.class=com.sun.identity.plugin.se
ssion.impl.FedletSessionProvider

# com.sun.identity.plugin.monitoring.agent.class specifies
implementation for
# com.sun.identity.plugin.monitoring.FedMonAgent interface.
com.sun.identity.plugin.monitoring.agent.class=com.sun.identity.
plugin.monitoring.impl.FedletAgentProvider

# com.sun.identity.plugin.monitoring.saml2.class specifies
implementation for
# com.sun.identity.plugin.monitoring.FedMonSAML2Svc interface.
com.sun.identity.plugin.monitoring.saml2.class=com.sun.identity.
plugin.monitoring.impl.FedletMonSAML2SvcProvider

# com.sun.identity.saml.xmlsig.keyprovider.class specified the
implementation
# class for com.sun.identity.saml.xmlsig.KeyProvider interface
com.sun.identity.saml.xmlsig.keyprovider.class=com.sun.identity.
saml.xmlsig.JKSKeyProvider

# com.sun.identity.saml.xmlsig.signatureprovider.class specified
the
# implementation class for
com.sun.identity.saml.xmlsig.SignatureProvider
# interface
com.sun.identity.saml.xmlsig.signatureprovider.class=com.sun.ide
ntity.saml.xmlsig.AMSignatureProvider

com.iplanet.am.server.protocol=http
com.iplanet.am.server.host=am.example.com
com.iplanet.am.server.port=8080
com.iplanet.am.services.deploymentDescriptor=/openam
com.iplanet.am.logstatus=ACTIVE

# Name of the webcontainer.
# Even though the servlet/JSP are web container independent,
# Access/Federation Manager uses servlet 2.3 API
request.setCharacterEncoding()
# to decode incoming non English characters. These APIs will not
work if
# Access/Federation Manager is deployed on Sun Java System Web
Server 6.1.
# We use gx_charset mechanism to correctly decode incoming data
in
# Sun Java System Web Server 6.1 and S1AS7.0. Possible values
# are BEA6.1, BEA 8.1, IBM5.1 or IAS7.0.
# If the web container is Sun Java System Webserver, the tag is
not replaced.
com.sun.identity.webcontainer=WEB_CONTAINER

# Identify saml xml signature keystore file, keystore password
file
# key password file
com.sun.identity.saml.xmlsig.keystore=%BASE_DIR%/security/keysto
res/keystore.jks
com.sun.identity.saml.xmlsig.storepass=%BASE_DIR%/.storepass
com.sun.identity.saml.xmlsig.keypass=%BASE_DIR%/.keypass
com.sun.identity.saml.xmlsig.certalias=test

# Type of keystore used for saml xml signature. Default is JKS.
#
# com.sun.identity.saml.xmlsig.storetype=JKS

# Specifies the implementation class for
# com.sun.identity.saml.xmlsig.PasswordDecoder interface.
com.sun.identity.saml.xmlsig.passwordDecoder=com.sun.identity.fe
dlet.FedletEncodeDecode

# The following key is used to specify the maximum content-
length
# for an HttpRequest that will be accepted by the OpenSSO
# The default value is 16384 which is 16k
com.iplanet.services.comm.server.pllrequest.maxContentLength=163
84

# The following keys are used to configure the Debug service.
# Possible values for the key 'level' are: off | error | warning
| message.
# The key 'directory' specifies the output directory where the
debug files
# will be created.
# Trailing spaces are significant.
# Windows: Use forward slashes "/" separate directories, not
backslash "\".
# Windows: Spaces in the file name are allowed for Windows.
#
com.iplanet.services.debug.level=message
com.iplanet.services.debug.directory=%BASE_DIR%%SERVER_URI%/debu
g

# The following keys are used to configure the Stats service.
# Possible values for the key 'level' are: off | file | console
# Stats state 'file' will write to a file under the specified
directory,
# and 'console' will write into  webserver log files
# The key 'directory' specifies the output directory where the
debug files
# will be created.
# Trailing spaces are significant.
# Windows: Use forward slashes "/" separate directories, not
backslash "\".
# Windows: Spaces in the file name are allowed for Windows.
# Stats interval should be atleast 5 secs to avoid CPU
saturation,
# the product would assume any thing less than 5 secs is 5 secs.
com.iplanet.am.stats.interval=60
com.iplanet.services.stats.state=file
com.iplanet.services.stats.directory=%BASE_DIR%/var/stats

# The key that will be used to encrypt and decrypt passwords.
am.encryption.pwd=@AM_ENC_PWD@

# SecureRandom Properties: The key
# "com.iplanet.security.SecureRandomFactoryImpl"
# specifies the factory class name for SecureRandomFactory
# Available impl classes are:
#   com.iplanet.am.util.JSSSecureRandomFactoryImpl (uses JSS)
#   com.iplanet.am.util.SecureRandomFactoryImpl (pure Java)
com.iplanet.security.SecureRandomFactoryImpl=com.iplanet.am.util
.SecureRandomFactoryImpl
```

```
# SocketFactory properties: The key
"com.iplanet.security.SSLSocketFactoryImpl"
# specifies the factory class name for LDAPSocketFactory
# Available classes are:
#    com.iplanet.services.ldap.JSSSocketFactory (uses JSS)
#    com.sun.identity.shared.ldap.factory.JSSESocketFactory
(pure Java)
com.iplanet.security.SSLSocketFactoryImpl=com.sun.identity.share
d.ldap.factory.JSSESocketFactory

# Encryption: The key "com.iplanet.security.encryptor" specifies
# the encrypting class implementation.
# Available classes are:
#    com.iplanet.services.util.JCEEncryption
#    com.iplanet.services.util.JSSEncryption
com.iplanet.security.encryptor=com.iplanet.services.util.JCEEncr
yption

# Determines if JSS will be added with highest priority to JCE
# Set this to "true" if other JCE providers should be used for
# digial signatures and encryptions.
com.sun.identity.jss.donotInstallAtHighestPriority=true

# Configuration File (serverconfig.xml) Location
com.iplanet.services.configpath=@BASE_DIR@
```

▼ fedlet.cot

```
cot-name=Circle of Trust
sun-fm-cot-status=Active
sun-fm-trusted-providers=idp-entityID, sp
sun-fm-saml2-readerservice-url=
sun-fm-saml2-writerservice-url=
```

▼ idp-extended.xml

```
<EntityConfig entityID="idp-entityID" hosted="0"
xmlns="urn:sun:fm:SAML:2.0:entityconfig">
    <IDPSSOConfig>
        <Attribute name="description">
            <Value/>
        </Attribute>
        <Attribute name="cotlist">
            <Value>Circle of Trust</Value>
        </Attribute>
    </IDPSSOConfig>
    <AttributeAuthorityConfig>
        <Attribute name="cotlist">
            <Value>Circle of Trust</Value>
        </Attribute>
    </AttributeAuthorityConfig>
    <XACMLPDPConfig>
        <Attribute name="wantXACMLAuthzDecisionQuerySigned">
            <Value></Value>
        </Attribute>
        <Attribute name="cotlist">
            <Value>Circle of Trust</Value>
        </Attribute>
    </XACMLPDPConfig>
</EntityConfig>
```

▼ sp.xml

```
<EntityDescriptor entityID="sp"
xmlns="urn:oasis:names:tc:SAML:2.0:metadata">
    <SPSSODescriptor AuthnRequestsSigned="false"
WantAssertionsSigned="false"
protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol
">
        <SingleLogoutService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect"
Location="https://sp.example.com:8443/home/saml/fedletSloRedirec
t"
ResponseLocation="https://sp.example.com:8443/home/saml/fedletSl
oRedirect"/>
        <SingleLogoutService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
Location="https://sp.example.com:8443/home/saml/fedletSloPOST"
ResponseLocation="https://sp.example.com:8443/home/saml/fedletSl
oPOST"/>
        <SingleLogoutService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP"
Location="https://sp.example.com:8443/home/saml/fedletSloSoap"/>
        <NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-
format:transient</NameIDFormat>
        <AssertionConsumerService isDefault="true" index="0"
Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
Location="https://sp.example.com:8443/home/saml/fedletapplicatio
n"/>
        <AssertionConsumerService index="1"
```

```
Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Artifact"
Location="https://sp.example.com:8443/home/saml/fedletapplicatio
n"/>
    </SPSSODescriptor>
    <RoleDescriptor xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns:query="urn:oasis:names:tc:SAML:metadata:ext:query"
xsi:type="query:AttributeQueryDescriptorType"
protocolSupportEnumeration=
"urn:oasis:names:tc:SAML:2.0:protocol">
    </RoleDescriptor>
    <XACMLAuthzDecisionQueryDescriptor
WantAssertionsSigned="false"
protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol
">
    </XACMLAuthzDecisionQueryDescriptor>
</EntityDescriptor>
```

▼ sp-extended.xml

```
<EntityConfig xmlns="urn:sun:fm:SAML:2.0:entityconfig"
xmlns:fm="urn:sun:fm:SAML:2.0:entityconfig" hosted="1"
entityID="sp">
    <SPSSOConfig metaAlias="/sp">
        <Attribute name="description">
            <Value></Value>
        </Attribute>
        <Attribute name="signingCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="encryptionCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="basicAuthOn">
            <Value>false</Value>
        </Attribute>
        <Attribute name="basicAuthUser">
            <Value></Value>
        </Attribute>
        <Attribute name="basicAuthPassword">
            <Value></Value>
        </Attribute>
        <Attribute name="autofedEnabled">
            <Value>false</Value>
        </Attribute>
        <Attribute name="autofedAttribute">
            <Value></Value>
        </Attribute>
        <Attribute name="transientUser">
            <Value>anonymous</Value>
        </Attribute>
        <Attribute name="spAdapter">
            <Value></Value>
        </Attribute>
        <Attribute name="spAdapterEnv">
            <Value></Value>
        </Attribute>
        <Attribute name="fedletAdapter">
            <Value></Value>
        </Attribute>
        <Attribute name="fedletAdapterEnv">
            <Value></Value>
        </Attribute>
        <Attribute name="spAccountMapper">

<Value>com.sun.identity.saml2.plugins.DefaultLibrarySPAccountMap
per</Value>
        </Attribute>
        <Attribute name="spAttributeMapper">

<Value>com.sun.identity.saml2.plugins.DefaultSPAttributeMapper</
Value>
        </Attribute>
        <Attribute name="spAuthncontextMapper">

<Value>com.sun.identity.saml2.plugins.DefaultSPAuthnContextMappe
r</Value>
        </Attribute>
        <Attribute name="spAuthncontextClassrefMapping">

<Value>urn:oasis:names:tc:SAML:2.0:ac:classes:unspecified|0|defa
ult</Value>
        </Attribute>
        <Attribute name="spAuthncontextComparisonType">
           <Value>exact</Value>
        </Attribute>
        <Attribute name="attributeMap">
           <Value>*=*</Value>
        </Attribute>
        <Attribute name="saml2AuthModuleName">
```

```
            <Value></Value>
        </Attribute>
        <Attribute name="localAuthURL">
            <Value></Value>
        </Attribute>
        <Attribute name="intermediateUrl">
            <Value></Value>
        </Attribute>
        <Attribute name="defaultRelayState">
            <Value></Value>
        </Attribute>
        <Attribute name="appLogoutUrl">

<Value>https://sp.example.com:8443/home/saml/logout</Value>
        </Attribute>
        <Attribute name="assertionTimeSkew">
            <Value>300</Value>
        </Attribute>
        <Attribute name="wantAttributeEncrypted">
            <Value></Value>
        </Attribute>
        <Attribute name="wantAssertionEncrypted">
            <Value></Value>
        </Attribute>
        <Attribute name="wantNameIDEncrypted">
            <Value></Value>
        </Attribute>
        <Attribute name="wantPOSTResponseSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="wantArtifactResponseSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="wantLogoutRequestSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="wantLogoutResponseSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="wantMNIRequestSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="wantMNIResponseSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="cotlist">
            <Value>Circle of Trust</Value></Attribute>
        <Attribute name="saeAppSecretList">
        </Attribute>
        <Attribute name="saeSPUrl">
            <Value></Value>
        </Attribute>
        <Attribute name="saeSPLogoutUrl">
        </Attribute>
        <Attribute name="ECPRequestIDPListFinderImpl">

<Value>com.sun.identity.saml2.plugins.ECPIDPFinder</Value>
        </Attribute>
        <Attribute name="ECPRequestIDPList">
            <Value></Value>
        </Attribute>
        <Attribute name="enableIDPProxy">
            <Value>false</Value>
        </Attribute>
        <Attribute name="idpProxyList">
            <Value></Value>
        </Attribute>
        <Attribute name="idpProxyCount">
            <Value>0</Value>
        </Attribute>
        <Attribute name="useIntroductionForIDPProxy">
            <Value>false</Value>
        </Attribute>
     </SPSSOConfig>
     <AttributeQueryConfig metaAlias="/attrQuery">
         <Attribute name="signingCertAlias">
             <Value></Value>
         </Attribute>
         <Attribute name="encryptionCertAlias">
             <Value></Value>
         </Attribute>
         <Attribute name="wantNameIDEncrypted">
            <Value></Value>
         </Attribute>
         <Attribute name="cotlist">
            <Value>Circle of Trust</Value>
         </Attribute>
     </AttributeQueryConfig>
     <XACMLAuthzDecisionQueryConfig metaAlias="/pep">
         <Attribute name="signingCertAlias">
```

```
            <Value></Value>
        </Attribute>
        <Attribute name="encryptionCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="basicAuthOn">
            <Value>false</Value>
        </Attribute>
        <Attribute name="basicAuthUser">
            <Value></Value>
        </Attribute>
        <Attribute name="basicAuthPassword">
            <Value></Value>
        </Attribute>
        <Attribute name="wantXACMLAuthzDecisionResponseSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="wantAssertionEncrypted">
            <Value></Value>
        </Attribute>
        <Attribute name="cotlist">
            <Value>Circle of Trust</Value>
        </Attribute>
    </XACMLAuthzDecisionQueryConfig>
</EntityConfig>
```

Token transformation

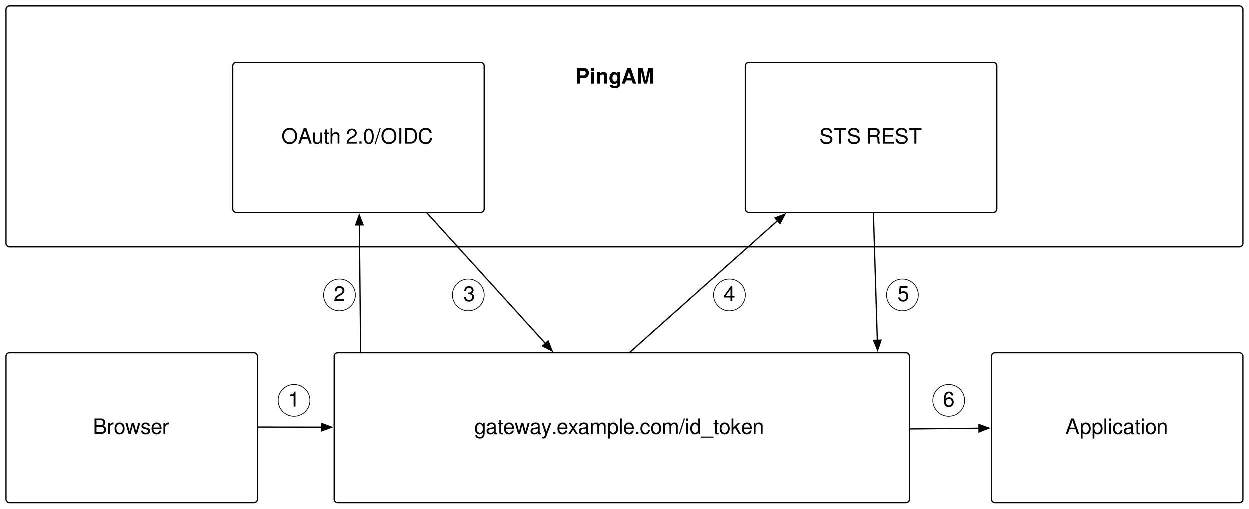
- OIDC ID tokens to SAML assertions
- OAuth 2.0 token exchange

OIDC ID tokens to SAML assertions

This page builds on the example in OpenID Connect to transform OpenID Connect ID tokens into SAML 2.0 assertions.

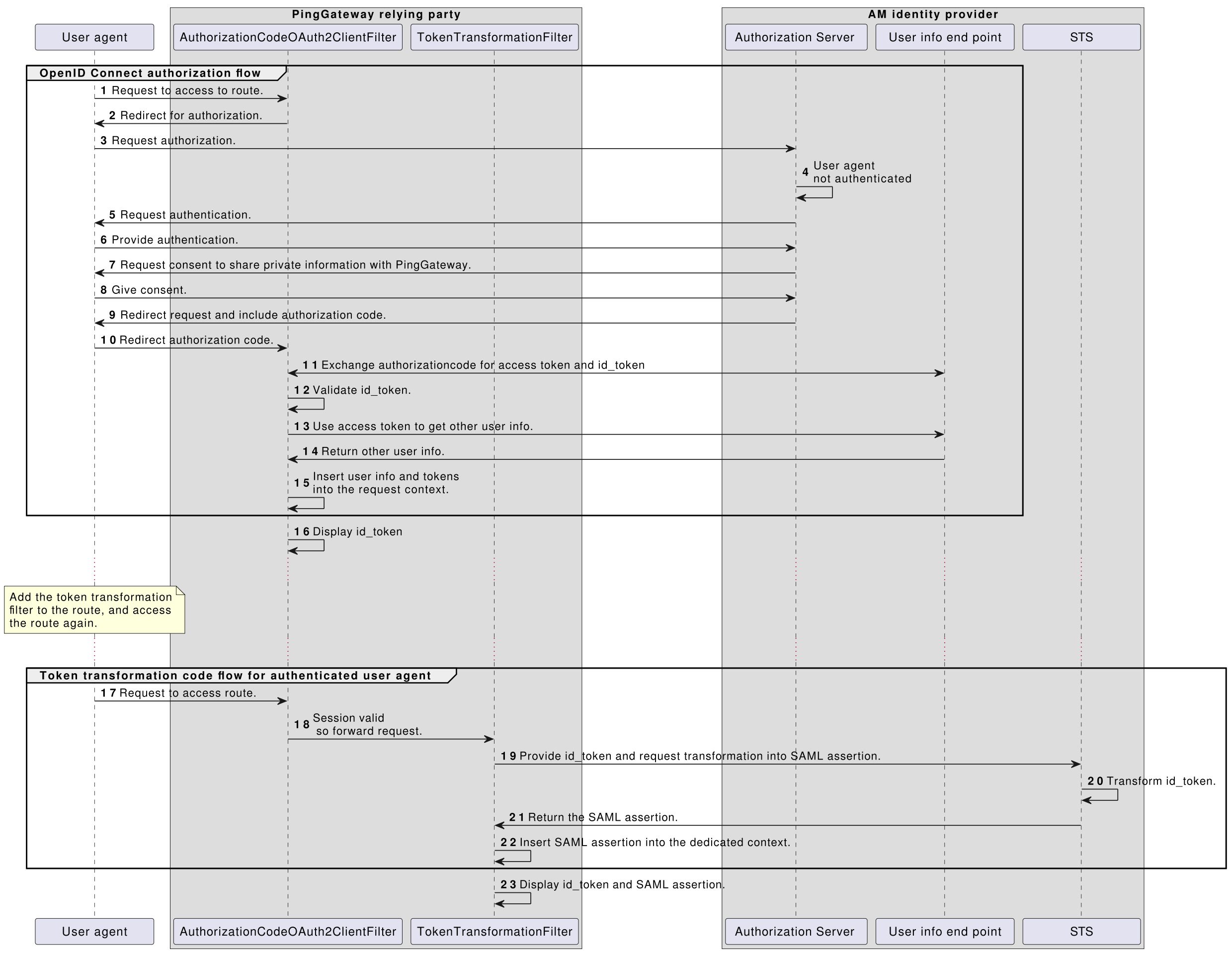
Many enterprises use existing or legacy, SAML 2.0-based SSO, but many mobile and social applications are managed by OpenID Connect. Use the PingGateway TokenTransformationFilter to bridge the gap between OpenID Connect and SAML 2.0 frameworks.

The following figure illustrates the data flow:



1. A user tries to access a protected resource.
2. If the user isn't authenticated, the `AuthorizationCodeOAuth2ClientFilter` redirects the request to AM. After authentication, AM asks for the user's consent to give PingGateway access to private information.
3. If the user consents, AM returns an `id_token` to the `AuthorizationCodeOAuth2ClientFilter`. The filter opens the `id_token` JWT and makes it available in `attributes.openid.id_token` and `attributes.openid.id_token_claims` for downstream filters.
4. The `TokenTransformationFilter` calls the AM STS to transform the `id_token` into a SAML 2.0 assertion.
5. The STS validates the signature, decodes the payload, and verifies the user issued the transaction. The STS then issues a SAML assertion to PingGateway on behalf of the user.
6. The `TokenTransformationFilter` puts the SAML assertion in `${contexts.sts.issuedToken}`.

The following sequence diagram shows a more detailed view of the flow:


PingGateway relying party
AM identity provider
User agent
AuthorizationCodeOAuth2ClientFilter
TokenTransformationFilter
Authorization Server
User info end point
STS
OpenID Connect authorization flow
1 Request to access to route.
2 Redirect for authorization.
3 Request authorization.
4 User agent not authenticated
5 Request authentication.
6 Provide authentication.
7 Request consent to share private information with PingGateway.
8 Give consent.
9 Redirect request and include authorization code.
10 Redirect authorization code.
11 Exchange authorizationcode for access token and id_token
12 Validate id_token.
13 Use access token to get other user info.
14 Return other user info.
15 Insert user info and tokens into the request context.
16 Display id_token
Add the token transformation filter to the route, and access the route again.
Token transformation code flow for authenticated user agent
17 Request to access route.
18 Session valid so forward request.
19 Provide id_token and request transformation into SAML assertion.
20 Transform id_token.
21 Return the SAML assertion.
22 Insert SAML assertion into the dedicated context.
23 Display id_token and SAML assertion.

Update AM settings

Before you start, set up and test the example in AM as OIDC provider.

The following example uses an authentication tree for the Security Token Service (STS) to validate the incoming ID token and map its subject to an account.

Token signing

Change the token signature algorithm in the `oidc_client` profile.

1. In the AM admin UI, select **Applications** > **OAuth 2.0** > **Clients** > `oidc_client` .
2. Under **Signing and Encryption**, set **ID Token Signing Algorithm** to `HS256` and click **Save Changes**.

Client secret

Store the client secret in a secret store.

1. Go to http://am.example.com:8088/openam/encode.jsp and use the page to encode the client secret, `password` .
2. Write the result to a AM default password secret store file named `clientpass` :

```
$ echo -n <encoded-client-secret> >
/path/to/openam/security/secrets/encrypted/clientpass
```

TIP

In production, use your own secret store.

ID token normalization script

Create a script to normalize the ID token for the STS.

1. Select **</> Scripts** > **+ New Script**.
2. Create a script with the following settings:
   - **Name**: `Normalize id_token`
   - **Script Type**: `Social Identity Provider Profile Transformation`
   - **Language**: `JavaScript`
   - **Evaluator Version**: `Legacy`
3. In the script editor, add the following JavaScript and click **Save Changes**:

```
// The ID token "subname" claim holds the account username:
(function () {
    var fr = JavaImporter( org.forgerock.json.JsonValue);
    var identity = fr.JsonValue.json(fr.JsonValue.object());
    identity.put('userName', jwtClaims.get('subname'));
    return identity;
}());
```

Account mapping script

Create a script to map the normalized ID token to the AM account username.

1. Select **</> Scripts** > **+ New Script**.
2. Create a script with the following settings:
   - **Name**: `Add username to shared state`
   - **Script Type**: `Decision node script for authentication trees`
   - **Evaluator Version**: `Next Generation`
3. In the script editor, add the following JavaScript and click **Save Changes**:

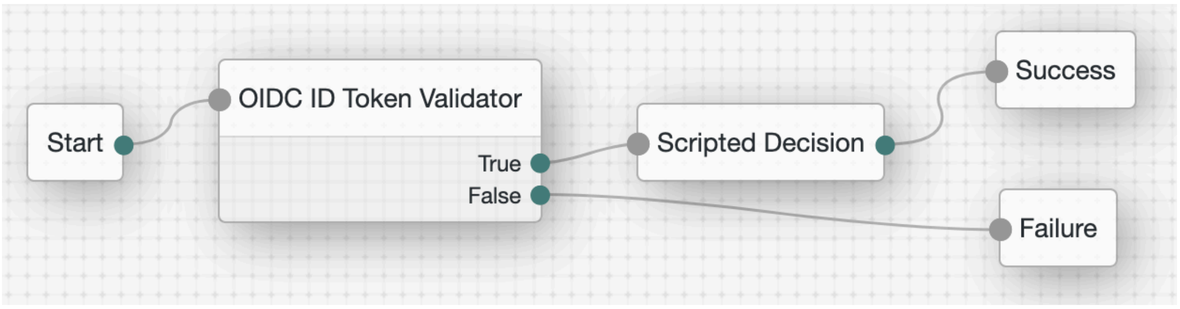
```
// Get the username from the identity returned by the previous
script:
```

```
var attributes = nodeState.get("lookupAttributes");
var userName = attributes.get("userName");
nodeState.putShared("username", userName);
action.goTo('true');
```

Tree for STS

Add a tree to validate the incoming ID token and map its subject to the AM demo user account.

1. Select **Authentication** > **Trees** > **+ Create Tree**.
2. Name the new tree `TransformIdToken` .
3. Add nodes to the tree as in the following image:



   - The OIDC ID Token Validator node lets the STS validate the incoming ID token.

     The node has the following non-default settings:

     - **OpenID Connect Validation Type**: `Client Secret`
     - **OpenID Connect Validation Value**: `password` (Although required, the value isn't used.)
     - **Client Secret Label**: `clientpass`
     - **Token Issuer**: `http://am.example.com:8088/openam/oauth2`
     - **Audience name**: `oidc_client`
     - **Authorized parties**: `oidc_client`
     - **Transformation Script**: `Normalize id_token`

   - The Scripted Decision node maps the ID token subject claim to an AM username.

     The node has the following non-default settings:

     - **Script**: `Add username to shared state`
     - **Outcomes**: `true`

4. Click **Save**.

Add REST STS

Add a REST STS instance to transform the ID token to a SAML v2.0 assertion.

1. Click **STS** > **+ Add Rest STS**, add the following non-default settings:

   Deployment URL Element

   `openig` (must match the TokenTransformationFilter `"instance"`)

   Deployment

   - **Authentication Target Mappings**: replace the existing `OPENIDCONNECT|…` value with:

     ```
     OPENIDCONNECT|service|TransformIdToken|oidc_id_token_auth_tar
     get_header_key=oidc_id_token
     ```

   SAML2 Token

   > NOTE
   >
   > For STS, it isn't necessary to create a SAML SP configuration in AM.

   - **SAML2 issuer Id**: `OpenAM`
   - **Service Provider Entity Id**: `openig_sp`
   - **NameIdFormat**: `urn:oasis:names:tc:SAML:2.0:nameid-format:transient`

   OpenID Connect Token

   - **OpenID Connect Token Provider Issuer Id**: `oidc`
   - **Token signature algorithm**: `HMAC SHA 256`
   - **Client Secret**: `password`
   - **Issued Tokens Audience**: `oidc_client`

2. Click **Create**.
3. Under **SAML 2 Token** on the new STS instance, add the following **Attribute Mappings**:
   - **Key**: `userName` , **Value**: `uid`
   - **Key**: `password` , **Value**: `mail`
4. Click **Save Changes**.

Configure PingGateway

1. Set environment variables for `oidc_client` and `ig_agent` , then restart PingGateway:

```
$ export OIDC_SECRET_ID='cGFzc3dvcmQ='
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

2. Add the following route to PingGateway:

Linux Windows

```
$HOME/.openig/config/routes/50-idtoken.json
```

```
{
  "name": "50-idtoken",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/id_token')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AuthenticatedRegistrationHandler-1",
      "type": "Chain",
      "config": {
        "filters": [
          {
            "name": "ClientSecretBasicAuthenticationFilter-1",
            "type": "ClientSecretBasicAuthenticationFilter",
            "config": {
              "clientId": "oidc_client",
              "clientSecretId": "oidc.secret.id",
              "secretsProvider": "SystemAndEnvSecretStore-1"
            }
          }
        ],
        "handler": "ForgeRockClientHandler"
      }
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "AuthorizationCodeOAuth2ClientFilter-1",
          "type": "AuthorizationCodeOAuth2ClientFilter",
          "config": {
            "clientEndpoint": "/home/id_token",
            "failureHandler": {
              "type": "StaticResponseHandler",
              "config": {
                "status": 500,
                "headers": {
                  "Content-Type": [
                    "text/plain"
                  ]
                },
                "entity": "An error occurred during the OAuth2
setup."
              }
            },
            "registrations": [
              {
                "name": "oidc-user-info-client",
                "type": "ClientRegistration",
                "config": {
                  "clientId": "oidc_client",
                  "issuer": {
                    "name": "Issuer",
                    "type": "Issuer",
                    "config": {
                      "wellKnownEndpoint":
"http://am.example.com:8088/openam/oauth2/.well-known/openid-
configuration"
                    }
                  },
                  "clientSecretIdUsage":
"ID_TOKEN_VALIDATION_ONLY",
                  "secretsProvider": "SystemAndEnvSecretStore-
1",
                  "clientSecretId": "oidc.secret.id",
                  "scopes": [
                    "openid",
                    "profile",
                    "email"
                  ],
                  "authenticatedRegistrationHandler":
"AuthenticatedRegistrationHandler-1"
                }
              }
            ],
            "requireHttps": false,
            "cacheExpiration": "disabled"
          }
        },
        {
          "name": "TokenTransformationFilter-1",
          "type": "TokenTransformationFilter",
          "config": {
            "idToken": "${attributes.openid.id_token}",
            "instance": "openig",
            "amService": "AmService-1"
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/plain; charset=UTF-8" ]
          },
          "entity": "
{\"id_token\":\n\"${attributes.openid.id_token}\"}
\n\n\n{\"saml_assertions\":\n\"${contexts.sts.issuedToken}\"}"
        }
      }
    }
  }
}
```

Learn how to set up the PingGateway route in Studio in Token transformation in Structured Editor.

Notice the following features of the route:

- The route matches requests to `/home/id_token`.
- The `AmService` in the heap is used for authentication and REST STS requests.
- The `AuthorizationCodeOAuth2ClientFilter` enables PingGateway to act as an OIDC relying party:
    - The client endpoint is `/home/id_token`, so the service URIs for this filter in PingGateway are `/home/id_token/login`, `/home/id_token/logout`, and `/home/id_token/callback`.
    - For convenience in this test, `requireHttps` is false.
    - The target for storing authorization state information is `${attributes.openid}`. Later filters and handlers can find access tokens and user information at this target.
- The `ClientRegistration` holds configuration provided in AM as OIDC provider. PingGateway uses it to connect to AM and verify the ID token signature with the client secret.
- The `TokenTransformationFilter` transforms an ID token into a SAML assertion:
    - The `id_token` parameter defines where this filter gets the ID token.

      The `TokenTransformationFilter` puts the result in `${contexts.sts.issuedToken}`.
- On success, a `StaticResponseHandler` displays the ID token and SAML assertion.

Validation

1. In your browser’s privacy or incognito mode, go to https://ig.example.com:8443/home/id_token.

   AM displays the login screen.
2. Log in to AM as username `demo`, password `Ch4ng31t`.

   AM prompts you to allow access to your account information.
3. Select **Allow**.

   The `StaticResponseHandler` displays the ID token and SAML assertion:

```
{"id_token":
"<id-token>"}
```

```
{"saml_assertions":
"…<saml:Subject><saml:NameID…>demo</saml:NameID>…"}
```

TIP

If a request returns an HTTP 414 URI Too Long error, refer to URI Too Long error.

OAuth 2.0 token exchange

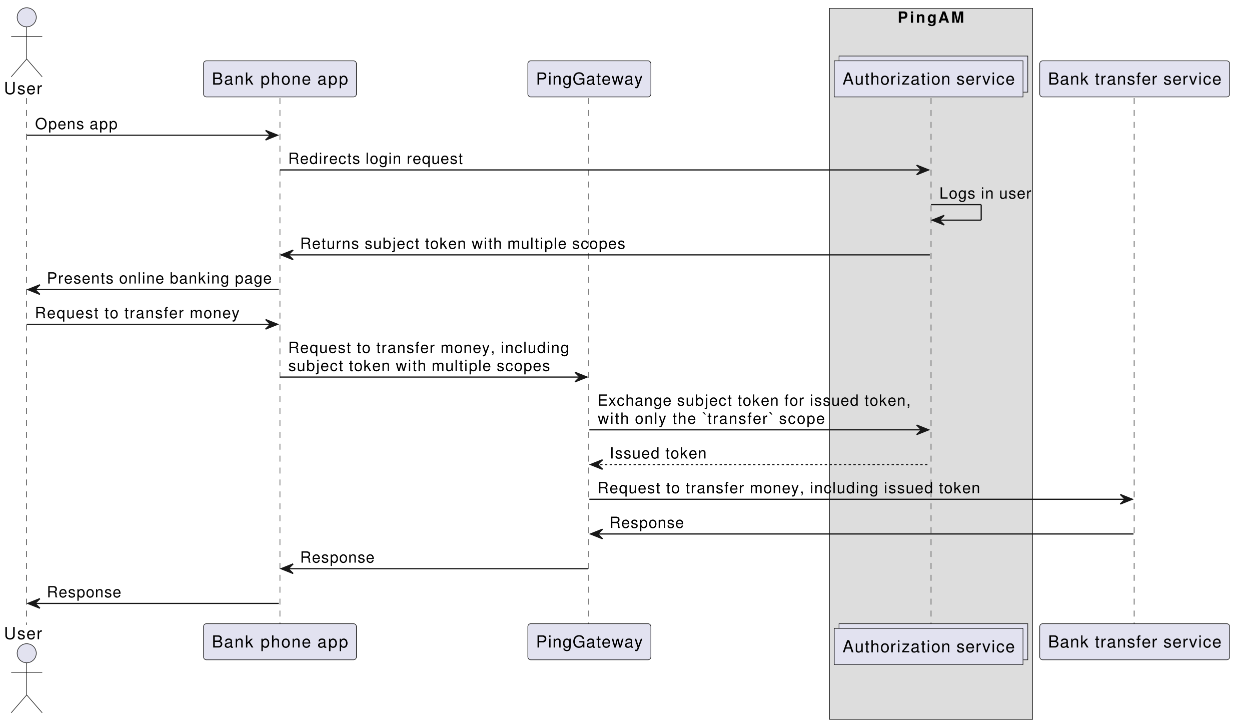
This page describes how to exchange an OAuth 2.0 access token for another access token with AM as an Authorization Server. Other authorization providers can be used instead of AM.

Token exchange requires a *subject token* and provides an *issued token*. The subject token is the original access token, obtained using the OAuth 2.0/OpenID Connect flow. The issued token is provided in exchange for the subject token.

The token exchange can be conducted only by an OAuth 2.0 client that "may act" on the subject token, as configured in the authorization service.

This example is a typical scenario for *token impersonation*. For more information, refer to Token exchange in the AM documentation.

The following sequence diagram shows the flow of information during token exchange between PingGateway and AM:


PingAM
User
Bank phone app
PingGateway
Authorization service
Bank transfer service
Opens app
Redirects login request
Logs in user
Returns subject token with multiple scopes
Presents online banking page
Request to transfer money
Request to transfer money, including subject token with multiple scopes
Exchange subject token for issued token, with only the `transfer` scope
Issued token
Request to transfer money, including issued token
Response
Response
Response

IMPORTANT

IMPORTANT

This procedure uses the *Resource Owner Password Credentials* grant type. As suggested in The OAuth 2.0 Authorization Framework, use other grant types whenever possible.

Before you begin, prepare AM, PingGateway, and the sample application. Learn more in the example installation for this guide.

1. Set up AM:
   a. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:
      - `https://ig.example.com:8443/*`
      - `https://ig.example.com:8443/*?*`
   b. Register a PingGateway agent with the following values, as described in Register a PingGateway agent in AM:
      - **Agent ID**: `ig_agent`
      - **Password**: `password`
      - **Token Introspection**: `Realm Only`

      IMPORTANT

      Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.
   c. (Optional) Authenticate the agent to AM as described in Authenticate a PingGateway agent to AM.

      IMPORTANT

      PingGateway agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.
   d. Select **Services** > **Add a Service**, and add an **OAuth2 Provider** service with the following values:
      - **OAuth2 Access Token May Act Script** : `OAuth2 May Act Script`
      - **OAuth2 ID Token May Act Script** : `OAuth2 May Act Script`
   e. Select **</> Scripts#** > **OAuth2 May Act Script**, and replace the example script with the following script:

```
import org.forgerock.json.JsonValue
token.setMayAct(
    JsonValue.json(JsonValue.object(
        JsonValue.field("client_id",
"serviceConfidentialClient"))))
```

This script adds a `may_act` claim to the token, indicating that the OAuth 2.0 client, `serviceConfidentialClient`, may act to exchange the subject token in the impersonation use case.

f. Add an OAuth 2.0 Client to request OAuth 2.0 access tokens:

   i. Select **Applications** > **OAuth 2.0** > **Clients**, and add a client with the following values:

      - **Client ID** : `client-application`
      - **Client secret** : `password`
      - **Scope(s)** : `mail` , `employeenumber`

   ii. On the **Advanced** tab, select **Grant Types** : `Resource Owner Password Credentials` .

g. Add an OAuth 2.0 client to perform the token exchange:

   i. Select **Applications** > **OAuth 2.0** > **Clients**, and add a client with the following values:

      - **Client ID** : `serviceConfidentialClient`
      - **Client secret** : `password`
      - **Scope(s)** : `mail` , `employeenumber`

   ii. On the **Advanced** tab, select:

      - **Grant Types** : `Token Exchange`
      - **Token Endpoint Authentication Methods** : `client_secret_post`

2. Set up PingGateway:

   a. Set up PingGateway for HTTPS, as described in Configure PingGateway for TLS (server-side).

   b. Set the following environment variables for the serviceConfidentialClient password:

```
$ export CLIENT_SECRET_ID='cGFzc3dvcmQ='
```

   c. Set an environment variable for the PingGateway agent password, and then restart PingGateway:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

d. Add the following route to PingGateway to exchange the access token:

Linux | Windows

```
$HOME/.openig/config/routes/token-exchange.json
```

```
{
  "name": "token-exchange",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/token-
exchange')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    },
    {
      "name": "ExchangeHandler",
      "type": "Chain",
      "capture": "all",
      "config": {
        "handler": "ForgeRockClientHandler",
        "filters": [
          {
            "type":
"ClientSecretBasicAuthenticationFilter",
            "config": {
              "clientId": "serviceConfidentialClient",
              "clientSecretId": "client.secret.id",
              "secretsProvider" :
"SystemAndEnvSecretStore-1"
```

```
            }
          }
        ]
      }
    },
    {
      "name": "ExchangeFailureHandler",
      "type": "StaticResponseHandler",
      "capture": "all",
      "config": {
        "status": 400,
        "entity": "${contexts.oauth2Failure.error}:
${contexts.oauth2Failure.description}",
        "headers": {
          "Content-Type": [
            "application/json"
          ]
        }
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "oauth2TokenExchangeFilter",
          "type": "OAuth2TokenExchangeFilter",
          "config": {
            "amService": "AmService-1",
            "endpointHandler": "ExchangeHandler",
            "subjectToken": "#
{request.entity.form['subject_token'][0]}",
            "scopes": ["mail"],
            "failureHandler": "ExchangeFailureHandler"
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "content-type": [
              "application/json"
            ]
          },
          "entity": "{\"access_token\":
\"${contexts.oauth2TokenExchange.issuedToken}\",
\"issued_token_type\":
\"${contexts.oauth2TokenExchange.issuedTokenType}\"}"
        }
      }
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/token-exchange`
- PingGateway reads the `subjectToken` from the request entity.
- The StaticResponseHandler returns an issued token.

3. Test the setup:

   a. In a terminal window, use a **`curl`** command similar to the following to retrieve an access token, which is the *subject token*:

```
$ subjecttoken=$(curl -s \
--user "client-application:password" \
--data
"grant_type=password&username=demo&password=Ch4ng31t&scope
=mail%20employeenumber" \
http://am.example.com:8088/openam/oauth2/access_token | jq
-r ".access_token") \
&& echo $subjecttoken

hc-...c6A
```

   b. Introspect the subject token at the AM introspection endpoint:

```
$ curl --location \
--request POST
'http://am.example.com:8088/openam/oauth2/realms/root/intr
ospect' \
--header 'Content-Type: application/x-www-form-urlencoded'
\
--data-urlencode "token=${subjecttoken}" \
--data-urlencode 'client_id=client-application' \
--data-urlencode 'client_secret=password'
```

```
Decoded access_token: {
  "active": true,
  "scope": "employeenumber mail",
  "realm": "/",
  "client_id": "client-application",
  "user_id": "demo",
  "username": "demo",
  "token_type": "Bearer",
  "exp": 1626796888,
  "sub": "(usr!demo)",
  "subname": "demo",
  "iss": "http://am.example.com:8088/openam/oauth2",
  "auth_level": 0,
  "authGrantId": "W-j...E1E",
  "may_act": {
    "client_id": "serviceConfidentialClient"
  },
  "auditTrackingId": "4be...169"
}
```

Note that in the subject token, the `client_id` is `client-application`, and the scopes are `employeenumber` and `mail`. The `may_act` claim indicates that `serviceConfidentialClient` is authorized to exchange this token.

c. Exchange the subject token for an issued token:

```
$ issuedtoken=$(curl \
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--location \
--request POST 'https://ig.example.com:8443/token-
exchange' \
--header 'Content-Type: application/x-www-form-urlencoded'
\
--data "subject_token=${subjecttoken}" | jq -r
".access_token") \
&& echo $issuedtoken

F8e...Q3E
```

d. Introspect the issued token at the AM introspection endpoint:

```
$ curl --location \
--request POST
'http://am.example.com:8088/openam/oauth2/realms/root/intr
ospect' \
--header 'Content-Type: application/x-www-form-urlencoded'
\
--data-urlencode "token=${issuedtoken}" \
--data-urlencode 'client_id=serviceConfidentialClient' \
--data-urlencode 'client_secret=password'

{
  "active": true,
  "scope": "mail",
  "realm": "/",
  "client_id": "serviceConfidentialClient",
  "user_id": "demo",
  "username": "demo",
  "token_type": "Bearer",
  "exp": 1629200490,
  "sub": "(usr!demo)",
  "subname": "demo",
  "iss": "http://am.example.com:8088/openam/oauth2",
  "auth_level": 0,
  "authGrantId": "aYK...EPA",
  "may_act": {
    "client_id": "serviceConfidentialClient"
  },
  "auditTrackingId": "814...367"
}
```

Note that in the issued token, the `client_id` is `serviceConfidentialClient`, and the only the scope is `mail`.

Not-enforced URIs

By default, PingGateway routes protect access to resources for all requests matching the route's condition path. For assets that don't need protection, like the welcome page of a website, public images, and favicons, you can avoid enforcing protection.

The following sections show routes that don't enforce authentication for some request URLs or URL patterns.

With a SwitchFilter

Before you start:

- Prepare PingGateway and the sample app as described in Quick install.

- Install and configure AM on http://am.example.com:8088/openam using the default configuration.

1. On your system, add the following data in a comma-separated value file:

   Linux | Windows

   ```
   /tmp/userfile.txt
   ```

   ```
   username,password,fullname,email
   george,C0stanza,George Costanza,george@example.com
   kramer,N3wman12,Kramer,kramer@example.com
   bjensen,H1falutin,Babs Jensen,bjensen@example.com
   demo,Ch4ng31t,Demo User,demo@example.com
   kvaughan,B5ibery12,Kirsten Vaughan,kvaughan@example.com
   scarter,S9rain12,Sam Carter,scarter@example.com
   wolkig,Geh3imnis!,Wilhelm Wolkig,wolkig@example.com
   ```

2. Set up AM:

   a. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:

      - `https://ig.example.com:8443/*`
      - `https://ig.example.com:8443/*?*`

   b. Register a PingGateway agent with the following values, as described in Register a PingGateway agent in AM:

      - **Agent ID**: `ig_agent`
      - **Password**: `password`

      IMPORTANT

      Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

   c. (Optional) Authenticate the agent to AM as described in Authenticate a PingGateway agent to AM.

      IMPORTANT

      PingGateway agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

3. Set up PingGateway:

a. Set an environment variable for the PingGateway agent password, and then restart PingGateway:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

b. Add the following route to PingGateway to serve the sample application .css and other static resources:

Linux Windows

```
$HOME/.openig/config/routes/00-static-
resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or
matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

c. Add the following route to PingGateway:

Linux Windows

```
$HOME/.openig/config/routes/not-enforced-
switch.json
```

```
{
  "properties": {
    "notEnforcedPathPatterns":
"^/home|^/favicon.ico|^/css"
  },
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "name": "not-enforced-switch",
  "condition": "${find(request.uri.path, '^/')}",
  "baseURI": "http://app.example.com:8081",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SwitchFilter-1",
          "type": "SwitchFilter",
          "config": {
            "onRequest": [{
              "condition": "${find(request.uri.path, '&
{notEnforcedPathPatterns}')}",
              "handler": "ReverseProxyHandler"
            }]
          }
        },
        {
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        },
        {
          "type": "PasswordReplayFilter",
          "config": {
            "loginPage": "${true}",
            "credentials": {
              "type": "FileAttributesFilter",
              "config": {
                "file": "/tmp/userfile.txt",
                "key": "email",
                "value":
"${contexts.ssoToken.info.uid}@example.com"
              }
            },
            "request": {
              "method": "POST",
              "uri":
"http://app.example.com:8081/login",
              "form": {
                "username": [

"${contexts.fileAttributes.record.username}"
                ],
                "password": [

"${contexts.fileAttributes.record.password}"
                ]
              }
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route:

- The route condition is `/`, so the route matches all requests.
- The SwitchFilter passes requests on the path `^/home`, `^/favicon.ico`, and `^/css` directly to the ReverseProxyHandler. All other requests continue the along the chain to the SingleSignOnFilter.
- If the request doesn't have a valid AM session cookie, the SingleSignOnFilter redirects the request to AM for authentication. The SingleSignOnFilter stores the cookie value in an `SsoTokenContext`.
- Because the PasswordReplayFilter detects that the response is a login page, it uses the FileAttributesFilter to replay the password, and logs the request into the sample application.

4. Test the setup:

    a. If you are signed in to AM, sign off and clear any cookies.

b. Access the route on the not-enforced URL http://ig.example.com:8080/home. The home page of the sample app is displayed without authentication.

c. Access the route on the enforced URL http://ig.example.com:8080/profile. The SingleSignOnFilter redirects the request to AM for authentication.

d. Sign on to AM as user `demo`, password `Ch4ng31t`. The PasswordReplayFilter replays the credentials for the demo user. The request is passed to the sample app's profile page for the demo user.

With a DispatchHandler

To use a DispatchHandler for not-enforced URIs, replace the route in With a SwitchFilter with the following route. If the request is on the path `^/home`, `^/favicon.ico`, or `^/css`, the DispatchHandler sends it directly to the ReverseProxyHandler, without authentication. It passes all other requests into the Chain for authentication.

```
{
  "properties": {
    "notEnforcedPathPatterns": "^/home|^/favicon.ico|^/css"
  },
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "name": "not-enforced-dispatch",
  "condition": "${find(request.uri.path, '^/')}",
  "baseURI": "http://app.example.com:8081",
  "handler": {
    "type": "DispatchHandler",
    "config": {
      "bindings": [
        {
          "condition": "${find(request.uri.path, '&
{notEnforcedPathPatterns}')}",
          "handler": "ReverseProxyHandler"
        },
        {
          "handler": {
            "type": "Chain",
            "config": {
              "filters": [
                {
                  "type": "SingleSignOnFilter",
                  "config": {
                    "amService": "AmService-1"
                  }
                },
                {
                  "type": "PasswordReplayFilter",
                  "config": {
                    "loginPage": "${true}",
                    "credentials": {
                      "type": "FileAttributesFilter",
                      "config": {
                        "file": "/tmp/userfile.txt",
                        "key": "email",
                        "value":
"${contexts.ssoToken.info.uid}@example.com"
                      }
                    },
                    "request": {
                      "method": "POST",
                      "uri": "http://app.example.com:8081/login",
                      "form": {
                        "username": [

"${contexts.fileAttributes.record.username}"
                        ],
                        "password": [

"${contexts.fileAttributes.record.password}"
                        ]
                      }
                    }
                  }
```

```
                    }
                  ],
                  "handler": "ReverseProxyHandler"
                }
              }
            }
          ]
        }
      }
    }
```

POST data preservation

The DataPreservationFilter triggers POST data preservation when an unauthenticated client posts HTML form data to a protected resource.

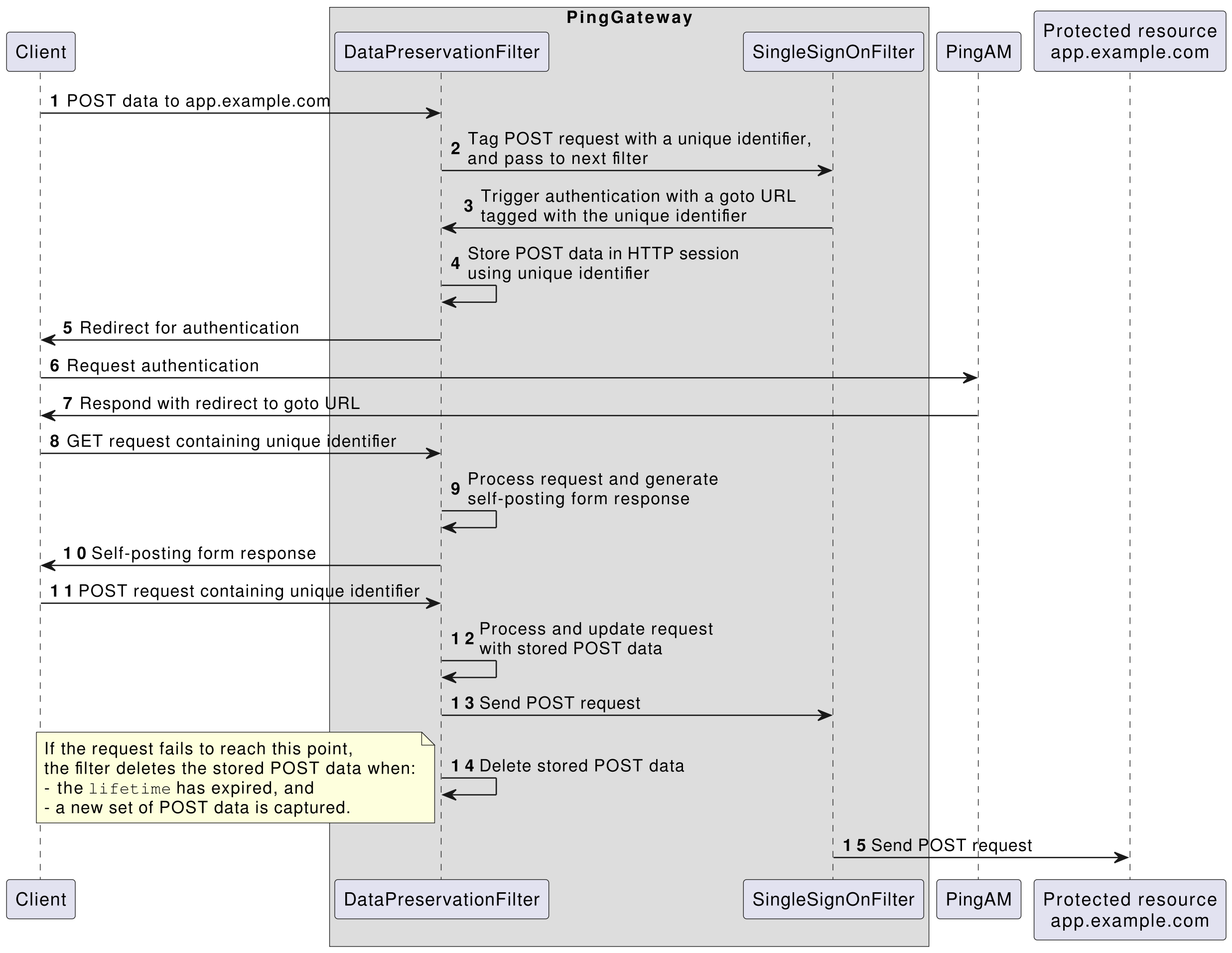
When an authentication redirect is triggered, the filter stores the data in the HTTP session, and redirects the client for authentication. After authentication, the filter generates an empty self-submitting form POST to emulate the original POST. It then replays the stored data into the request before passing it along the chain.

The data can be any POST content, such as HTML form data or a file upload.

Consider the following points for POST data preservation:

- The size of the POST data is important because the data is stored in the HTTP session.
- Stateless sessions store form content in encrypted JWT session cookies. To prevent requests from being rejected because the HTTP headers are too long, configure `connectors:maxTotalHeadersSize` in admin.json.
- Sticky sessions may be required for deployments with stateful sessions, and multiple PingGateway instances.

The following image shows a simplified data flow for POST data preservation:



PingGateway
Client
DataPreservationFilter
SingleSignOnFilter
PingAM
Protected resource app.example.com
1 POST data to app.example.com
2 Tag POST request with a unique identifier, and pass to next filter
3 Trigger authentication with a goto URL tagged with the unique identifier
4 Store POST data in HTTP session using unique identifier
5 Redirect for authentication
6 Request authentication
7 Respond with redirect to goto URL
8 GET request containing unique identifier
9 Process request and generate self-posting form response
10 Self-posting form response
11 POST request containing unique identifier
12 Process and update request with stored POST data
13 Send POST request
If the request fails to reach this point, the filter deletes the stored POST data when: - the lifetime has expired, and - a new set of POST data is captured.
14 Delete stored POST data
15 Send POST request

1. An unauthenticated client requests a POST to a protected resource.

2. The DataPreservationFilter tags the the request with a unique identifier, and passes it along the chain. The next filter should be an authentication filter such as a SingleSignOnFilter.

3. The next filter triggers the authentication, and includes a goto URL tagged with the unique identifier from the previous step.

4-5. The DataPreservationFilter stores the POST data in the HTTP session, and redirects the request for authentication. The POST data is identified by the unique identifier.

6-7. The client authenticates with AM, and AM provides an authentication response to the goto URL.

8. The authenticated client sends a GET request containing the unique identifier.

9-10. The DataPreservationFilter validates the unique identifier, and generates a self-posting form response for the client.

The presence of the unique identifier in the goto URL ensures that requests at the URL can be individually identified. Additionally, it is more difficult to hijack user data, because there is little chance of guessing the code within the login window.

If the identifier is not validated, PingGateway denies the request.

11. The client resends the POST request, including the identifier.

12-13. The DataPreservationFilter updates the request with the POST data, and sends it along the chain.

Preserve POST data during CDSSO

Before you start, set up and test the example in Cross-domain single sign-on (CDSSO).

1. In PingGateway, replace `cdsso.json` with the following route:

   Linux | Windows

   ```
   $HOME/.openig/config/routes/pdp.json
   ```

   ```
   {
     "name": "pdp",
     "baseURI": "http://app.example.com:8081",
     "condition": "${find(request.uri.path, '^/home/cdsso')}",
     "heap": [
       {
         "name": "SystemAndEnvSecretStore-1",
         "type": "SystemAndEnvSecretStore"
       },
       {
         "name": "AmService-1",
         "type": "AmService",
         "config": {
           "url": "http://am.example.com:8088/openam",
           "realm": "/",
           "agent": {
             "username": "ig_agent_cdsso",
             "passwordSecretId": "agent.secret.id"
           },
           "secretsProvider": "SystemAndEnvSecretStore-1",
           "sessionCache": {
             "enabled": false
           }
         }
       }
     ],
     "handler": {
       "type": "Chain",
       "config": {
   ```

```
      "filters": [
        {
          "name": "DataPreservationFilter",
          "type": "DataPreservationFilter"
        },
        {
          "name": "CrossDomainSingleSignOnFilter-1",
          "type": "CrossDomainSingleSignOnFilter",
          "config": {
            "redirectEndpoint": "/home/cdsso/redirect",
            "authCookie": {
              "path": "/home",
              "name": "ig-token-cookie"
            },
            "amService": "AmService-1",
            "logoutExpression": "${find(request.uri.query,
'logOff=true')}",
            "defaultLogoutLandingPage": "/form"
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [
              "text/html; charset=UTF-8"
            ]
          },
          "entity": [
            "<html>",
            "  <body>",
            "    <h1>Request Information</h1>",
            "      <p>Request method: #{request.method}",
            "      <p>Request URI: #{request.uri}",
            "      <p>Query string: #{request.queryParams}",
            "      <p>Form: #{request.entity.form}",
            "      <p>Content length: #
{request.headers['Content-Length'][0]}",
            "      <p>Content type: #
{request.headers['Content-Type'][0]}",
            "  </body>",
            "</html>"
          ]
          }
        }
      }
    }
  }
```

Notice the following differences compared to `cdsso.json`:

- A DataPreservationFilter is positioned in front of the CrossDomainSingleSignOnFilter to manage POST data preservation before authentication.
- The ReverseProxyHandler is replaced by a StaticResponseHandler, which displays the POST data provided in the request.

2. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/form.json
```

```
{
  "condition": "${request.uri.path == '/form'}",
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "headers": {
        "Content-Type": [ "text/html" ]
      },
      "entity" : [
        "<html>",
        "  <body>",
        "    <h1>Test page : POST Data Preservation containing
visible and hidden form elements</h1>",
        "    <form id='testingPDP' enctype='application/x-www-
form-urlencoded' name='test_form'
action='/home/cdsso/pdp.info?foo=bar&baz=pdp' method='post'>",
        "      <input name='email' value='user@example.com'
size='60'>",
        "      <input type='hidden' name='phone' value='555-
123-456'/>",
        "      <input type='hidden' name='manager'
value='Bob'/>",
        "      <input type='hidden' name='dept'
value='Engineering'/>",
        "      <input type='submit' value='Press to demo form
posting' id='form_post_button'/>",
        "    </form>",
        "  </body>",
        "</html>"
      ]
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/home/form`.
- The StaticResponseHandler includes the following entity to present visible and hidden form elements from the original request:

  ```
  <!DOCTYPE html>
  <html>
    <body>
      <h1>Test page : POST Data Preservation containing
  visible and hidden form elements</h1>
      <form
        id='testingPDP'
        enctype='application/x-www-form-urlencoded'
        name='test_form'
        action='/home/cdsso/pdp.info?foo=bar&baz=pdp'
        method='post'>
        <input
          name='email'
          value='user@example.com'
          size='60'>
        <input
          type='hidden'
          name='phone'
          value='555-123-456'/>
        <input
          type='hidden'
          name='manager'
          value='Bob'/>
        <input
          type='hidden'
          name='dept'
          value='Engineering'/>
        <input
  ```

```
                type='submit'
                value='Press to demo form posting'
                id='form_post_button'/>
        </form>
    </body>
</html>
```

3. Test the setup:

   a. In your browser's privacy or incognito mode, go to https://ig.ext.com:8443/form.

      The script in the StaticResponseHandler entity of `form.json` creates a button to demonstrate form posting.

   b. Press the button, and log in to AM as user `demo`, password `Ch4ng31t`.

      When you have authenticated, the script presents the POST data from the original request.

CSRF protection

In a Cross Site Request Forgery (CSRF) attack, a user unknowingly executes a malicious request on a website where they're authenticated. A CSRF attack usually includes a link or script in a web page. When a user accesses the link or script, the page executes an HTTP request on the site where the user is authenticated.

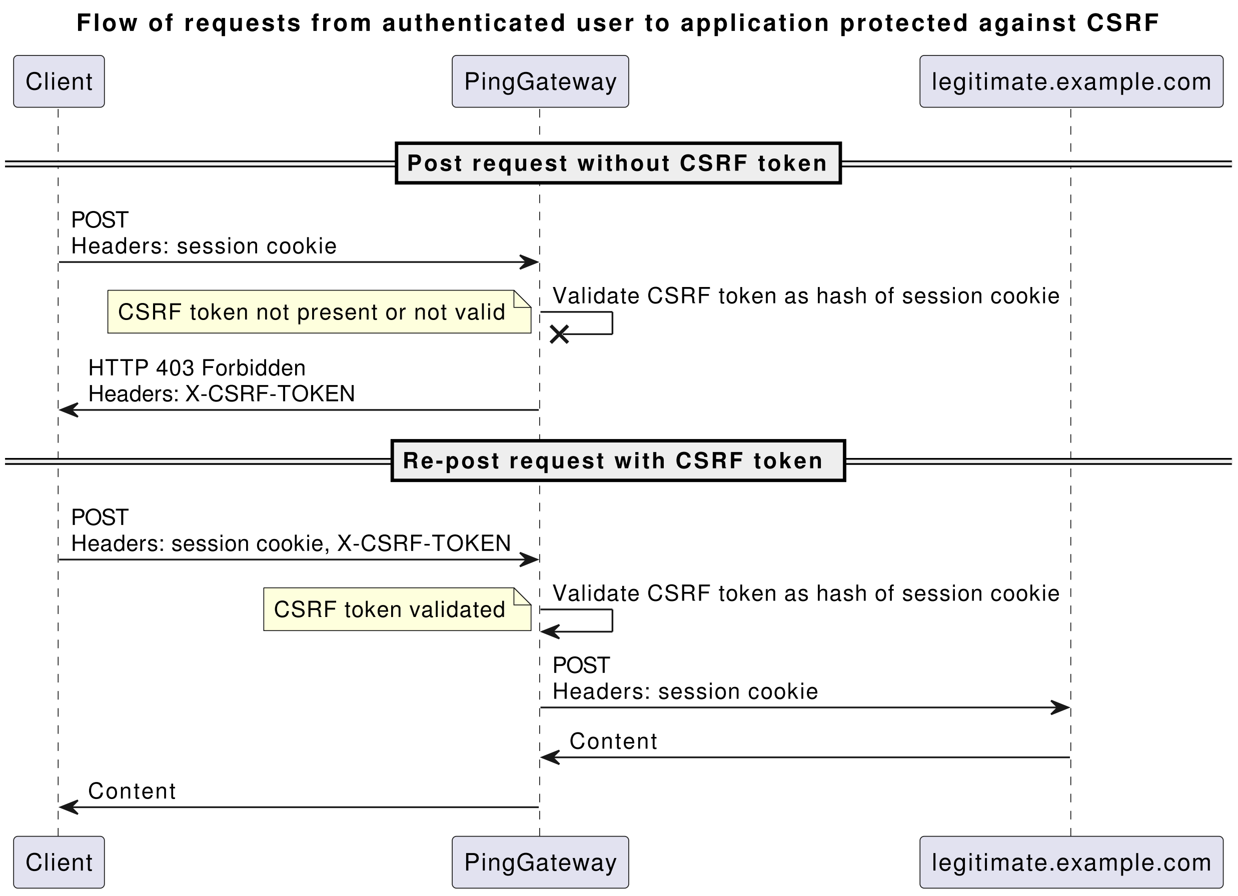
CSRF attacks interact with HTTP requests as follows:

- CSRF attacks can execute POST, PUT, and DELETE requests on the targeted server. For example, a CSRF attack can transfer funds out of a bank account or change a user's password.
- Because of same-origin policy, CSRF attacks **cannot** access any response from the targeted server.

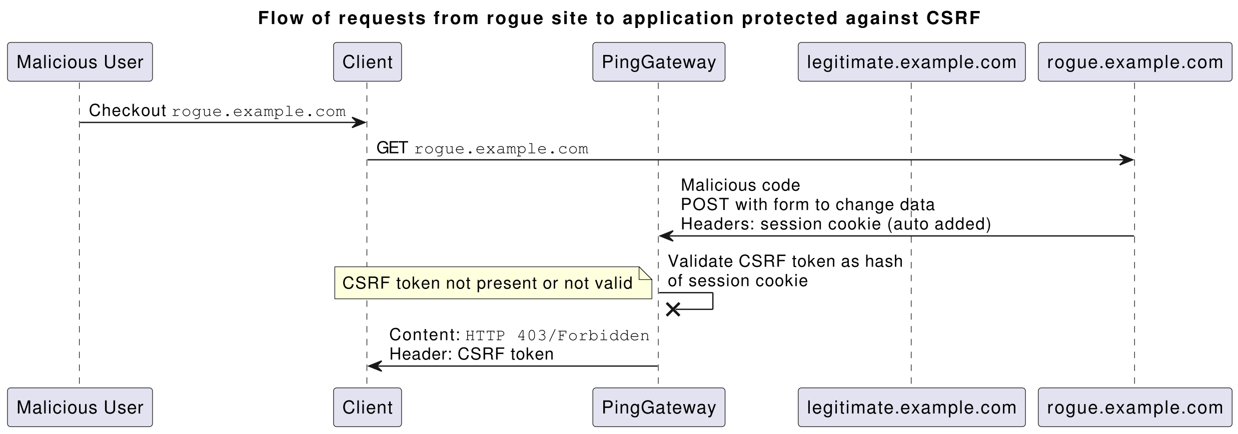
When PingGateway processes POST, PUT, and DELETE requests, it checks a custom HTTP header in the request. If a CSRF token isn't present in the header or not valid, PingGateway rejects the request and returns a valid CSRF token in the response.

Rogue websites that attempt CSRF attacks operate in a different website domain to the targeted website. Because of same-origin policy, rogue websites can't access a response from the targeted website, and can't, therefore, access the response or CSRF token.

The following example shows the data flow when an authenticated user sends a POST request to an application protected against CSRF:


Flow of requests from authenticated user to application protected against CSRF
Client
PingGateway
legitimate.example.com
Post request without CSRF token
POST
Headers: session cookie
CSRF token not present or not valid
Validate CSRF token as hash of session cookie
HTTP 403 Forbidden
Headers: X-CSRF-TOKEN
Re-post request with CSRF token
POST
Headers: session cookie, X-CSRF-TOKEN
CSRF token validated
Validate CSRF token as hash of session cookie
POST
Headers: session cookie
Content
Content

The following example shows the data flow when an authenticated user sends a POST request from a rogue site to an application protected against CSRF:


Flow of requests from rogue site to application protected against CSRF
Malicious User
Client
PingGateway
legitimate.example.com
rogue.example.com
Checkout rogue.example.com
GET rogue.example.com
Malicious code
POST with form to change data
Headers: session cookie (auto added)
Validate CSRF token as hash of session cookie
CSRF token not present or not valid
Content: HTTP 403/Forbidden
Header: CSRF token

1. Set up SSO, so that AM authenticates users to the sample app through PingGateway:
    a. Set up AM and PingGateway as described in Use the default journey.
    b. Remove the condition in `sso.json`, so that the route matches all requests:

```
"condition": "${find(request.uri.path, '^/home/sso')}"
```

2. Test the setup without CSRF protection:
    a. Go to https://ig.example.com:8443/bank/index, and log in to the Sample App Bank through AM, as user `demo`, password `Ch4ng31t`.

b. Send a bank transfer of $10 to Bob, and note that the transfer is successful.

c. Go to http://localhost:8081/bank/attack-autosubmit to simulate a CSRF attack.

> TIP
>
> In deployments that use a different protocol, hostname, or port for PingGateway, append the `igUrl` parameter, as follows:
>
> ```
> http://localhost:8081/bank/attack-autosubmit?
> igUrl=protocol://hostname:port
> ```

When you access this page, a hidden HTML form is automatically submitted to transfer $1000 to the rogue user, using the PingGateway session cookie to authenticate to the bank.

In the bank transaction history, note that $1000 is debited.

3. Test the setup with CSRF protection:

a. In PingGateway, replace `sso.json` with the following route:

```
{
  "name": "Csrf",
  "baseURI": "http://app.example.com:8081",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    },
    {
      "name": "FailureHandler-1",
      "type": "StaticResponseHandler",
      "config": {
        "status": 403,
```

```
          "headers": {
            "Content-Type": [ "text/plain; charset=UTF-8" ]
          },
          "entity": "Request forbidden"
        }
      }
    ],
    "handler": {
      "type": "Chain",
      "config": {
        "filters": [
          {
            "name": "SingleSignOnFilter-1",
            "type": "SingleSignOnFilter",
            "config": {
              "amService": "AmService-1"
            }
          },
          {
            "name": "CsrfFilter-1",
            "type": "CsrfFilter",
            "config": {
              "cookieName": "iPlanetDirectoryPro",
              "failureHandler": "FailureHandler-1"
            }
          }
        ],
        "handler": "ReverseProxyHandler"
      }
    }
  }
```

Notice the following features of the route compared to `sso.json`:

- The CsrfFilter checks the AM session cookie for the `X-CSRF-Token` header. If a CSRF token isn't present in the header or not valid, the filter rejects the request and provides a valid CSRF token in the header.

b. Go to https://ig.example.com:8443/bank/index, and send a bank transfer of $10 to Alice.

   Because there is no CSRF token, PingGateway responds with an HTTP 403, and provides the token.

c. Send the transfer again, and note that because the CSRF token is provided the transfer is successful.

d. Go to http://localhost:8081/bank/attack-autosubmit to automatically send a rogue transfer.

TIP

In deployments that use a different protocol, hostname, or port for PingGateway, append the `igUrl` parameter, as follows:

```
http://localhost:8081/bank/attack-autosubmit?
igUrl=protocol://hostname:port
```

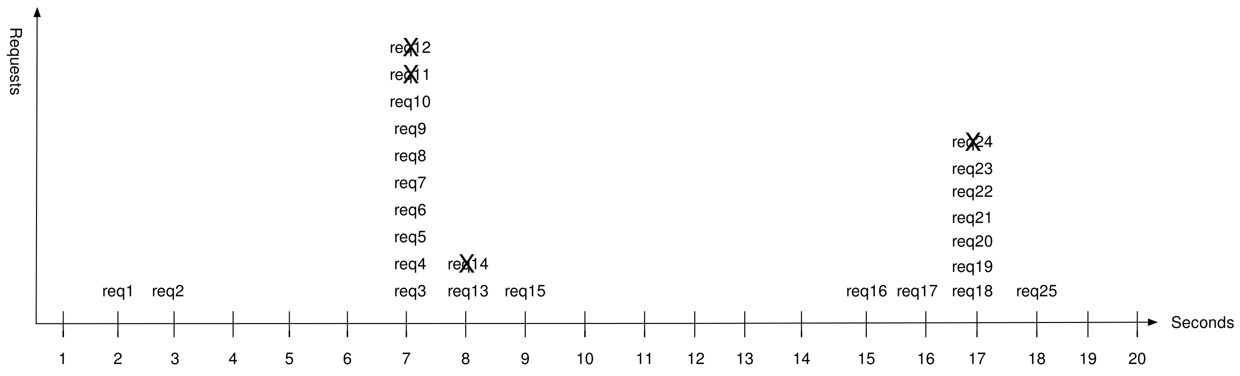
Because there is no CSRF token, PingGateway rejects the request and provides the CSRF token. However, because the rogue site is in a different domain to `ig.example.com` it can't access the CSRF token.

Throttling

To protect applications from being overused by clients, use a throttling filter to limit how many requests can be made in a defined time. The maximum number of requests that are allowed in a defined time is called the *throttling rate*. The following sections describe how to set up simple, mapped, and scriptable throttling filters:

About throttling

The throttling filter uses the token bucket algorithm, allowing some unevenness or bursts in the request flow. The following image shows how PingGateway manages requests for a throttling rate of 10 requests/10 seconds:



- At 7 seconds, 2 requests have previously passed when there is a burst of 9 requests. PingGateway allows 8 requests, but disregards the 9th because the throttling rate for the 10-second throttling period has been reached.
- At 8 and 9 seconds, although 10 requests have already passed in the 10-second throttling period, PingGateway allows 1 request each second.

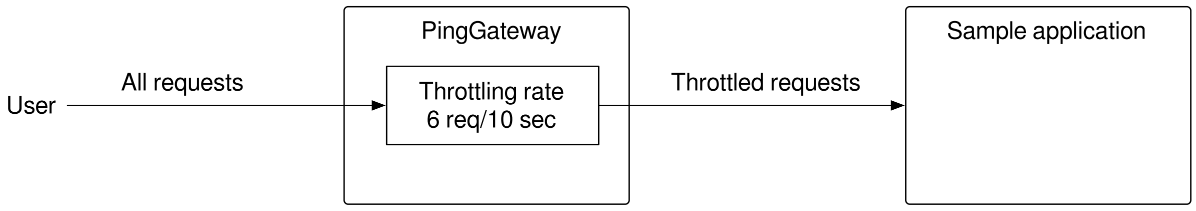
- At 17 seconds, 4 requests have passed in the previous 10-second throttling period, and PingGateway allows another burst of 6 requests.

When the throttling rate is reached, PingGateway issues an HTTP status code 429 `Too Many Requests` and a `Retry-After` header like the following, where the value is the number of seconds to wait before trying the request again:

```
GET https://ig.example.com:8443/home/throttle-scriptable HTTP/2
. . .

HTTP/2 429 Too Many Requests
Retry-After: 10
```

Configure simple throttling

This section describes how to configure a simple throttling filter that applies a throttling rate of 6 requests/10 seconds. When an application is protected by this throttling filter, no more than 6 requests, irrespective of their origin, can access the sample application in a 10 second period.



1. Set up PingGateway for HTTPS, as described in Configure PingGateway for TLS (server-side).
2. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/00-throttle-simple.json
```

```
{
  "name": "00-throttle-simple",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/throttle-
simple')}",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "ThrottlingFilter",
          "name": "ThrottlingFilter-1",
          "config": {
            "requestGroupingPolicy": "",
            "rate": {
              "numberOfRequests": 6,
              "duration": "10 s"
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

For information about how to set up the PingGateway route in Studio, refer to Simple throttling filter in Structured Editor.

Notice the following features of the route:

- The route matches requests to `/home/throttle-simple`.
- The ThrottlingFilter contains a request grouping policy that is blank. This means that all requests are in the same group.
- The rate defines the number of requests allowed to access the sample application in a given time.

3. Test the setup:

   a. In a terminal window, use a **`curl`** command similar to the route in a loop:

   ```
   $ curl -v \
   --cacert /path/to/secrets/ig.example.com-certificate.pem \
   https://ig.example.com:8443/home/throttle-simple/\[01-10\]
   \
   > /tmp/throttle-simple.txt 2>&1
   ```

   b. Search the output file for the result:

   ```
   $ grep "< HTTP/2" /tmp/throttle-simple.txt | sort | uniq -
   c

   6 < HTTP/2 200 OK
   4 < HTTP/2 429 Too Many Requests
   ```

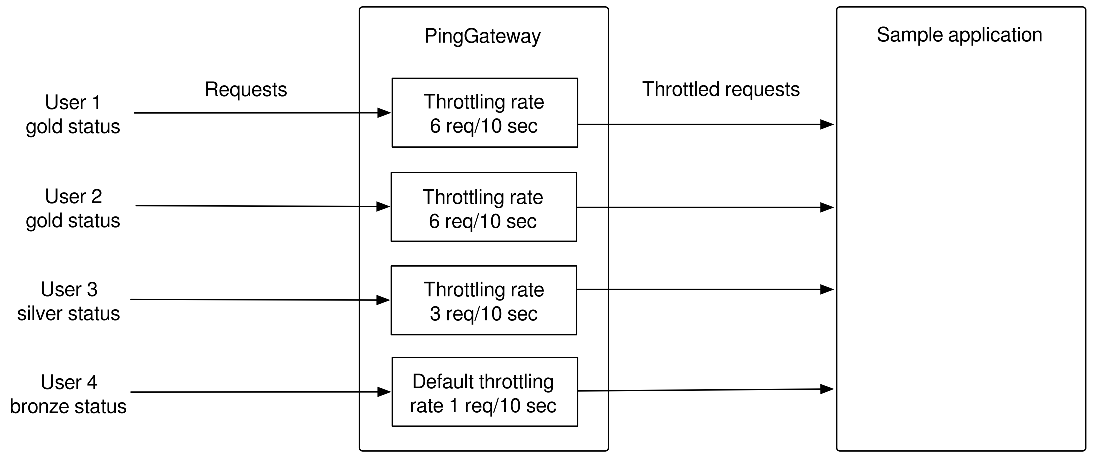
Notice that the first six requests returned a success response, and the following four requests returned an HTTP 429 `Too Many Requests`. This result demonstrates that the throttling filter has allowed only six requests to access the application, and has blocked the other requests.

Configure mapped throttling

This section describes how to configure a mapped throttling policy, where the grouping policy defines criteria to group requests, and the rate policy defines the criteria by which rates are mapped.

The following image illustrates how different throttling rates can be applied to users.

The following image illustrates how each user with a `gold` status has a throttling rate of 6 requests/10 seconds, and each user with a `silver` status has 3 requests/10 seconds. The `bronze` status is not mapped to a throttling rate, and so a user with the `bronze` status has the default rate.



Before you start, set up and test the example in Validate access tokens with introspection.

1. In the AM console, select **</> Scripts** > **OAuth2 Access Token Modification Script** and replace the default script as follows:

```
import org.forgerock.http.protocol.Request
import org.forgerock.http.protocol.Response

def attributes = identity.getAttributes(["mail",
"employeeNumber"].toSet())
accessToken.setField("mail", attributes["mail"][0])
def mail = attributes['mail'][0]
if (mail.endsWith('@example.com')) {
  status = "gold"
} else if (mail.endsWith('@other.com')) {
  status = "silver"
} else {
  status = "bronze"
}
accessToken.setField("status", status)
```

The AM script adds user profile information to the access token, and defines the content of the users `status` field according to the email domain.

2. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/00-throttle-mapped.json
```

```
{
  "name": "00-throttle-mapped",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/throttle-
mapped')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [
              "mail",
              "employeenumber"
            ],
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name": "token-resolver-1",
              "type": "TokenIntrospectionAccessTokenResolver",
              "config": {
                "amService": "AmService-1",
                "providerHandler": {
                  "type": "Chain",
                  "config": {
                    "filters": [
                      {
                        "type":
"HttpBasicAuthenticationClientFilter",
                        "config": {
                          "username": "ig_agent",
                          "passwordSecretId":
"agent.secret.id",
                          "secretsProvider":
"SystemAndEnvSecretStore-1"
                        }
                      }
                    ],
                    "handler": "ForgeRockClientHandler"
                  }
                }
              }
            }
          }
        },
        {
          "name": "ThrottlingFilter-1",
          "type": "ThrottlingFilter",
          "config": {
            "requestGroupingPolicy":
"${contexts.oauth2.accessToken.info.mail}",
            "throttlingRatePolicy": {
              "name": "MappedPolicy",
              "type": "MappedThrottlingPolicy",
              "config": {
                "throttlingRateMapper":
"${contexts.oauth2.accessToken.info.status}",
                "throttlingRatesMapping": {
                  "gold": {
                    "numberOfRequests": 6,
                    "duration": "10 s"
                  },
                  "silver": {
                    "numberOfRequests": 3,
                    "duration": "10 s"
                  },
                  "bronze": {
                    "numberOfRequests": 1,
                    "duration": "10 s"
                  }
                },
                "defaultRate": {
                  "numberOfRequests": 1,
                  "duration": "10 s"
                }
              }
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

For information about how to set up the PingGateway route in Studio, refer to Mapped throttling filter in Structured Editor.

Notice the following features of the route:

- The route matches requests to `/home/throttle-mapped`.
- The OAuth2ResourceServerFilter validates requests with the AccessTokenResolver, and makes it available for downstream components in the `oauth2` context.
- The ThrottlingFilter bases the request grouping policy on the AM user's email. The throttling rate is applied independently to each email address.

  The throttling rate is mapped to the AM user's `status`, which is defined by the email domain, in the AM script.

3. Test the setup:

a. In a terminal window, use a **`curl`** command similar to this to retrieve an access token:

```
$ mytoken=$(curl -s \
--user "client-application:password" \
--data
"grant_type=password&username=demo&password=Ch4ng31t&scope
=mail%20employeenumber" \
http://am.example.com:8088/openam/oauth2/access_token | jq
-r ".access_token")
```

b. Using the access token, access the route multiple times. The following example accesses the route 10 times and writes the output to a file:

```
$ curl -v \
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--header "Authorization: Bearer ${mytoken}" \
https://ig.example.com:8443/home/throttle-mapped/\[01-10\]
\
> /tmp/throttle-mapped.txt 2>&1
```

c. Search the output file for the result:

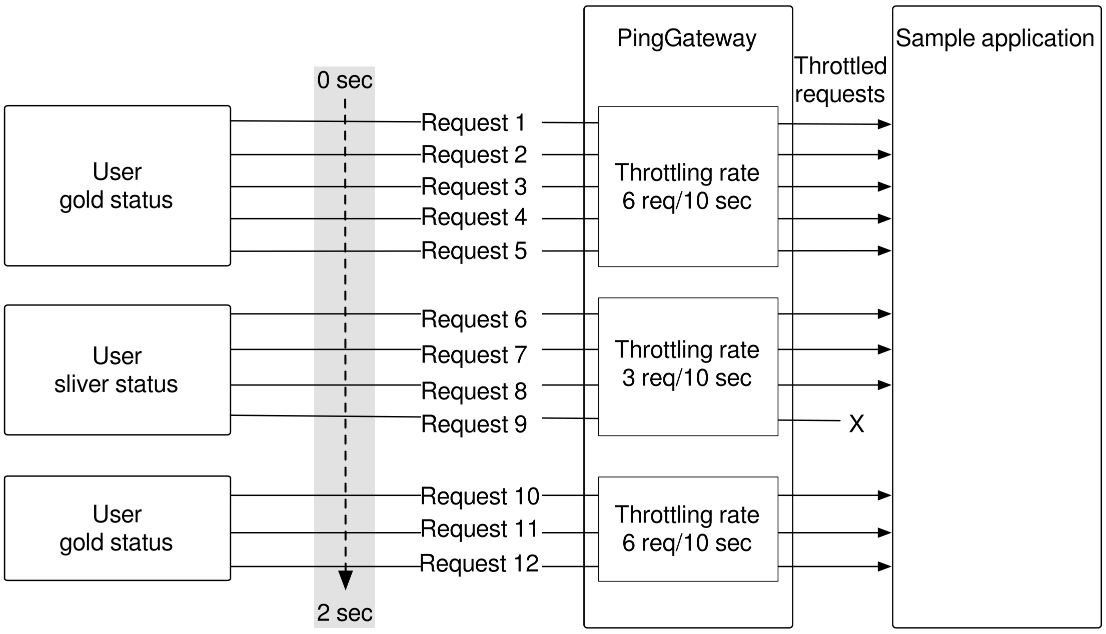
```
$ grep "< HTTP/2" /tmp/throttle-mapped.txt | sort | uniq -
c

6 < HTTP/2 200
4 < HTTP/2 429
```

Notice that with a `gold` status, the user can access the route 6 times in 10 seconds.

d. In AM, change the demo user's email to `demo@other.com`, and then run the last two steps again to find how the access is reduced.

Considerations for dynamic throttling

The following image illustrates what can happen when the throttling rate defined by `throttlingRateMapping` changes frequently or quickly:



In the image, the user starts out with a `gold` status. In a two second period, the users sends five requests, is downgraded to silver, sends four requests, is upgraded back to `gold`, and then sends three more requests.

After making five requests with a `gold` status, the user has almost reached their throttling rate. When his status is downgraded to silver, those requests are disregarded and the full throttling rate for `silver` is applied. The user can now make three more requests even though they have nearly reached their throttling rate with a `gold` status.

After making three requests with a `silver` status, the user has reached their throttling rate. When the user makes a fourth request, the request is refused.

The user is now upgraded back to `gold` and can now make six more requests even though they had reached his throttling rate with a `silver` status.

When you configure `requestGroupingPolicy` and `throttlingRateMapper`, bear in mind what happens when the throttling rate defined by the `throttlingRateMapper` is changed.

Configure scriptable throttling

This section builds on the example in Configure mapped throttling. It creates a scriptable throttling filter, where the script applies a throttling rate of 6 requests/10 seconds to requests from gold status users. For all other requests, the script returns `null`, and applies the default rate of 1 request/10 seconds.

Before you start, set up and test the example in Configure mapped throttling.

1. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/00-throttle-scriptable.json
```

```
{
  "name": "00-throttle-scriptable",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/throttle-
scriptable')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [
              "mail",
              "employeenumber"
            ],
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name": "token-resolver-1",
              "type": "TokenIntrospectionAccessTokenResolver",
              "config": {
                "amService": "AmService-1",
                "providerHandler": {
                  "type": "Chain",
                  "config": {
                    "filters": [
                      {
                        "type":
"HttpBasicAuthenticationClientFilter",
                        "config": {
                          "username": "ig_agent",
                          "passwordSecretId":
"agent.secret.id",
                          "secretsProvider":
"SystemAndEnvSecretStore-1"
                        }
                      }
                    ],
                    "handler": "ForgeRockClientHandler"
                  }
                }
              }
            }
          }
        },
        {
          "name": "ThrottlingFilter-1",
          "type": "ThrottlingFilter",
          "config": {
            "requestGroupingPolicy":
"${contexts.oauth2.accessToken.info.mail}",
            "throttlingRatePolicy": {
              "type": "DefaultRateThrottlingPolicy",
              "config": {
                "delegateThrottlingRatePolicy": {
                  "name": "ScriptedPolicy",
                  "type": "ScriptableThrottlingPolicy",
                  "config": {
                    "type": "application/x-groovy",
                    "source": [
                      "if
(contexts.oauth2.accessToken.info.status == status) {",
                      "  return new ThrottlingRate(rate,
duration)",
                      "} else {",
                      "  return null",
                      "}"
                    ],
                    "args": {
                      "status": "gold",
                      "rate": 6,
                      "duration": "10 seconds"
                    }
                  }
                },
                "defaultRate": {
                  "numberOfRequests": 1,
                  "duration": "10 s"
                }
              }
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

For information about how to set up the PingGateway route in Studio, refer to Scriptable throttling filter in Structured Editor.

Notice the following features of the route, compared to path] `00-throttle-mapped.json`:

- The route matches requests to `/home/throttle-scriptable`.
- The DefaultRateThrottlingPolicy delegates the management of throttling to the ScriptableThrottlingPolicy.
- The script applies a throttling rate to requests from users with gold status. For all other requests, the script returns null and the default rate is applied.

2. Test the setup:

   a. In a terminal window, use a **`curl`** command similar to this to retrieve an access token:

```
$ mytoken=$(curl -s \
--user "client-application:password" \
--data
"grant_type=password&username=demo&password=Ch4ng31t&scope
=mail%20employeenumber" \
http://am.example.com:8088/openam/oauth2/access_token | jq
-r ".access_token")
```

b. Using the access token, access the route multiple times. The following example accesses the route 10 times and writes the output to a file:

```
$ curl -v \
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--header "Authorization: Bearer ${mytoken}" \
https://ig.example.com:8443/home/throttle-scriptable/\[01-
10\] \
> /tmp/throttle-script.txt 2>&1
```

c. Search the output file for the result:

```
$ grep "< HTTP/2" /tmp/throttle-script.txt | sort | uniq -
c

6 < HTTP/2 200
4 < HTTP/2 429
```

Notice that with a `gold` status, the user can access the route 6 times in 10 seconds.

d. In AM, change the user's email to `demo@other.com`, and then run the last two steps again to find how the access is reduced.

URI fragments in redirect

URI fragments are optional last parts of a URL for a document, typically used to identify or navigate to a particular part of the document. The fragment part follows the URL after a hash `#`, for example `https://www.rfc-editor.org/rfc/rfc1234#section5`.

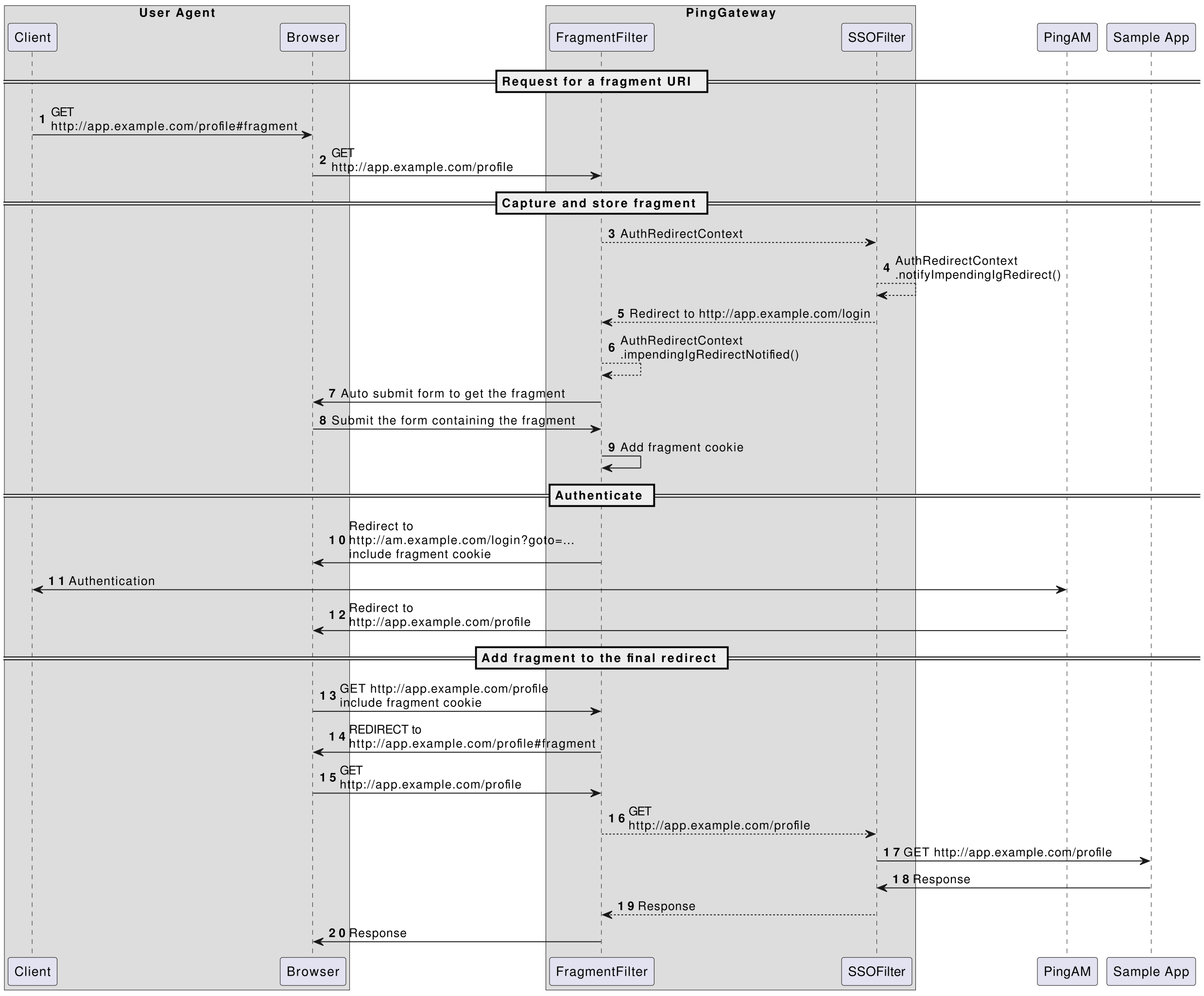
When an unauthenticated user requests a resource that includes a URI fragment, the user agent sends the URI but doesn't send the fragment. The fragment is lost during the authentication flow.

PingGateway provides a FragmentFilter to track the fragment part of a URI when a request triggers a login redirect.

The FragmentFilter doesn't handle multiple fragment captures in parallel. If a fragment capture is in progress while PingGateway performs another login redirect, a second fragment capture process isn't triggered and the fragment is lost.

When a browser request loads a favicon, it can cause the fragment part of a URI to be lost. Prevent problems by serving static resources with a separate route. As an example, use the route in Serve static resources.

The following image shows the flow of information when the FragmentFilter is included in the SSO authentication flow:



User Agent
PingGateway
Client
Browser
FragmentFilter
SSOFilter
PingAM
Sample App
Request for a fragment URI
1 GET http://app.example.com/profile#fragment
2 GET http://app.example.com/profile
Capture and store fragment
3 AuthRedirectContext
4 AuthRedirectContext .notifyImpendingIgRedirect()
5 Redirect to http://app.example.com/login
6 AuthRedirectContext .impendingIgRedirectNotified()
7 Auto submit form to get the fragment
8 Submit the form containing the fragment
9 Add fragment cookie
Authenticate
10 Redirect to http://am.example.com/login?goto=... include fragment cookie
11 Authentication
12 Redirect to http://app.example.com/profile
Add fragment to the final redirect
13 GET http://app.example.com/profile include fragment cookie
14 REDIRECT to http://app.example.com/profile#fragment
15 GET http://app.example.com/profile
16 GET http://app.example.com/profile
17 GET http://app.example.com/profile
18 Response
19 Response
20 Response

1-2. An unauthenticated client requests access to a fragment URL.

3. The FragmentFilter adds the AuthRedirectContext, so that downstream filters can mark the response as redirected.

4-5. The SingleSignOnFilter adds to the context to notify upstream filters that a redirect is pending, and redirects the request for authentication.

6-7. The FragmentFilter is notified by the context that a redirect is pending, and returns a new response object containing the response cookies, an autosubmit HTML form, and Javascript.

8. The user agent runs the Javascript or displays the form’s submit button for the user to click on. This operation POSTs a form request back to a fragment endpoint URI, containing the following parts:

- Request URI path (`/profile`)

- Captured fragment (`#fragment`)
- Login URI (`http://am.example.com/login?goto=…`)

9. The FragmentFilter creates the fragment cookie.

10-12. The client authenticates with AM.

13. The FragmentFilter intercepts the request because it contains a fragment cookie, and its URI matches the original request URI.

The filter redirects the client to the original request URI containing the fragment. The fragment cookie then expires.

14-19. The client follows the final redirect to the original request URI containing the fragment, and the sample app returns the response.

This procedure shows how to persist a URI fragment in an SSO authentication. Before you start, set up and test the example in Use the default journey.

1. In PingGateway, replace sso.json with the following route:

   Linux | Windows

   ```
   $HOME/.openig/config/routes/fragment.json
   ```

   ```
   {
     "name": "fragment",
     "baseURI": "http://app.example.com:8081",
     "condition": "${find(request.uri.path, '^/home/sso')}",
     "heap": [
       {
         "name": "SystemAndEnvSecretStore-1",
         "type": "SystemAndEnvSecretStore"
       },
       {
         "name": "AmService-1",
         "type": "AmService",
         "config": {
           "agent": {
             "username": "ig_agent",
             "passwordSecretId": "agent.secret.id"
           },
           "secretsProvider": "SystemAndEnvSecretStore-1",
           "url": "http://am.example.com:8088/openam/"
         }
   ```

```
      }
    ],
    "handler": {
      "type": "Chain",
      "config": {
        "filters": [
          {
            "name": "FragmentFilter-1",
            "type": "FragmentFilter",
            "config": {
              "fragmentCaptureEndpoint": "/home/sso"
            }
          },
          {
            "name": "SingleSignOnFilter-1",
            "type": "SingleSignOnFilter",
            "config": {
              "amService": "AmService-1"
            }
          }
        ],
        "handler": "ReverseProxyHandler"
      }
    }
  }
```

Notice the following feature of the route compared to `sso.json`:

- The `FragmentFilter` captures the fragment form data from the route condition endpoint.

2. Test the setup:

   a. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/sso#fragment.

      The SingleSignOnFilter redirects the request to AM for authentication.

   b. Log in to AM as user `demo`, password `Ch4ng31t`.

      The SingleSignOnFilter passes the request to sample application, which returns the home page. Note that the URL of the page has preserved the fragment: `https://ig.example.com:8443/home/sso?_ig=true#fragment`

   c. Remove the FragmentFilter from the route and test the route again.

      Note that this time the URL of the page hasn't preserved the fragment.

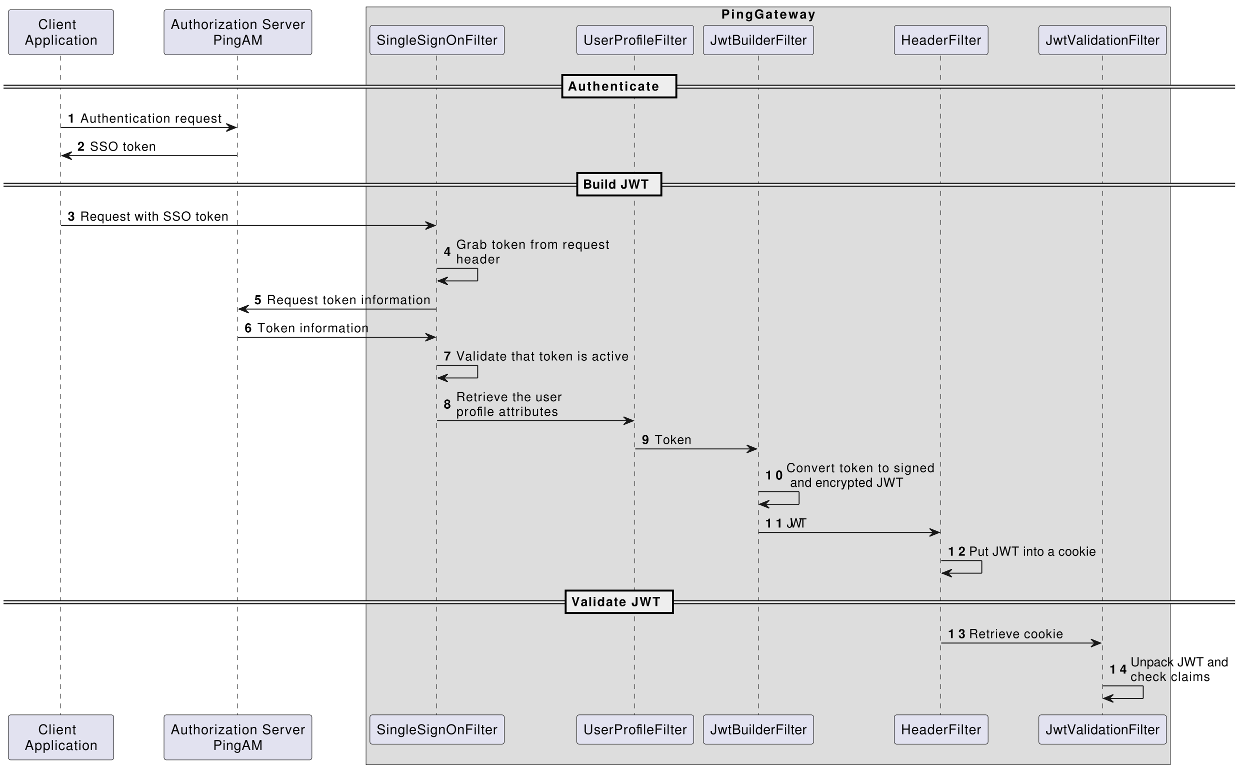
JWT validation

The following examples show how to use the JwtValidationFilter to validate signed and encrypted JWT.

The JwtValidationFilter can access JWTs in the request, provided in a header, query parameter, form parameter, cookie, or other way. If an upstream filter makes the JWT available in the request's attributes context, the JwtValidationFilter can access the JWT through the context, for example, at `${attributes.jwtToValidate}`.

For convenience, the JWT in this example is provided by the JwtBuilderFilter, and passed to the JwtValidationFilter in a cookie.

The following figure shows the flow of information in the example:



Before you start, set up and test the example in Pass runtime data in a JWT signed with PEM then encrypted with a symmetric key.

1. Add a second route to PingGateway, replacing value of the property `secretsDir` with the directory for the PEM files:

Linux | **Windows**

```
$HOME/.openig/config/routes/jwt-validate.json
```

```
{
  "name": "jwt-validate",
  "condition": "${find(request.uri.path, '^/jwt-validate')}",
  "properties": {
    "secretsDir": "path/to/secrets"
  },
  "capture": "all",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore",
      "type": "SystemAndEnvSecretStore",
      "config": {
        "mappings": [{
          "secretId": "id.decrypted.key.for.signing.jwt",
          "format": "BASE64"
        }]
      }
    },
    {
      "name": "pemPropertyFormat",
      "type": "PemPropertyFormat",
      "config": {
        "decryptionSecretId":
"id.decrypted.key.for.signing.jwt",
        "secretsProvider": "SystemAndEnvSecretStore"
      }
    },
    {
      "name": "FileSystemSecretStore-1",
      "type": "FileSystemSecretStore",
      "config": {
        "format": "PLAIN",
        "directory": "&{secretsDir}",
        "mappings": [{
          "secretId": "id.encrypted.key.for.signing.jwt.pem",
          "format": "pemPropertyFormat"
        }, {
          "secretId": "symmetric.key.for.encrypting.jwt",
          "format": {
            "type": "SecretKeyPropertyFormat",
            "config": {
              "format": "BASE64",
              "algorithm": "AES"
            }
          }
        }]
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [{
        "type": "JwtValidationFilter",
        "config": {
          "jwt": "${request.cookies['my-jwt'][0].value}",
          "secretsProvider": "FileSystemSecretStore-1",
          "decryptionSecretId":
"symmetric.key.for.encrypting.jwt",
          "customizer": {
            "type": "ScriptableJwtValidatorCustomizer",
            "config": {
              "type": "application/x-groovy",
              "source": [
                "builder.claim('name', JsonValue::asString,
isEqualTo('demo'))",
                "builder.claim('email', JsonValue::asString,
isEqualTo('demo@example.com'));"
              ]
            }
          },
          "failureHandler": {
            "type": "ScriptableHandler",
            "config": {
              "type": "application/x-groovy",
              "source": [
                "def response = new
Response(Status.FORBIDDEN)",
                "response.headers['Content-Type'] =
'text/html; charset=utf-8'",
                "def errors =
contexts.jwtValidationError.violations.collect{it.description}
",
                "def display = \"<html>Can't validate JWT:<br>
${contexts.jwtValidationError.jwt} \"",
                "display <<=\"<br><br>For the following
errors:<br> ${errors.join(\"<br>\")}</html>\"",
                "response.entity=display as String",
                "return response"
              ]
            }
          }
        }
      }],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": [
            "<html>",
            "   <h2>Validated JWT:</h2>",
            "     <p>${contexts.jwtValidation.value}</p>",
            "   <h2>JWT payload:</h2>",
            "     <p>${contexts.jwtValidation.info}</p>",
            "</html>"
          ]
        }
      }
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/jwt-validate`.
- The JwtValidationFilter takes the value of the JWT from `my-jwt`.
- The SystemAndEnvSecretStore, PemPropertyFormat, and FileSystemSecretStore objects in the heap are the same as those in the route to create the JWT. The JwtValidationFilter uses the same objects to validate the JWT.
- The JwtBuilderFilter `customizer` requires that the JWT claims match `name:demo and email:demo@example.com`.
- If the JWT is validated, the StaticResponseHandler displays the validated value. Otherwise, the FailureHandler displays the reason for the failed validation.

2. Test the setup:

   a. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/jwtbuilder-sign-then-encrypt to build a JWT.

   b. Log in to AM as user `demo`, password `Ch4ng31t`. The sample application displays the signed JWT.

c. Go to https://ig.example.com:8443/jwt-validate to validate the JWT. The validated JWT and its payload are displayed.

d. Test the setup again, but log in to AM as a different user, or change the email address of the demo user in AM. The JWT isn't validated, and an error is displayed.

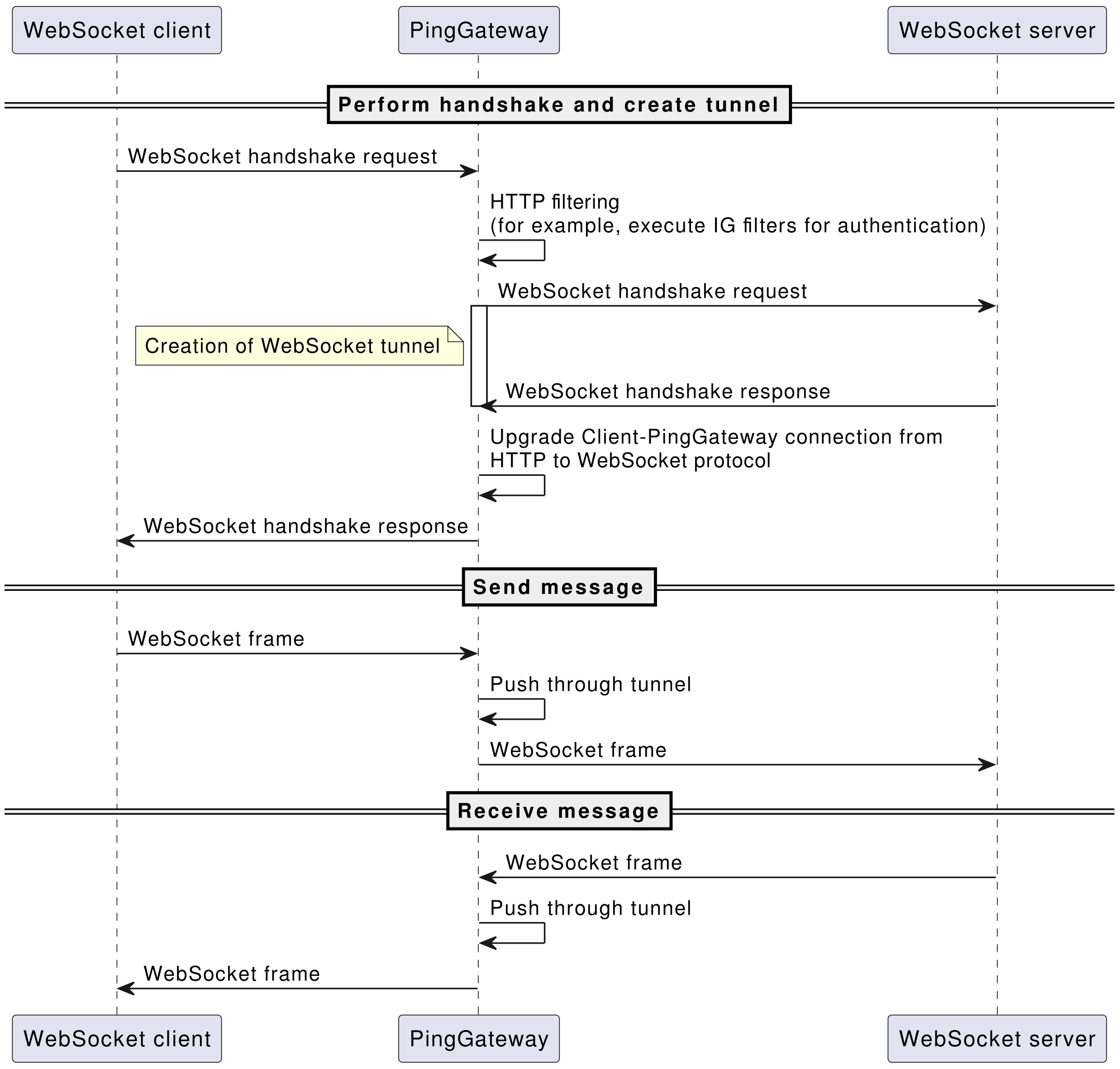
WebSocket traffic

When a user agent requests an upgrade from HTTP or HTTPS to the WebSocket protocol, PingGateway detects the request and performs an HTTP handshake request between the user agent and the protected application.

If the handshake is successful, PingGateway upgrades the connection and provides a dedicated tunnel to route WebSocket traffic between the user agent and the protected application. PingGateway doesn't intercept messages to or from the WebSocket server.

The tunnel remains open until it's closed by the user agent or protected application. When the user agent closes the tunnel, the connection between PingGateway and the protected application is automatically closed.

The following sequence diagram shows the flow of information when PingGateway proxies WebSocket traffic:


WebSocket client
PingGateway
WebSocket server
Perform handshake and create tunnel
WebSocket handshake request
HTTP filtering
(for example, execute IG filters for authentication)
WebSocket handshake request
Creation of WebSocket tunnel
WebSocket handshake response
Upgrade Client-PingGateway connection from HTTP to WebSocket protocol
WebSocket handshake response
Send message
WebSocket frame
Push through tunnel
WebSocket frame
Receive message
WebSocket frame
Push through tunnel
WebSocket frame
WebSocket client
PingGateway
WebSocket server

To configure PingGateway to proxy WebSocket traffic, configure the `websocket` property of ReverseProxyHandler. By default, PingGateway doesn't proxy WebSocket traffic.

Proxy WebSocket traffic

1. Set up AM:

   a. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:

      - `https://ig.example.com:8443/*`
      - `https://ig.example.com:8443/*?*`

   b. Register a PingGateway agent with the following values, as described in Register a PingGateway agent in AM:

      - **Agent ID**: `ig_agent`
      - **Password**: `password`

IMPORTANT

IMPORTANT

Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

c. (Optional) Authenticate the agent to AM as described in Authenticate a PingGateway agent to AM.

IMPORTANT

PingGateway agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

2. Set up PingGateway:

   a. Set up PingGateway for HTTPS, as described in Configure PingGateway for TLS (server-side).

   b. Set an environment variable for the PingGateway agent password, and then restart PingGateway:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

   The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

   c. Add the following route to PingGateway to serve the sample application .css and other static resources:

Linux Windows

```
$HOME/.openig/config/routes/00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or
matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

   d. Add the following route to PingGateway:

Linux Windows

```
$HOME/.openig/config/routes/websocket.json
```

```
{
  "name": "websocket",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/websocket')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    },
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler",
      "config": {
        "websocket": {
          "enabled": true,
          "vertx": {
            "maxFrameSize": 200000000,
            "maxMessageSize": 200000000,
            "tryUsePerMessageCompression": true
          }
        }
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SingleSignOnFilter-1",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

You can find information about how to set up the route in Studio in Proxy for WebSocket traffic in Structured Editor.

Learn more about Vert.x options for WebSocket connections in HttpClientOptions.

Notice the following features of the route:

- The route matches requests to `/websocket`, the endpoint on the sample app that exposes a WebSocket server.
- The SingleSignOnFilter redirects unauthenticated requests to AM for authentication.
- The ReverserProxyHandler enables PingGateway to proxy WebSocket traffic. After PingGateway upgrades the HTTP connection to the WebSocket protocol, the ReverserProxyHandler passes the messages to the WebSocket server.

3. Test the setup:

   a. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/websocket.

   b. Sign on to AM as user `demo`, password `Ch4ng31t`.

      AM authenticates the user, creates an SSO token, and redirects the request back to the original URI, with the token in a cookie.

      The request then passes to the ReverseProxyHandler, which routes the request to the HTML page `/websocket/index.html` of the sample app. The page initiates the HTTP handshake for connecting to the WebSocket endpoint `/websocket/echo`.

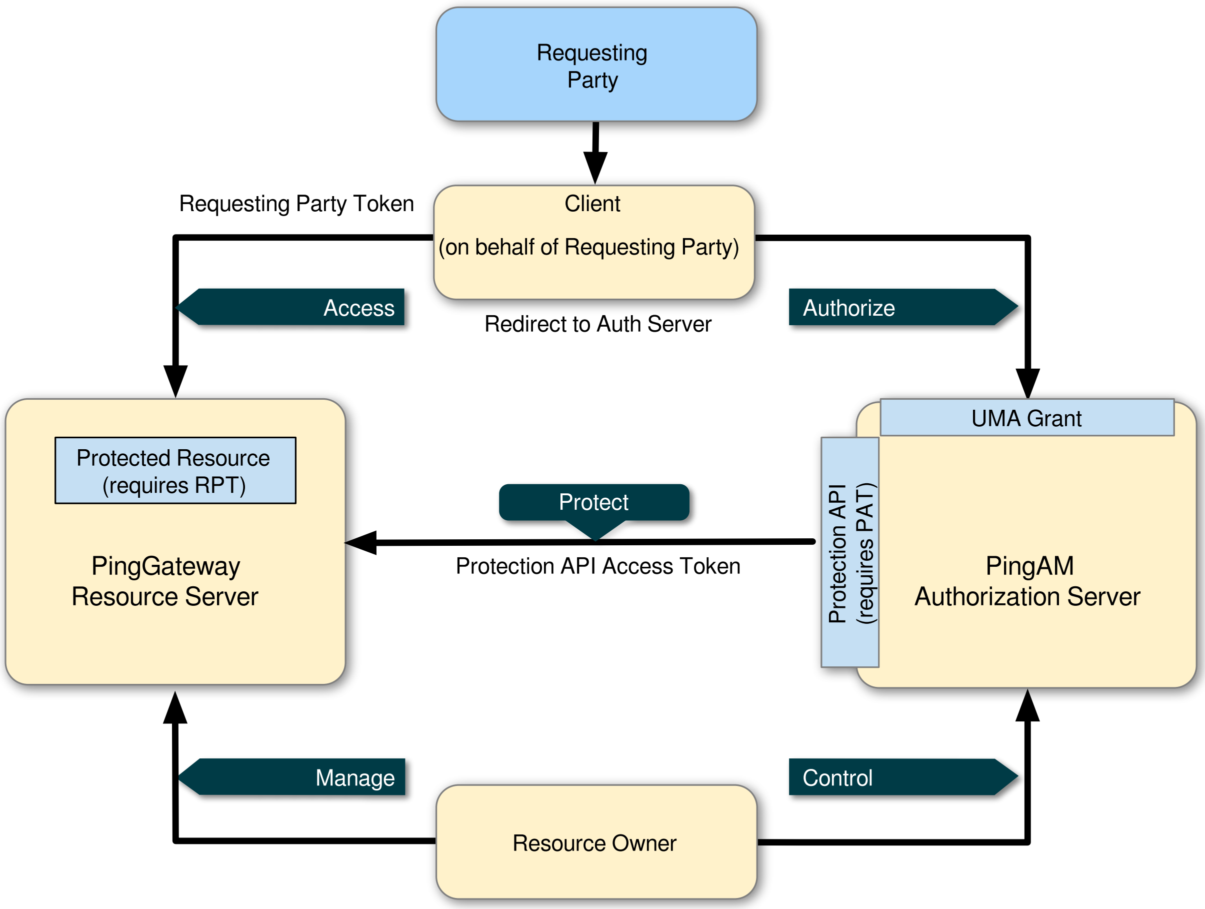
   c. Enter text on the WebSocket echo screen and note that the text is echoed back.

UMA support

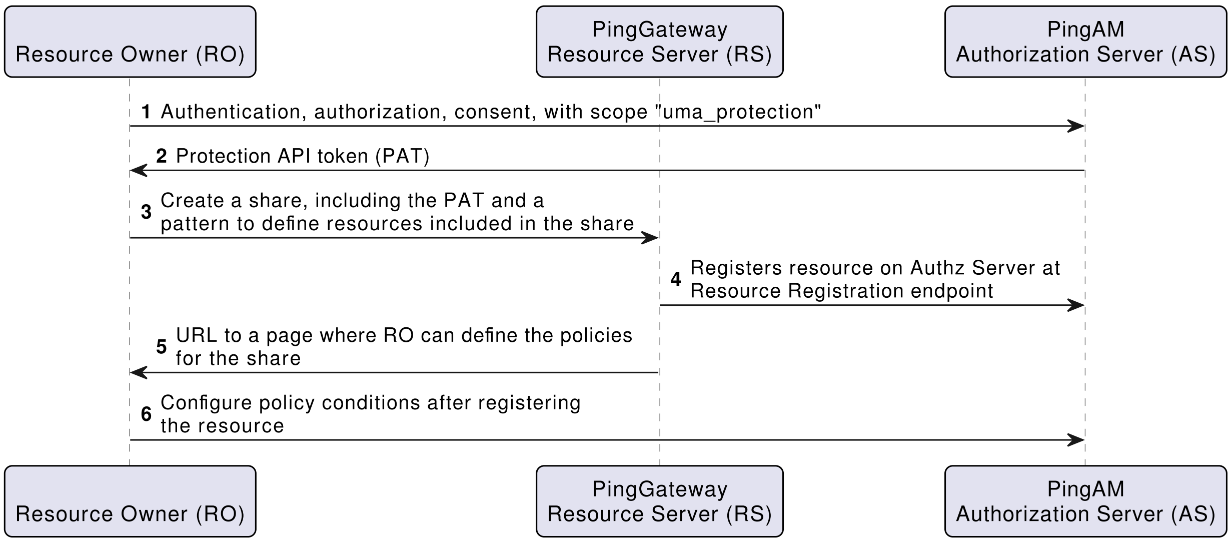
PingGateway includes support for User-Managed Access (UMA) 2.0 Grant for OAuth 2.0 Authorization specifications.

About PingGateway as an UMA resource server

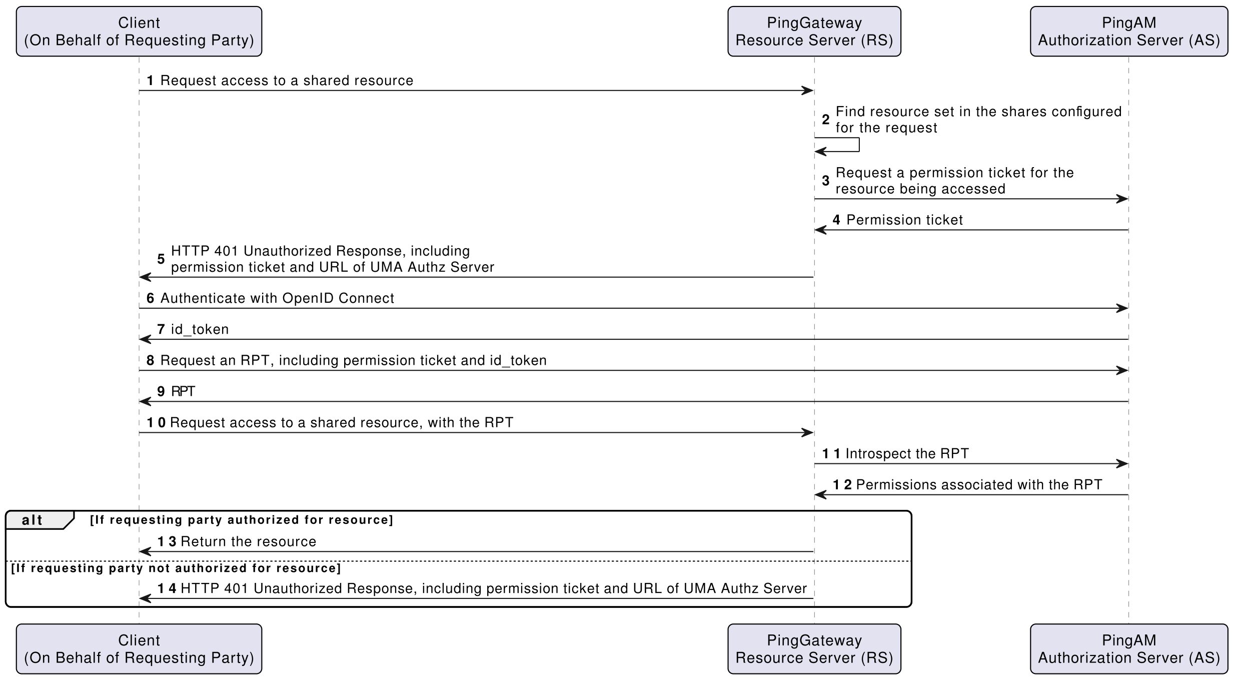
The following figure shows an UMA environment, with PingGateway protecting a resource, and AM acting as an Authorization Server. Learn more in the AM documentation on User-Managed Access (UMA) 2.0.


Requesting Party
Requesting Party Token
Client
(on behalf of Requesting Party)
Access
Redirect to Auth Server
Authorize
Protected Resource (requires RPT)
PingGateway Resource Server
Protect
Protection API Access Token
UMA Grant
Protection API (requires PAT)
PingAM Authorization Server
Manage
Control
Resource Owner

The following figure shows the data flow when the resource owner registers a resource with AM, and sets up a share using a Protection API Token (PAT):


Resource Owner (RO)
PingGateway Resource Server (RS)
PingAM Authorization Server (AS)
1 Authentication, authorization, consent, with scope "uma_protection"
2 Protection API token (PAT)
3 Create a share, including the PAT and a pattern to define resources included in the share
4 Registers resource on Authz Server at Resource Registration endpoint
5 URL to a page where RO can define the policies for the share
6 Configure policy conditions after registering the resource
Resource Owner (RO)
PingGateway Resource Server (RS)
PingAM Authorization Server (AS)

The following figure shows the data flow when the client accesses the resource, using a Requesting Party Token (RPT):


Client (On Behalf of Requesting Party)
PingGateway Resource Server (RS)
PingAM Authorization Server (AS)
1 Request access to a shared resource
2 Find resource set in the shares configured for the request
3 Request a permission ticket for the resource being accessed
4 Permission ticket
5 HTTP 401 Unauthorized Response, including permission ticket and URL of UMA Authz Server
6 Authenticate with OpenID Connect
7 id_token
8 Request an RPT, including permission ticket and id_token
9 RPT
10 Request access to a shared resource, with the RPT
11 Introspect the RPT
12 Permissions associated with the RPT
alt
[If requesting party authorized for resource]
13 Return the resource
[If requesting party not authorized for resource]
14 HTTP 401 Unauthorized Response, including permission ticket and URL of UMA Authz Server
Client (On Behalf of Requesting Party)
PingGateway Resource Server (RS)
PingAM Authorization Server (AS)

You can find information about CORS support in the AM documentation on configuring CORS support. This procedure describes how to modify the AM configuration to allow cross-site access.

Limitations of PingGateway as an UMA resource server

When using PingGateway as an UMA resource server, note the following points:

- When you deploy the administrative endpoint on another port using `"adminConnector"` settings in `admin.json`, expose the port for use in the UMA deployment.
- PingGateway depends on the resource owner for the PAT.

When a PAT expires, no refresh token is available to PingGateway. The resource owner must repeat the entire share process with a new PAT in order to authorize access to protected resources. The resource owner should delete the old resource and create a new one.

- Data about PATs and shared resources is held in memory.

  PingGateway has no mechanism for persisting the data across restarts. When PingGateway stops and starts again, the resource owner must repeat the entire share process.

- UMA client applications for sharing and accessing protected resources must deal with UMA error conditions and PingGateway error conditions.
- By default, the REST API to manage share objects exposed by PingGateway is protected only by CORS.
- When matching protected resource paths with share patterns, PingGateway takes the longest match.

  For example, if resource owner Alice shares `/photos/.*` with Bob, and `/photos/vacation.png` with Charlie, and then Bob attempts to access `/photos/vacation.png`, PingGateway applies the sharing permissions for Charlie, not Bob. As a result, Bob can be denied access.

Set up the UMA example

This section describes tasks to set up AM as an Authorization Server:

- Enabling cross-origin resource sharing (CORS) support in AM
- Configuring AM as an Authorization Server
- Registering UMA client profiles with AM
- Setting up a resource owner (Alice) and requesting party (Bob)

CAUTION

The settings in this section are suggestions for this tutorial. They aren't intended as instructions for setting up AM CORS support on a server in production.

If you need to accept all origins, by allowing the use of `Access-Control-Allowed-Origin=*`, don't allow `Content-Type` headers. Allowing the use of both types of headers exposes AM to cross-site request forgery (CSRF) attacks.

IMPORTANT

This procedure uses the *Resource Owner Password Credentials* grant type. As suggested in The OAuth 2.0 Authorization Framework, use other grant types whenever possible.

Before you begin, prepare AM, PingGateway, and the sample application. Learn more in the example installation for this guide.

You can find more information about using different settings for the sample application in Edit the example to match custom settings.

1. Set up AM:
   a. Find the name of the AM session cookie at the `/json/serverinfo/*` endpoint. This procedure assumes that you are using the default AM session cookie, `iPlanetDirectoryPro`.
   b. Create an OAuth 2.0 Authorization Server:
      i. Select **Services** > **Add a Service** > **OAuth2 Provider**.
      ii. Add a service with the default values.
   c. Configure an UMA Authorization Server:
      i. Select **Services** > **Add a Service** > **UMA Provider**.
      ii. Add a service with the default values.
   d. Add an OAuth 2.0 client for UMA protection:
      i. Select **Applications** > **OAuth 2.0** > **Clients**.
      ii. Add a client with these values:
         - **Client ID** : `OpenIG`
         - **Client secret** : `password`
         - **Scope** : `uma_protection`
      iii. (Optional) On the **Core** tab, switch to using a client secret associated with a secret label by setting a **Secret Label Identifier** and mapping the label to a secret.

         To learn more, read Create a client profile and Map and rotate secrets in the AM documentation.
      iv. On the **Advanced** tab, select the following option:
         - **Grant Types** : `Resource Owner Password Credentials`
   e. Add an OAuth 2.0 client for accessing protected resources:
      i. Select **Applications** > **OAuth 2.0** > **Clients**.
      ii. Add a client with these values:
         - **Client ID** : `UmaClient`
         - **Client secret** : `password`
         - **Scope** : `openid`
      iii. On the **Advanced** tab, select the following option:

- **Grant Types**: `Resource Owner Password Credentials and UMA`

f. Select **Identities**, and add an identity for a resource owner with the following values:

- **ID**: `alice`
- **Password**: `UMAexamp1e`
- **Email Address**: `alice@example.com`

g. Select **Identities**, and add an identity for a requesting party, with the following values:

- **ID**: `bob`
- **Password**: `UMAexamp1e`
- **Email Address**: `bob@example.com`

h. Enable the CORS filter on AM:

i. In a terminal window, retrieve an SSO token from AM:

```
$ mytoken=$(curl --request POST \
--header "Accept-API-Version: resource=2.1" \
--header "X-OpenAM-Username: amadmin" \
--header "X-OpenAM-Password: password" \
--header "Content-Type: application/json" \
--data "{}"  \
http://am.example.com:8088/openam/json/authenticate |
jq -r ".tokenId")
```

ii. Using the token retrieved in the previous step, enable the CORS filter on AM by using the `/global-config/services/CorsService` REST endpoint:

```
$ curl  \
  --request PUT \
  --header "Content-Type: application/json" \
  --header "iPlanetDirectoryPro: $mytoken"
http://am.example.com:8088/openam/json/global-
config/services/CorsService/configuration/CorsService \
  --data '{
     "acceptedMethods": [
       "POST",
       "GET",
       "PUT",
       "DELETE",
       "PATCH",
       "OPTIONS"
```

```
    ],
    "acceptedOrigins": [
      "http://app.example.com:8081",
      "http://ig.example.com:8080",
      "http://am.example.com:8088/openam"
    ],
    "allowCredentials": true,
    "acceptedHeaders": [
      "Authorization",
      "Content-Type",
      "iPlanetDirectoryPro",
      "X-OpenAM-Username",
      "X-OpenAM-Password",
      "Accept",
      "Accept-Encoding",
      "Connection",
      "Content-Length",
      "Host",
      "Origin",
      "User-Agent",
      "Accept-Language",
      "Referer",
      "Dnt",
      "Accept-Api-Version",
      "If-None-Match",
      "Cookie",
      "X-Requested-With",
      "Cache-Control",
      "X-Password",
      "X-Username",
      "X-NoSession"
    ],
    "exposedHeaders": [
      "Access-Control-Allow-Origin",
      "Access-Control-Allow-Credentials",
      "Set-Cookie",
      "WWW-Authenticate"
    ],
    "maxAge": 600,
    "enabled": true,
    "allowCredentials": true
}'
```

A CORS configuration is diplayed.

TIP

To delete the CORS configuration and create another, first run the following command:

```
$ curl \
 --request DELETE \
 --header "X-Requested-With: XMLHttpRequest" \
 --header "iPlanetDirectoryPro: $mytoken" \
 http://am.example.com:8088/openam/json/global-
config/services/CorsService/CorsService/configuration/CorsSer
vice
```

2. Set up PingGateway as an UMA resource server:

   a. Add the following route to PingGateway to serve the sample application .css and other static resources:

   Linux | Windows

   ```
   $HOME/.openig/config/routes/00-static-resources.json
   ```

   ```
   {
     "name" : "00-static-resources",
     "baseURI" : "http://app.example.com:8081",
     "condition": "${find(request.uri.path,'^/css') or
   matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
   matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
     "handler": "ReverseProxyHandler"
   }
   ```

   b. Add the following settings to `admin.json` configuration and restart PingGateway:

   ```
   {
     "connectors": [
       { "port" : 8080 }
     ],
     "heap": [
       {
         "name": "ClientHandler",
         "type": "ClientHandler"
       }
     ],
     "apiProtectionFilter": {
   ```

```
      "type": "CorsFilter",
      "config": {
        "policies": [
          {
            "acceptedOrigins": [
"http://app.example.com:8081" ],
            "acceptedMethods": [ "GET", "POST", "DELETE" ],
            "acceptedHeaders": [ "Content-Type" ]
          }
        ]
      }
    }
}
```

Notice the following feature:

- The default `"apiProtectionFilter"` uses a CorsFilter to allow requests from the origin `http://app.example.com:8081`.

c. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/00-uma.json
```

```
{
  "name": "00-uma",
  "condition": "${request.uri.host == 'app.example.com'}",
  "heap": [
    {
      "name": "UmaService",
      "type": "UmaService",
      "config": {
        "protectionApiHandler": "ClientHandler",
        "wellKnownEndpoint":
"http://am.example.com:8088/openam/uma/.well-known/uma2-
configuration",
        "resources": [
          {
            "comment": "Protects all resources matching
the following pattern.",
            "pattern": ".*",
            "actions": [
              {
                "scopes": [
                  "#read"
                ],
                "condition": "${request.method == 'GET'}"
              },
              {
                "scopes": [
                  "#create"
                ],
                "condition": "${request.method == 'POST'}"
              }
            ]
          }
        ]
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "CorsFilter",
          "config": {
            "policies": [
              {
                "acceptedOrigins": [
"http://app.example.com:8081" ],
                "acceptedMethods": [ "GET" ],
                "acceptedHeaders": [ "Authorization" ],
                "exposedHeaders": [ "WWW-Authenticate" ],
                "allowCredentials": true
              }
            ]
          }
        },
        {
          "type": "UmaFilter",
          "config": {
            "protectionApiHandler": "ClientHandler",
            "umaService": "UmaService"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
      }
    }
  }
```

Notice the following features of the route:

- The route matches requests from `app.example.com`.
- The UmaService describes the resources that a resource owner can share, using AM as the Authorization Server. It provides a REST API to manage sharing of resource sets.
- The CorsFilter defines the policy for cross-origin requests, listing the methods and headers that the request can use, the headers that are exposed to the frontend JavaScript code, and whether the request can use credentials.
- The UmaFilter manages requesting party access to protected resources, using the UmaService. Protected resources are on the sample application, which responds to requests on port 8081.

3. Test the setup:

   a. In your browser's privacy or incognito mode, go to http://app.example.com:8081/uma/.

   b. Share resources:

      i. Select **Alice shares resources**.

      ii. On Alice's page, select **Share with Bob**. The following items are displayed:

         - The PAT that Alice receives from AM.
         - The metadata for the resource set that Alice registers through PingGateway.
         - The result of Alice authenticating with AM in order to create a policy.
         - The successful result when Alice configures the authorization policy attached to the shared resource.

         If the step fails, run the following command to get an access token for Alice:

```
$ curl -X POST \
-H "Cache-Control: no-cache" \
-H "Content-Type: application/x-www-form-
urlencoded" \
-d
'grant_type=password&scope=uma_protection&username=
alice&password=UMAexamp1e&client_id=OpenIG&client_s
ecret=password' \
http://am.example.com:8088/openam/oauth2/access_tok
en
```

If you fail to get an access token, check that AM is configured as described in this procedure. If you continue to have problems, make sure that your PingGateway configuration matches that shown when you are running the test on http://app.example.com:8081/uma/.

c. Access resources:

   i. Go back to the first page, and select **Bob accesses resources**.

   ii. On Bob's page, select **Get Alice's resources**. The following items are displayed:

      - The WWW-Authenticate Header.
      - The OpenID Connect Token that Bob gets to obtain the RPT.
      - The RPT that Bob gets in order to request the resource again.
      - The final response containing the body of the resource.

Edit the example to match custom settings

If you use a configuration that is different to that described in this chapter, consider the following tasks to adjust the sample to your configuration:

1. Unpack the UMA files from the sample application described in Use the sample application to temporary folder:

```
$ mkdir /tmp/uma
$ cd /tmp/uma
$ jar -xvf /path/to/PingGateway-sample-application-2024.11.0-
jar-with-dependencies.jar webroot-uma

created: webroot-uma/
inflated: webroot-uma/bob.html
inflated: webroot-uma/common.js
inflated: webroot-uma/alice.html
inflated: webroot-uma/index.html
inflated: webroot-uma/style.css
```

2. Edit the configuration in `common.js`, `alice.html`, and `bob.html` to match your settings.
3. Repack the UMA sample client files and then restart the sample application:

```
$ jar -uvf /path/to/PingGateway-sample-application-2024.11.0-
jar-with-dependencies.jar webroot-uma
```

```
adding: webroot-uma/(in = 0) (out= 0)(stored 0%)
adding: webroot-uma/bob.html(in = 26458) (out= 17273)(deflated
34%)
adding: webroot-uma/common.js(in = 3652) (out= 1071)(deflated
70%)
adding: webroot-uma/alice.html(in = 27775) (out= 17512)
(deflated 36%)
adding: webroot-uma/index.html(in = 22046) (out= 16060)
(deflated 27%)
adding: webroot-uma/style.css(in = 811) (out= 416)(deflated
48%)
updated module-info: module-info.class
```

4. If necessary, adjust the CORS settings for AM.

Understand the UMA API with an API descriptor

The UMA share endpoint serves API descriptors at runtime. When you retrieve an API descriptor for the endpoint, a JSON that describes the API for the endpoint is returned.

You can use the API descriptor with a tool such as Swagger UI to generate a web page that helps you to view and test the endpoint. For information, refer to API descriptors.

Protect PingAM

In its role as a reverse proxy, PingGateway can protect PingAM.

Choose what you allow

The best practice is to allow access only to required AM services:

- Limit access to specified AM realms.
- Prevent AM admin UI access.

Learn more about this process in the following articles:

- Best practice for blocking the top-level realm in a proxy for PingAM
- How do I remove admin UI access in PingAM?

Example route

The following example allows access to the following:

- Key services in the `customers` realm.
- Authentication and logout through XUI and the legacy UI.
- SAML v2.0 services.

Before you begin

- Install and run PingGateway as described in the Quick install.
- Install and configure AM with a `customers` subrealm of the top-level realm.

Allow-only route

Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/protect-am.json
```

```
{
  "name": "protect-am",
  "comment": "Allow authentication and subrealm access only (not
the top-level realm or admin UI)",
  "properties": {
    "amBase": "/am",
    "amInstanceUrl": "http://am.example.com:8088/am",
    "subrealm": "customers"
  },
  "baseURI": "&{amInstanceUrl}",
  "condition": "${find(request.uri.path, '^&{amBase}')}",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "AllowOnlyFilter",
          "config": {
            "rules": [
              {
                "comment": "Allow authentication to the subrealm",
                "destination": [
                  {
                    "paths": [
                      "&{amBase}/json/realms/root/realms/&
{subrealm}/authenticate"
                    ],
                    "when": "${find(request.queryParams, 'realm')
and !(contains(request.queryParams['realm'], '/') or
contains(request.queryParams['realm'], '2F') or
contains(request.queryParams['authIndexValue'], 'Application') or
contains(request.queryParams['module'], 'Application'))}"
                  }
                ]
              },
              {
                "comment": "Allow authentication to the legacy
UI",
                "destination": [
                  {
                    "paths": [
                      "&{amBase}/UI/Login"
                    ],
                    "when": "${find(request.queryParams, 'realm')
and !(contains(request.queryParams['realm'], '/') or
contains(request.queryParams['realm'], '2F') or
contains(request.queryParams['authIndexValue'], 'Application') or
contains(request.queryParams['module'], 'Application'))}"
                  }
                ]
              },
              {
                "comment": "Allow OAuth 2.0 and OID to the
subrealm",
                "destination": [
                  {
                    "paths": [
                      "&{amBase}/oauth2/realms/root/realms/&
{subrealm}/authorize",
                      "&{amBase}/oauth2/realms/root/realms/&
{subrealm}/access_token",
                      "&{amBase}/oauth2/realms/root/realms/&
{subrealm}/userinfo",
                      "&{amBase}/oauth2/realms/root/realms/&
{subrealm}/connect/endSession"
                    ]
                  }
                ]
              },
              {
                "comment": "Allow base requests for sessions and
users",
                "destination": [
                  {
                    "paths": [
                      "&{amBase}/json/sessions",
                      "&{amBase}/json/users"
                    ]
                  }
                ]
              },
              {
                "comment": "Allow base action requests for
sessions",
                "destination": [
                  {
                    "paths": [
                      "&{amBase}/json/realms/root/sessions"
                    ]
                  }
                ],
                "when": "${(request.queryParams['_action'] ==
'getMaxIdle') or (request.queryParams['_action'] == 'logout') or
(request.queryParams['_action'] == 'validate')}"
              },
              {
                "comment": "Allow base action requests for users",
                "destination": [
                  {
                    "paths": [
                      "&{amBase}/json/realms/root/users"
                    ]
                  }
                ],
                "when": "${request.queryParams['_action'] ==
'idFromSession'}"
              },
              {
                "comment": "Allow subrealm requests for sessions,
serverinfo, and users",
                "destination": [
                  {
                    "paths": [
                      "&{amBase}/json/realms/root/realms/&
{subrealm}/sessions",
                      "&{amBase}/json/realms/root/realms/&
{subrealm}/serverinfo/*",
                      "^&{amBase}/json/realms/root/realms/&
{subrealm}/users"
                    ]
                  }
                ]
              },
              {
                "comment": "Allow access to the XUI",
                "destination": [
                  {
                    "paths": [
                      "^&{amBase}/XUI"
                    ]
                  }
                ]
              },
              {
                "comment": "Allow access to the legacy UI for
logout",
                "destination": [
                  {
                    "paths": [
                      "&{amBase}/UI/Logout"
                    ]
                  }
                ]
              },
              {
                "comment": "Allow SAML v2.0 requests",
                "destination": [
                  {
                    "paths": [
                      "^&{amBase}/ArtifactResolver/",
                      "^&{amBase}/Consumer/",
                      "^&{amBase}/IDPSloPOST/",
                      "^&{amBase}/IDPSloRedirect/",
                      "^&{amBase}/IDPSloSoap/",
                      "^&{amBase}/SSORedirect/",
                      "^&{amBase}/idpsaehandler/",
                      "^&{amBase}/saml2/jsp/"
                    ]
                  }
                ]
              }
            ],
            "failureHandler": {
              "type": "StaticResponseHandler",
              "config": {
                "status": 404,
                "headers": {
                  "Content-Type": [
                    "text/html; charset=UTF-8"
                  ]
                },
                "entity": "<html><p>HTTP 404 Not Found</p></html>"
              }
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the key features of the route:

- An AllowOnlyFilter defines the rules for requests PingGateway allows; PingGateway denies all requests not explicitly allowed.

  TIP

  Adapt the route for the deployment, for example, by setting the `amBase`, `amInstanceUrl`, and `subrealm` properties.

- When access to a resource is denied, PingGateway returns HTTP 404 Not Found.

  TIP

  In deployment, avoid leaking information by returning the same response for missing and denied resources.

When you save the updated route file, PingGateway reloads it.

Validation

1. Try an allowed request.

   In your browser's privacy or incognito mode, go to the XUI login page for the subrealm, such as https://ig.example.com:8443/am/XUI/?realm=/customers#login in this example.

PingGateway displays the login page for the realm.

2. Try a request that's not allowed.

   In your browser's privacy or incognito mode, go to the top-level realm server version information resource, such as https://ig.example.com:8443/am/json/serverinfo/version.

   PingGateway returns 404 and displays a message: `HTTP 404 Not Found`.

3. Notice you can still access AM directly, as long as you don't traverse PingGateway.

   For example, go to the base URL for AM, such as http://am.example.com:8088/am.

   AM displays the login page.

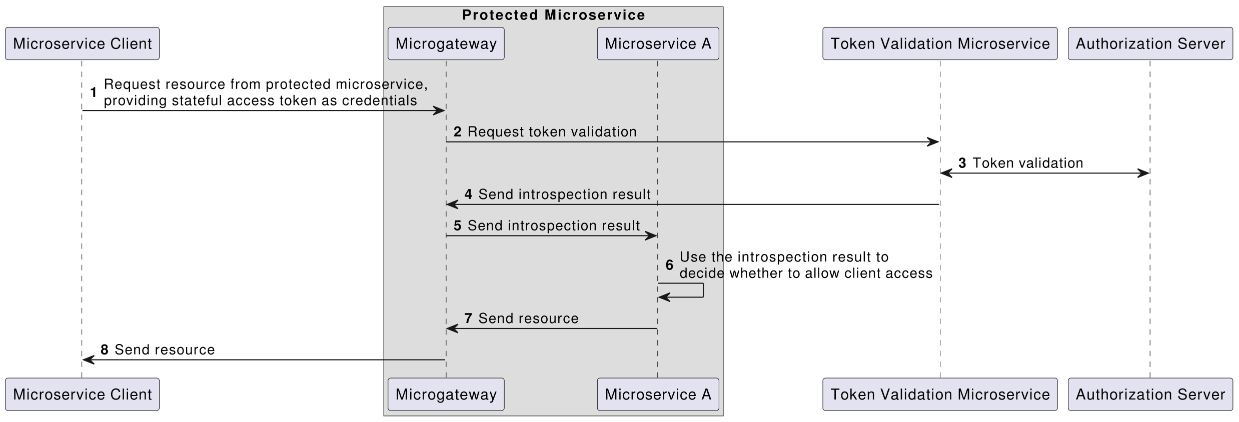
   TIP

   In deployment, route client traffic to AM through PingGateway, as for other protected applications. Don't expose AM endpoints directly.

PingGateway as a microgateway

This section describes how to use the ForgeRock Token Validation Microservice to resolve and cache OAuth 2.0 access tokens when protecting API resources. The section is based on the example in Introspecting stateful access tokens, in the Token Validation Microservice's *User guide*.

For information about the architecture, refer to PingGateway as a microgateway. The following figure illustrates the flow of information when a client requests access to a protected microservice, providing a stateful access token as credentials:


Protected Microservice
Microservice Client
Microgateway
Microservice A
Token Validation Microservice
Authorization Server
1 Request resource from protected microservice, providing stateful access token as credentials
2 Request token validation
3 Token validation
4 Send introspection result
5 Send introspection result
6 Use the introspection result to decide whether to allow client access
7 Send resource
8 Send resource

Before you start, download and run the sample application as described in Use the sample application. The sample application acts as Microservice A.

1. Set up the example in Introspect stateful access tokens, in the Token Validation Microservice's *User guide*.
2. In AM, edit the microservice client to add a scope to access the protected microservice:
   a. Select **Applications** > **OAuth 2.0** > **Clients**.
   b. Select `microservice-client`, and add the scope `microservice-A`.
3. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.openig/config/routes/mgw.json
```

```
{
  "properties": {
    "introspectOAuth2Endpoint":
"http://mstokval.example.com:9090"
  },
  "capture": "all",
  "name": "mgw",
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/home/mgw')}",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "requireHttps": false,
            "accessTokenResolver": {
              "name": "TokenIntrospectionAccessTokenResolver-
1",
              "type": "TokenIntrospectionAccessTokenResolver",
              "config": {
                "endpoint": "&
{introspectOAuth2Endpoint}/introspect",
                "providerHandler": "ForgeRockClientHandler"
              }
            },
            "scopes": ["microservice-A"]
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to PingGateway on `http://ig.example.com:8080/home/mgw`, and rebases them to the sample application, on `http://app.example.com:8081`.
- The OAuth2ResourceServerFilter expects an OAuth 2.0 access token in the header of the incoming authorization request, with the scope `microservice-A`.
- If the filter successfully validates the access token, the ReverseProxyHandler passes the request to the sample application.

4. Test the setup:

   a. With AM, PingGateway, the Token Validation Microservice, and the sample application running, get an access token from AM, using the scope `microservice-A`:

   ```
   $ mytoken=$(curl -s \
   --request POST \
   --url
   http://am.example.com:8088/openam/oauth2/access_token \
   --user microservice-client:password \
   --data grant_type=client_credentials \
   --data scope=microservice-A --silent | jq -r
   .access_token)
   ```

   b. View the access token:

   ```
   $ echo $mytoken
   ```

   c. Call PingGateway to access microservice A:

   ```
   $ curl -v --header "Authorization: Bearer ${mytoken}"
   http://ig.example.com:8080/home/mgw
   ```

   The home page of the sample application is displayed.

Was this helpful?