Gateway guide

ForgeRock® Identity Platform serves as the basis for our simple and comprehensive Identity and Access Management solution. We help our customers deepen their relationships with their customers, and improve the productivity and connectivity of their employees and partners. For more information about ForgeRock and about the platform, see https://www.forgerock.com.

IG integrates web applications, APIs, and microservices with the ForgeRock Identity Platform. Based on reverse proxy architecture, IG enforces security and access control in conjunction with Access Management modules.

This guide is for access management designers and administrators who develop, build, deploy, and maintain IG for their organizations. It helps you to get started quickly, and learn more as you progress through the guide.

This guide assumes basic 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 IG 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:

- Lightweight Directory Access Protocol (LDAP) if you use IG with LDAP directory services
- Structured Query Language (SQL) if you use IG 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 IG with scripts
- The Java programming language if you plan to extend IG 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:

- IG accessible on `http://ig.example.com:8080` and `https://ig.example.com:8443`, as described in Downloading, starting, and stopping IG.
- Sample application installed on http://app.example.com:8081, as described in Using 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 Identity Cloud and AM for use with IG

This section contains procedures for setting up items in ForgeRock Identity Cloud and AM that you can use with IG. For more information about setting up Identity Cloud, refer to the ForgeRock Identity Cloud docs. For more information about setting up AM, refer to the Access Management docs.

Authenticate an IG agent to Identity Cloud

IMPORTANT

IG agents are automatically authenticated to Identity Cloud by a non-configurable authentication module. Authentication chains and modules are deprecated in Identity Cloud and replaced by journeys.

You can now authenticate IG agents to Identity Cloud with a journey. The procedure is currently optional, but will be required when authentication chains and modules are removed in a future release of Identity Cloud.

For more information, refer to Identity Cloud's Journeys.

This section describes how to create a journey to authenticate an IG agent to Identity Cloud. The journey 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 journey in Identity Cloud, that same journey is used for all instances of IG, Java agent, and Web agent. Consider this point if you change the journey configuration.

1. Log in to the Identity Cloud admin UI as an administrator.
2. Click **Journeys** > **New Journey**.

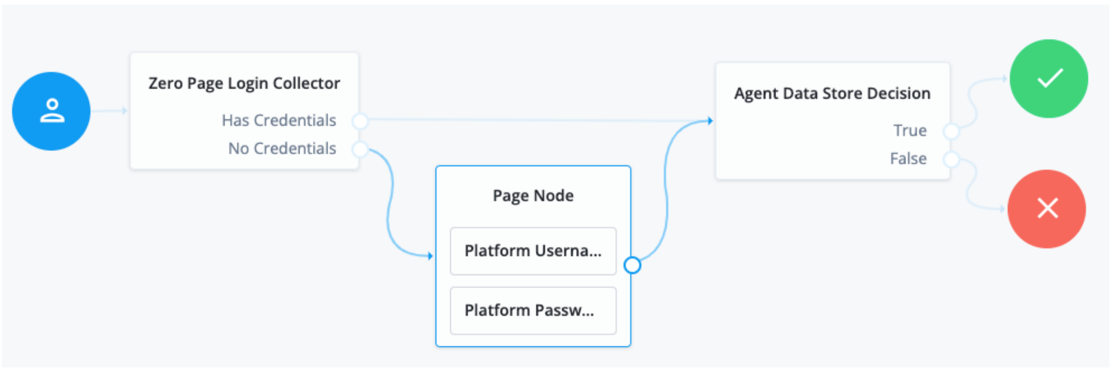
3. Add a journey with the following information and click **Create journey**:
   - **Name**: `Agent`
   - **Identity Object**: The user or device to authenticate.
   - (Optional) **Description**: Authenticate an IG agent to Identity Cloud

   The journey designer is displayed, with the `Start` entry point connected to the `Failure` exit point, and a `Success` node.
4. Using the **Filter nodes** 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 are not 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 IG agent profile.

   IMPORTANT

   Many nodes can be configured in the panel on the right side of the page. Unless otherwise stated, do not configure the nodes, and use only the default values.
5. Drag the following nodes from the **Components** panel into the Page node:
   - Platform Username node to prompt the user to enter their username.
   - Platform Password node to prompt the user to enter their password.
6. Connect the nodes as follows and save the journey:



Authenticate an IG agent to AM

IMPORTANT

IMPORTANT

From AM 7.3

When AM 7.3 is installed with a default configuration, as described in Evaluation, IG is automatically authenticated to AM by an authentication tree. Otherwise, IG 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 are not using the default configuration.

For more information, refer to AM's Authentication Nodes and Trees.

This section describes how to create an authentication tree to authenticate an IG 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 IG, 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:

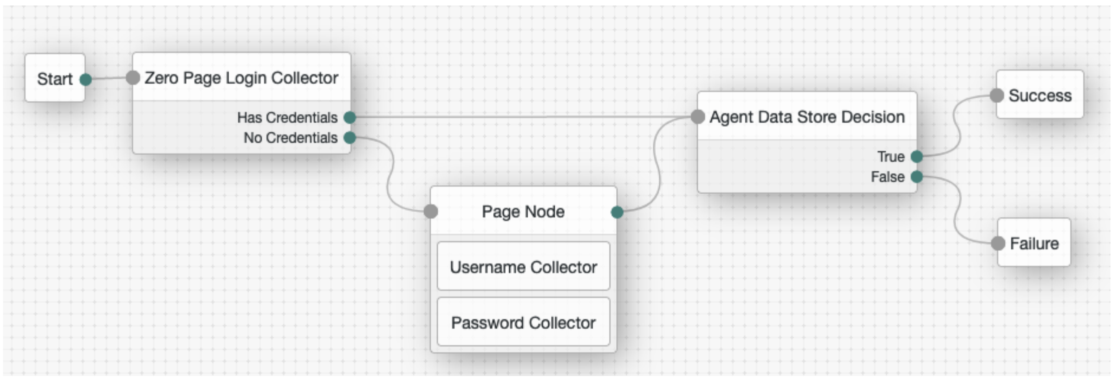
- Zero Page Login Collector node to check whether the agent credentials are provided in the incoming authentication request, and use their values in the following nodes.

  This node is required for compatibility with Java agent and Web agent.
- Page node to collect the agent credentials if they are not 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 IG agent profile.

IMPORTANT

Many nodes can be configured in the panel on the right side of the page. Unless otherwise stated, do not 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 an IG agent in Identity Cloud

This procedure registers an agent that acts on behalf of IG.

1. Log in to the Identity Cloud admin UI as an administrator.
2. Click **Gateways & Agents** > **+ New Gateway/Agent** > **Identity Gateway** > **Next**, and add an agent profile:
   - **ID**: agent-name
   - **Password**: agent-password

IMPORTANT

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

3. Click **Save Profile** > **Done**. The agent profile page is displayed.
4. To add a redirect URL for CDSSO, go to the agent profile page and add the URL.
5. To change the introspection scope, click **Native Consoles** > **Access Management**, and update the agent in the AM admin UI. By default, the agent can introspect OAuth 2.0 tokens issued to any client, in the realm and subrealm where it is created.

Register an IG agent in AM 7 and later

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

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 values:
   1. For SSO
   2. For CDSSO
   - **Agent ID** : `ig_agent`
   - **Password** : `password`
   - **Agent ID** : `ig_agent`
   - **Password** : `password`
   - **Redirect URL for CDSSO** : `https://ig.ext.com:8443/home/cdsso/redirect`

Register an IG agent in AM 6.5 and earlier

In AM 6.5 and earlier versions, follow these steps to register an agent that acts on behalf of IG.

1. In the AM admin UI, select the top-level realm, and then select **Applications** > **Agents** > **Java** (or **J2EE**).
2. Add an agent with the following values:
   1. For SSO

2. For CDSSO
    - **Agent ID** : `ig_agent`
    - **Agent URL** : `http://ig.example.com:8080/agentapp`
    - **Server URL** : `http://am.example.com:8088/openam`
    - **Password** : `password`
    - **Agent ID** : `ig_agent_cdsso`
    - **Agent URL** : `http://ig.ext.com:8080/agentapp`
    - **Server URL** : `http://am.example.com:8088/openam`
    - **Password** : `password`

3. On the **Global** tab, deselect **Agent Configuration Change Notification**.

   This option stops IG from being notified about agent configuration changes in AM. IG doesn't need these notifications.

4. (For CDSSO) On the **SSO** tab, select the following values:
    - **Cross Domain SSO** : Deselect this option
    - **CDSSO Redirect URI** : `/home/cdsso/redirect`

5. (For CDSSO and policy enforcement) On the **SSO** tab, select the following values:
    - **Cross Domain SSO** : Deselect this option
    - **CDSSO Redirect URI** : `/home/pep-cdsso/redirect`

Set up a demo user in Identity Cloud

This procedure sets up a demo user in the alpha realm.

a. Log in to the Identity Cloud admin UI as an administrator.

b. Go to **Identities** > **Manage** > **Alpha realm - Users**, and add a user with the following values:
   - **Username**: `demo`
   - **First name**: `demo`
   - **Last name**: `user`
   - **Email Address**: `demo@example.com`
   - **Password**: `Ch4ng3!t`

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, IG 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/11/tools/keytool.html

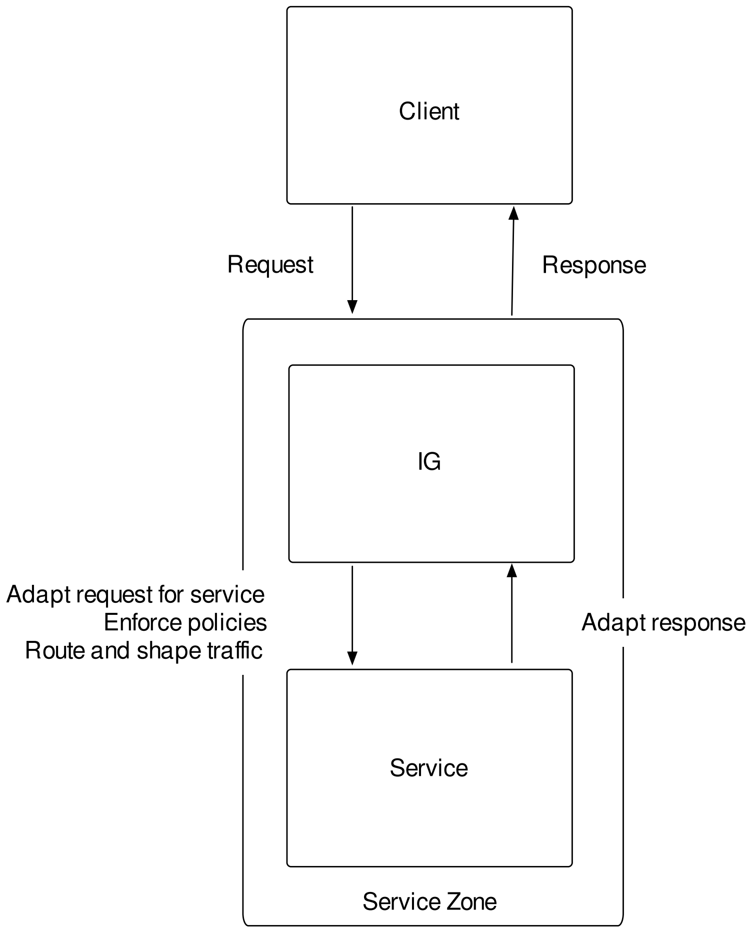
About IG

IG as a proxy

Many organizations have existing services that cannot easily be integrated into newer architectures. Similarly, many existing client applications cannot communicate with services. This section describes how IG acts as an intermediary, or proxy, between clients and services.

IG as a reverse proxy

IG as a reverse proxy server is an intermediate connection point between external clients and internal services. IG intercepts client requests and server responses, enforcing policies, and routing and shaping traffic. The following image illustrates IG as a reverse proxy:

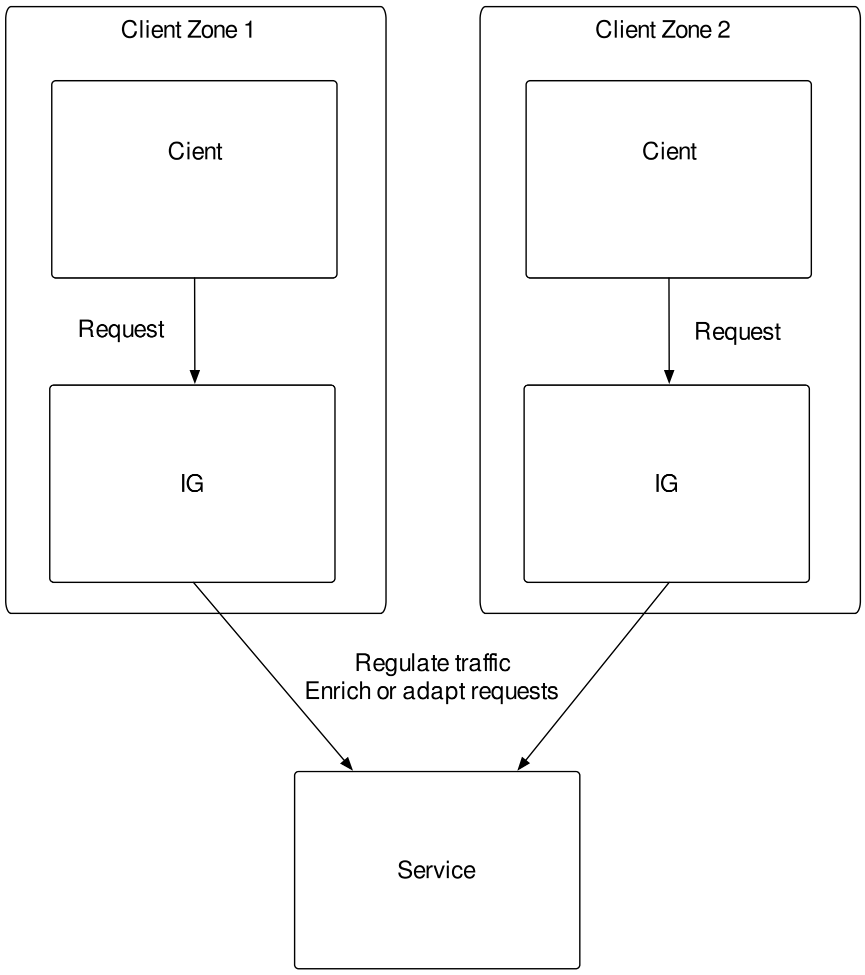


IG provides the following features as a reverse proxy:

- Access management integration
- Application and API security
- Credential replay
- OAuth 2.0 support
- OpenID Connect 1.0 support
- Network traffic control
- Proxy with request and response capture
- Request and response rewriting
- SAML 2.0 federation support
- Single sign-on (SSO)

IG as a forward proxy

In contrast, IG as a forward proxy is an intermediate connection point between an internal client and an external service. IG regulates outbound traffic to the service, and can adapt and enrich requests. The following image illustrates IG as a forward proxy:



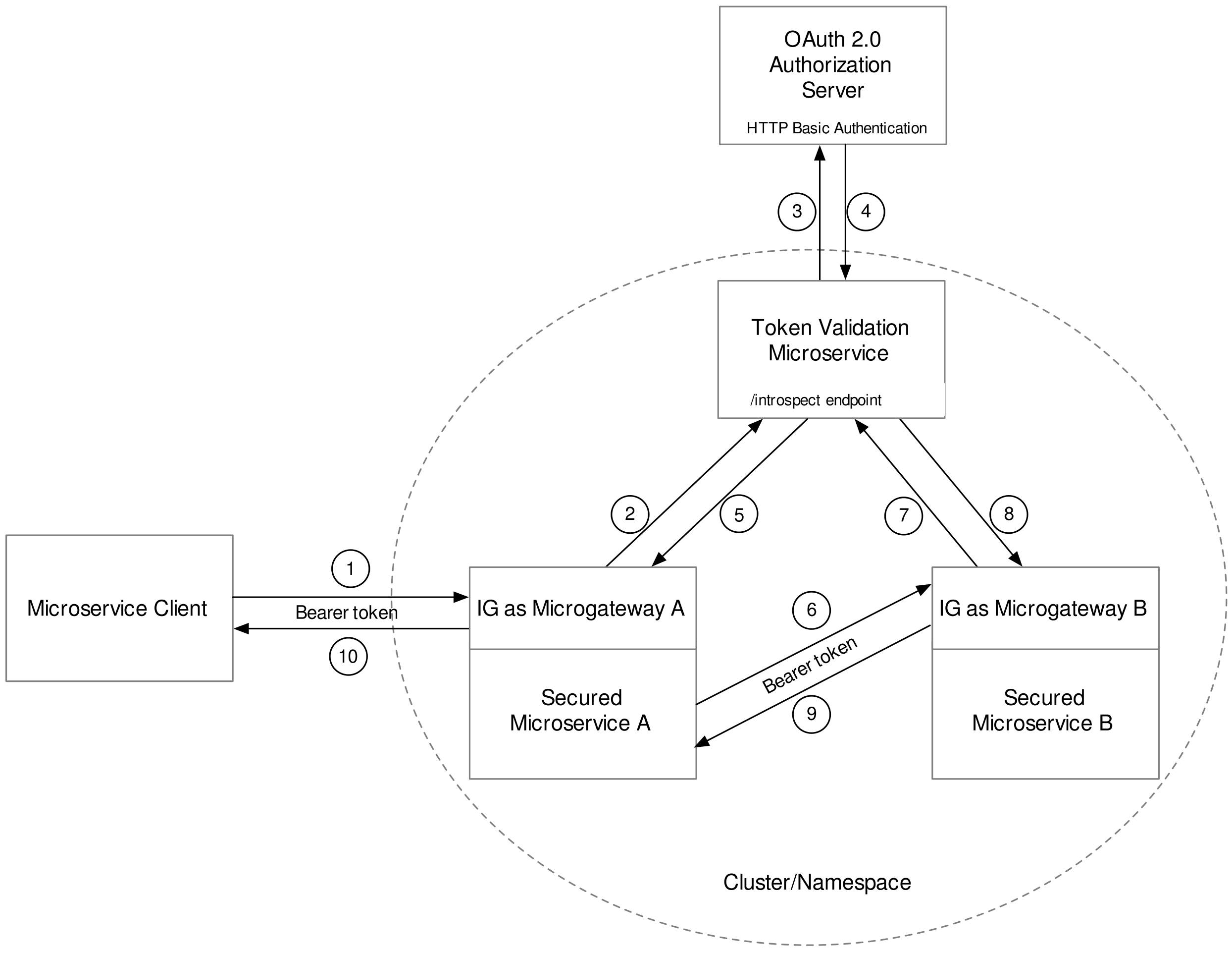
IG provides the following features as a forward proxy:

- Addition of authentication or authorization to the request
- Addition of tracer IDs to the requests
- Addition or removal of request headers or scopes

IG as a microgateway

IG is optimized to run as a microgateway in containerized environments. Use IG with business microservices to separate the security concerns of your applications from their business logic. For example, use IG with the ForgeRock Token Validation Microservice to provide access token validation at the edge of your namespace.

For an example, refer to IG as a microgateway. The following image illustrates the request flow in an example deployment:



The request is processed in the following sequence:

1. A client requests access to Secured Microservice A, providing a stateful OAuth 2.0 access token as credentials.
2. Microgateway A intercepts the request, and passes the access token for validation to the Token Validation Microservice, using the `/introspect` endpoint.
3. The Token Validation Microservice requests the Authorization Server to validate the token.
4. The Authorization Server introspects the token, and sends the introspection result to the Token Validation Microservice.
5. The Token Validation Microservice caches the introspection result, and sends it to Microgateway A, which forwards the result to Secured Microservice A.
6. Secured Microservice A uses the introspection result to decide how to process the request. In this case, it continues processing the request. Secured Microservice A asks for additional information from Secured Microservice B, providing the validated token as credentials.
7. Microgateway B intercepts the request, and passes the access token to the Token Validation Microservice for validation, using the `/introspect` endpoint.
8. The Token Validation Microservice retrieves the introspection result from the cache, and sends it back to Microgateway B, which forwards the result to Secured Microservice B.

9. Secured Microservice B uses the introspection result to decide how to process the request. In this case it passes its response to Secured Microservice A, through Microgateway B.
10. Secured Microservice A passes its response to the client, through Microgateway A.

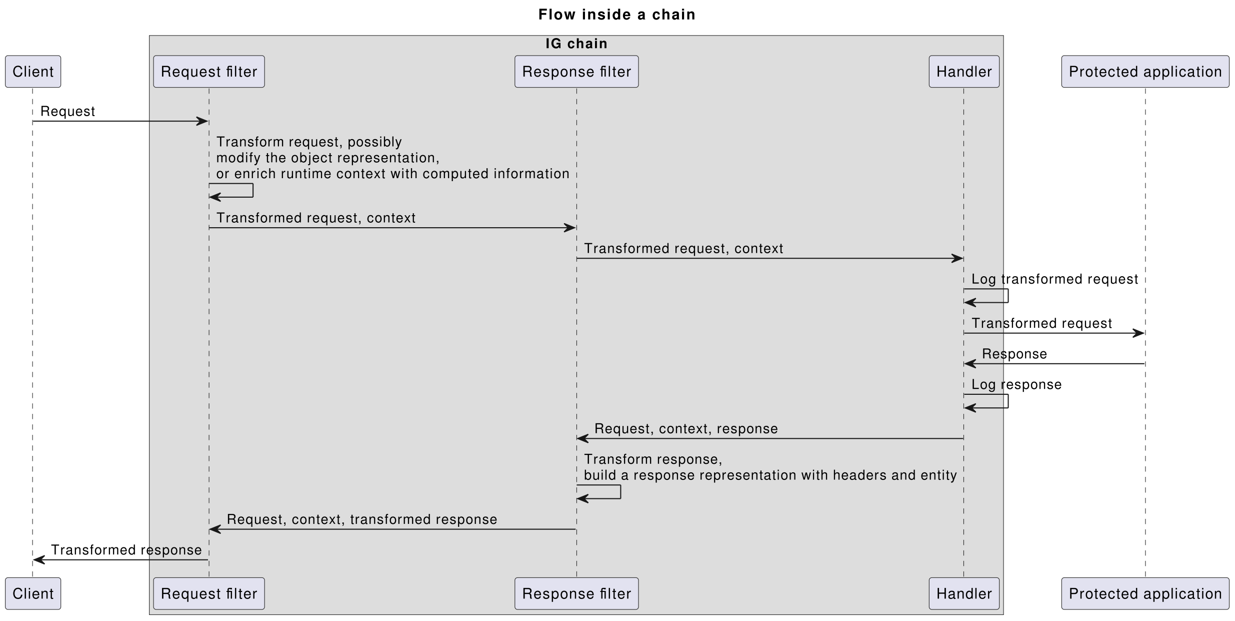
Processing requests and responses

The following sections describe how IG processes requests and responses:

IG object model

IG processes HTTP requests and responses by passing them through user-defined chains of filters and handlers. The filters and handlers provide access to the request and response at each step in the chain, and make it possible to alter the request or response, and collect contextual information.

The following image illustrates a typical sequence of events when IG processes a request and response through a chain:


Flow inside a chain
IG chain
Client
Request filter
Response filter
Handler
Protected application
Request
Transform request, possibly
modify the object representation,
or enrich runtime context with computed information
Transformed request, context
Transformed request, context
Log transformed request
Transformed request
Response
Log response
Request, context, response
Transform response,
build a response representation with headers and entity
Request, context, transformed response
Transformed response
Client
Request filter
Response filter
Handler
Protected application

When IG processes a request, it first builds an object representation of the request, including parsed query/form parameters, cookies, headers, and the entity. IG initializes a runtime context to provide additional metadata about the request and applied transformations. IG then passes the request representation into the chain.

In the request flow, filters modify the request representation and can enrich the runtime context with computed information. In the ClientHandler, the entity content is serialized, and additional query parameters can be encoded as described in RFC 3986: Query.

In the response flow, filters build a response representation with headers and the entity.

The route configuration in Adding headers and logging results demonstrates the flow through a chain to a protected application.

Configuring IG

The way that IG processes requests and responses is defined by the configuration files `admin.json` and `config.json`, and by Route configuration files. For information about the different files used by IG, refer to Configuration directories and files.

Configuration files are flat JSON files that define objects with the following parts:

- `name`: A unique string to identify the object. When you declare inline objects, the name is not required.
- `type`: The type name of the object. IG defines many object types for different purposes.
- `config`: Additional configuration settings for the object. The content of the configuration object depends on its type. For information about each object type available in the IG configuration, refer to the Reference.

  If all of the configuration settings for the type are optional, the `config` field is also optional. The object uses default settings when the `config` field isn't configured, or is configured as an empty object (`"config": {}`), or is configured as null (`"config": null`).

  Filters, handlers, and other objects whose configuration settings are defined by strings, integers, or booleans, can be defined by expressions that match the expected type.

For information about the objects available, refer to admin.json, config.json and Route. An IG route typically contains at least the following parts:

- `handler`: An object to specify the point where the request enters the route. If the handler type is a Chain, the request is dispatched to a list of filters, and then to another handler.

  Handler objects produce a response for a request, or delegate the request to another handler. Filter objects transform data in the request, response, or context, or perform an action when the request or response passes through the filter.
- `baseURI`: A handler decorator to override the scheme, host, and port of the request URI. After a route processes a request, it reroutes the request to the `baseURI`, which most usually points to the application or service that IG is protecting.
- `heap`: A collection of named objects configured in the top level of `config.json` or in individual routes. Heap objects can be configured once and used multiple times in the configuration.

A heap object in a route can be used in that route. A heap object in `config.json` can be used across the whole configuration, unless it is overridden in a route.

- `condition`: An object to define a condition that a request must meet. A route can handle a request if `condition` is not defined, or if the condition resolves to `true`.

Routes inherit settings from their parent configurations. This means that you can configure objects in the `config.json` heap, for example, and then reference those objects by name in any other IG configuration.

Configuration directories and files

By default, IG configuration files are located under `$HOME/.openig` (on Windows `%appdata%\OpenIG`). For information about how to change the location, refer to Change the base location of the IG configuration.

Using comments in IG configuration files

The JSON format does not specify a notation for comments. If IG does not recognize a JSON field name, it ignores the field. As a result, it is possible to use comments in configuration files.

The following conventions are available for commenting:

- A `comment` field to add text comments. The following example includes a text comment.

```
{
  "name": "capture",
  "type": "CaptureDecorator",
  "comment": "Write request and response information to the
logs",
  "config": {
    "captureEntity": true
  }
}
```

- An underscore (`_`) to comment a field temporarily. The following example comments out `"captureEntity": true`, and as a result it uses the default setting (`"captureEntity": false`).

```
{
  "name": "capture",
  "type": "CaptureDecorator",
  "config": {
    "_captureEntity": true
    }
}
```

Development mode and production mode

IG operates in the following modes:

- **Development mode (mutable mode)**

  In development mode, by default all endpoints are open and accessible.

  You can create, edit, and deploy routes through IG Studio, and manage routes through Common REST, without authentication or authorization.

  Use development mode to evaluate or demo IG, or to develop configurations on a single instance. This mode is not suitable for production.

  For information about how to switch to development mode, refer to Switching from production mode to development mode.

  For information about restricting access to Studio in development mode, refer to Restricting access to Studio.

- **Production mode (immutable mode)**

  In production mode, the `/routes` endpoint is not exposed or accessible.

  Studio is effectively disabled, and you cannot manage, list, or even read routes through Common REST.

  By default, other endpoints, such as `/share` and `api/info` are exposed to the loopback address only. To change the default protection for specific endpoints, configure an ApiProtectionFilter in `admin.json` and add it to the IG configuration.

  For information about how to switch to production mode, refer to Switching from development mode to production mode.

After installation, IG is by default in production mode.

Decorators

Decorators are heap objects to extend what another object can do. IG defines `baseURI`, `capture`, and `timer` decorators that you can use without explicitly configuring them. For more information about the types of decorators provided by IG, refer to Decorators.

Decorating objects, the route handler, and the heap

Use decorations that are compatible with the object type. For example, `timer` records the time to process filters and handlers, but does not record information for other object types. Similarly, `baseURI` overrides the scheme, host, and ports, but has no other effect.

In a route, you can decorate individual objects, the route handler, and the heap. IG applies decorations in this order:

1. Decorations declared on individual objects. Local decorations that are part of an object's declaration are inherited wherever the object is used.
2. globalDecorations declared in parent routes, then in child routes, and then in the current route.
3. Decorations declared on the route handler.

Decorating individual objects in a route

To decorate individual objects, add the decorator's name value as a top-level field of the object, next to `type` and `config`.

In this example, the decorator captures all requests going into the SingleSignOnFilter, and all responses coming out of the SingleSignOnFilter:

```
{
  "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": [
        {
          "capture": "all",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Decorating the route handler

To decorate the handler for a route, add the decorator as a top-level field of the route.

In this example, the decorator captures all requests and responses that traverse the route:

```
{
  "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/"
      }
    }
  ],
  "capture": "all",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Decorating the route heap

To decorate all compatible objects in a route, configure globalDecorators as a top-level field of the route. The globalDecorators field takes a map of the decorations to apply.

To decorate all compatible objects declared in `config.json` or `admin.json`, configure globalDecorators as a top-level field in `config.json` or `admin.json`.

In the following example, the route has capture and timer decorations. The capture decoration applies to AmService, Chain, SingleSignOnFilter, and ReverseProxyHandler. The timer decoration doesn't apply to AmService because it is not a filter or handler, but does apply to Chain, SingleSignOnFilter, and ReverseProxyHandler:

```
{
  "globalDecorators":
  {
    "capture": "all",
    "timer": true
  },
  "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": [
        {
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Decorating a named object differently in different parts of the configuration

When a filter or handler is configured in `config.json` or in the heap, it can be used many times in the configuration. To decorate each use of the filter or handler individually, use a Delegate. For more information, refer to Delegate

In the following example, an AmService heap object configures an `amHandler` to delegate tasks to `ForgeRockClientHandler`, and capture all requests and responses passing through the handler.

```
{
  "type": "AmService",
  "config": {
    "agent" : {
      "username" : "ig_agent",
      "passwordSecretId" : "agent.secret.id"
      },
    "secretsProvider": "SystemAndEnvSecretStore-1",
    "amHandler": {
      "type": "Delegate",
      "capture": "all",
      "config": {
        "delegate": "ForgeRockClientHandler"
      }
    },
    "url": "http://am.example.com:8088/openam"
  }
}
```

You can use the same `ForgeRockClientHandler` in another part of the configuration, in a different route for example, without adding a capture decorator. Requests and responses that pass through that use of the handler are not captured.

Decorating IG's interactions with AM

To log interactions between IG and AM, delegate message handling to a ForgeRockClientHandler, and capture the requests and responses passing through the handler. When the ForgeRockClientHandler communicates with an application, it sends ForgeRock Common Audit transaction IDs.

In the following example, the `accessTokenResolver` delegates message handling to a decorated ForgeRockClientHandler:

```
"accessTokenResolver": {
  "name": "token-resolver-1",
  "type": "TokenIntrospectionAccessTokenResolver",
  "config": {
    "amService": "AmService-1",
    "providerHandler": {
      "capture": "all",
      "type": "Delegate",
      "config": {
        "delegate": "ForgeRockClientHandler"
      }
    }
  }
}
```

To try the example, replace the `accessTokenResolver` in the IG route of Validate access tokens through the introspection endpoint. Test the setup as described for the example, and note that the route's log file contains an HTTP call to the introspection endpoint.

Using multiple decorators for the same object

Decorations can apply more than once. For example, if you set a decoration on a route and another decoration on an object defined within the route, IG applies the decoration twice. In the following route, the request is captured twice:

```
{
 "handler": {
   "type": "ReverseProxyHandler",
   "capture": "request"
 },
 "capture": "all"
}
```

When an object has multiple decorations, the decorations are applied in the order they appear in the JSON.

In the following route, the handler is decorated with a `baseURI` first, and a `capture` second:

```
{
  "name": "myroute",
  "baseURI": "http://app.example.com:8081",
  "capture": "all",
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "headers": {
        "Content-Type": [ "text/plain; charset=UTF-8" ]
      },
      "entity": "Hello world, from myroute!"
    }
  },
  "condition": "${find(request.uri.path, '^/myroute1')}"
}
```

The decoration can be represented as `capture[ baseUri[ handler ] ]`. When a request is processed, it is captured, and then rebased, and then processed by the handler: The log for this route shows that the capture occurs before the rebase:

```
2018-09-10T13:23:18,990Z | INFO  | http-nio-8080-exec-1 |
o.f.o.d.c.C.c.top-level-handler | @myroute |

--- (request) id:f792d2ad-4409-4907-bc46-28e1c3c19ac3-7 --->

GET http://ig.example.com:8080/myroute HTTP/1.1
...
```

Conversely, in the following route, the handler is decorated with a `capture` first, and a `baseURI` second:

```
{
  "name": "myroute",
  "capture": "all",
  "baseURI": "http://app.example.com:8081",
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "headers": {
        "Content-Type": [ "text/plain; charset=UTF-8" ]
      }
      "entity": "Hello, world from myroute1!"
    }
  },
  "condition": "${find(request.uri.path, '^/myroute')}"
}
```

The decoration can be represented as `baseUri[ capture[ handler ] ]`. When a request is processed, it is rebased, and then captured, and then processed by the handler. The log for this route shows that the rebase occurs before the capture:

```
2018-09-10T13:07:07,524Z | INFO  | http-nio-8080-exec-1 |
o.f.o.d.c.C.c.top-level-handler | @myroute |

--- (request) id:3c26ab12-3cc0-403e-bec6-43bf5621f657-7 --->

GET http://app.example.com:8081/myroute HTTP/1.1
...
```

Guidelines for naming decorators

To prevent unwanted behavior, consider the following points when you name decorators:

- Avoid decorators named `comment` or `comments`, and avoid reserved field names. Instead of using alphanumeric field names, consider using dots in your decorator names, such as `my.decorator`.
- For heap objects, avoid the reserved names `config`, `name`, and `type`.
- For routes, avoid the reserved names `auditService`, `baseURI`, `condition`, `globalDecorators`, `heap`, `handler`, `name`, `secrets`, and `session`.
- In `config.json`, avoid the reserved name `temporaryStorage`.

Configuration parameters declared as property variables

Configuration parameters, such as host names, port numbers, and directories, can be declared as property variables in the IG configuration or in an external JSON file. The variables can then be used in expressions in routes and in `config.json` to set the value of configuration parameters.

Properties can be inherited across the router, so a property defined in `config.json` can be used in any of the routes in the configuration.

Storing the configuration centrally and using variables for parameters that can be different for each installation makes it easier to deploy IG in different environments without changing a single line in your route configuration.

For more information, refer to Route properties.

Changing the configuration and restarting IG

You can change routes or change a property that is read at runtime or that relies on a runtime expression without needing to restart IG to take the change into account.

Stop and restart IG only when you make the following changes:

- Change the configuration of any route, when the `scanInterval` of Router is `disabled` (see Router).
- Add or change an external object used by the route, such as an environment variable, system property, external URL, or keystore.
- Add or update `config.json` or `admin.json`.

Understanding IG APIs with API descriptors

Common REST endpoints in IG serve API descriptors at runtime. When you retrieve an API descriptor for an endpoint, a JSON that describes the API for that endpoint is returned.

To help you discover and understand APIs, 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 different endpoints.

When you start IG, or add or edit routes, registered endpoint locations for the routes hosted by the main router are written in `$HOME/.openig/logs/route-system.log`, where `$HOME/.openig` is the instance directory. Endpoint locations for subroutes are written to other log files. To retrieve the API descriptor for a specific endpoint, append one of the following query string parameters to the endpoint:

- `_api`, to represent the API accessible over HTTP. This OpenAPI descriptor can be used with endpoints that are complete or partial URLs.

The returned JSON respects the OpenAPI specification and can be consumed by Swagger tools, such as Swagger UI.

- `_crestapi`, to provide a compact representation that is independent of the transport protocol. This ForgeRock® Common REST (Common REST) API descriptor cannot be used with partial URLs.

  The returned JSON respects a ForgeRock proprietary specification dedicated to describe Common REST endpoints.

For more information about Common REST API descriptors, refer to Common REST API Documentation.

Retrieving API descriptors for a router

With IG running as described in the Getting started, run the following query to generate a JSON that describes the router operations supported by the endpoint:

```
$ curl
http://ig.example.com:8080/openig/api/system/objects/_router/r
outes?_api

{
     "swagger": "2.0",
     "info": {
     "version": "IG version",
     "title": "IG"
     },
     "host": "0:0:0:0:0:0:0:1",
     "basePath": "/openig/api/system/objects/_router/routes",
     "tags": [{
     "name": "Routes Endpoint"
     }],
     . . .
```

Alternatively, generate a Common REST API descriptor by using the `?_crestapi` query string.

Retrieving API descriptors for the UMA service

With the UMA tutorial running as described in UMA support, run the following query to generate a JSON that describes the UMA share API:

```
$ curl
http://ig.example.com:8080/openig/api/system/objects/_router/r
outes/00-uma/objects/umaservice/share?_api
```

```
{
     "swagger": "2.0",
     "info": {
     "version": "IG version",
     "title": "IG"
     },
     "host": "0:0:0:0:0:0:0:1",
     "basePath":
"/openig/api/system/objects/_router/routes/00-
uma/objects/umaservice/share",
     "tags": [{
     "name": "Manage UMA Share objects"
     }],
     . . .
```

Alternatively, generate a Common REST API descriptor by using the `?_crestapi` query string.

Retrieving API descriptors for the main router

Run a query to generate a JSON that describes the API for the main router and its subsequent endpoints. For example:

```
$ curl
http://ig.example.com:8080/openig/api/system/objects/_router?
_api

{
     "swagger": "2.0",
     "info": {
     "version": "IG version",
     "title": "IG"
     },
     "host": "ig.example.com:8080",
     "basePath": "/openig/api/system/objects/_router",
     "tags": [{
     "name": "Monitoring endpoint"
     }, {
     "name": "Manage UMA Share objects"
     }, {
     "name": "Routes Endpoint"
     }],
     . . .
```

Because the above URL is a partial URL, you cannot use the `?_crestapi` query string to generate a Common REST API descriptor.

Retrieving API descriptors for an IG instance

Run a query to generate a JSON that describes the APIs provided by the IG instance that is responding to a request. For example:

```
$ curl http://ig.example.com:8080/openig/api?_api

{
     "swagger": "2.0",
     "info": {
     "version": "IG version",
     "title": "IG"
     },
     "host": "ig.example.com:8080",
     "basePath": "/openig/api",
     "tags": [{
     "name": "Internal Storage for UI Models"
     }, {
     "name": "Monitoring endpoint"
     }, {
     "name": "Manage UMA Share objects"
     }, {
     "name": "Routes Endpoint"
     }, {
     "name": "Server Info"
     }],
     . . .
```

If routes are added after the request is performed, they are not included in the returned JSON.

Because the above URL is a partial URL, you cannot use the `?_crestapi` query string to generate a Common REST API descriptor.

Sessions

IG uses sessions to group requests from a user agent or other source, and collect information from the requests. When multiple requests are made in the same session, the requests can share the session information. Because session sharing is not thread-safe, it is not suitable for concurrent exchanges.

The following table compares stateful and stateless sessions:

Feature	Stateful sessions	Stateless sessions
Cookie size.	Unlimited.	Maximum size of the JWT session cookie is 4 KBytes, as defined by the browser. If the cookie exceeds this size, IG automatically splits it into multiple cookies.
Default name of the session cookie.	`IG_SESSIONID`.	`openig-jwt-session`.
Object types that can be stored in the session.	Only Java serializable objects, when sessions are replicated. Any object, when sessions are not replicated.	JSON-compatible types, such as strings, numbers, booleans, null, structures such as arrays, and list and maps containing only JSON-compatible types.
Session sharing between instances of IG, for load balancing and failover.	Possible when sessions are replicated on multiple IG instances. Possible when sessions are not replicated, if session stickiness is configured.	Possible because the session content is a cookie on the user agent, that can be copied to multiple instances of IG.
Risk of data inconsistency when simultaneous requests modify the content of a session.	Low because the session content is stored on IG and shared by all exchanges. Processing is not thread-safe.	Higher because the session content is reconstructed for each request. Concurrent exchanges don't have the same content.

Stateful sessions

When a JwtSession is not configured for a request, stateful sessions are created automatically. Session information is stored in the IG cookie, called `IG_SESSIONID` by default. When the user agent sends a request with the cookie, the request can access the session information on IG.

When a JwtSession object is configured in the route that processes a request, or in its ascending configuration (a parent route or `config.json`), the session is always stateless and can't be stateful.

When a request enters a route without a JwtSession object in the route or its ascending configuration, a stateful session is created lazily.

Session duration is defined by the `session` property in admin.json, with a default of 30 minutes.

Even if the session is empty, the session remains usable until the timeout.

When IG is not configured for session replication, any object type can be stored in a stateful session.

Because session content is stored on IG, and shared by all exchanges, when IG processes simultaneous requests in a stateful session there is low risk that the data becomes inconsistent. However, sessions are not thread-safe; different requests can simultaneously read and modify a shared session.

Session information is available in SessionContext to downstream handlers and filters. For more information, refer to SessionContext.

Considerations for clustering IG

When a stateful session is replicated on the multiple IG instances, consider the following points:

- The session can store only object types that can be serialized.
- The network latency of session replication introduces a delay that can cause the session information of two IG instances to desynchronize.
- Because the session is replicated on the clustered IG instances, it can be shared between the instances, without configuring session stickiness.
- When sessions are not shared, configure session stickiness to ensure that load balancers serve requests to the same IG instance. For more information, refer to Prepare for load balancing and failover.

Configuring stateful sessions

To configure stateful sessions, update the `session` property of admin.json.

Stateless sessions

Stateless sessions are provided when a JwtSession object is configured in `config.json` or in a route. For more information about configuring stateless sessions, refer to JwtSession.

IG serializes stateless session information as JSON, stores it in a JWT that can be encrypted and then signed, and places the JWT in a cookie. The cookie contains all of the information about the session, including the session attributes as JSON, and a marker for the session timeout.

The maximum size of the JWT session cookie is 4 KBytes, as defined by the browser. If the cookie exceeds this size, IG automatically splits it into multiple cookies.

Only JSON-compatible object types can be stored in stateless sessions. These object types include strings, numbers, booleans, null, structures such as arrays, and list and maps containing only JSON-compatible types.

Stateless sessions are managed as follows:

- When a request enters a route with a JwtSession object in the route or its ascending configuration, IG creates the SessionContext, verifies the cookie signature, decrypts the content of the cookie, and checks that the current date is before the session timeout.
- When the request passes through the filters and handlers in the route, the request can read and modify the session content.
- When the request returns to the the point where the session was created, for example, at the entrance to a route or at `config.json`, IG updates the cookie as follows:
    - If the session content has changed, IG serializes the session, creates one or more new JWT session cookies with the new content, encrypts and then signs the cookies, assigns them an appropriate expiration time, and returns them in the response.
    - If the session is empty, IG deletes the session, creates a new JWT session cookie with an expiration time that has already passed, and returns the cookie in the response.
    - If the session content has not changed, IG does nothing.

Because the session content is stored in one or more JWT session cookies on the user agent, stateless sessions can be shared easily between IG instances. The cookies are automatically carried over in requests, and any IG instance can unpack and use the session content.

When IG processes simultaneous requests in stateless sessions, there is a high risk that the data becomes inconsistent. This is because the session content is reconstructed for each exchange, and concurrent exchanges don’t have the same content.

IG does not share sessions across requests. Instead, each request has its own session objects that it modifies as necessary, writing its own session to the session cookie regardless of what other requests do.

Session information is available in SessionContext to downstream handlers and filters. For more information, refer to SessionContext.

Secrets

IG uses the ForgeRock Commons Secrets API to manage secrets, such as passwords and cryptographic keys.

Repositories of secrets are managed through secret stores, provided to the configuration by the `SecretsProvider` object or `secrets` object. For more information about these objects and the types of secret stores provided in IG, refer to SecretsProvider and Secrets object and secret stores.

Secret types

IG uses two secret types:

- **`GenericSecret`**: An opaque blob of bytes, such as a password or API key, without any metadata. A `GenericSecret` cannot be used to perform cryptographic operations.
- **`CryptoKey`**: A secret that contains either a private or shared key, and/or a public certificate. A `CryptoKey` contains the secret material itself and its metadata; for example, the associated algorithm or key type. This secret type can be used for cryptographic operations.

For example:

- A `Base64EncodedSecretStore` can only serve secrets of the `GenericSecret` type.
- An `HsmSecretStore` can only server secrets of the `CryptoKey` type.
- A `FileSystemSecretStore` can serve secrets of both types.

To learn more about secret store specificities, refer to Secret Stores.

Secret terminology

The following terms are used to describe secrets:

- **Secret ID**: A label to indicate the purpose of a secret. A secret ID is generally associated with one or more aliases of a key in a keystore or HSM.
- **Stable ID**: A label to identify a secret. The stable ID corresponds to the following values in each type of secret store:
  - Base64EncodedSecretStore: The value of `secret-id` in the `"secret-id": "string"` pair.

    - FileSystemSecretStore: The filename of a file in the specified directory, without the prefix/suffix defined in the store configuration.
    - HsmSecretStore: The value of an `alias` in a `secret-id` / `aliases` mapping.
    - JwkSetSecretStore: The value of the `kid` of a JWK stored in a JwkSetSecretStore.
    - KeyStoreSecretStore: The value of an `alias` in a `secret-id` / `aliases` mapping.
    - SystemAndEnvSecretStore: The name of a system property or environment. variable
- **Valid secret**: A secret whose purpose matches the secret ID **and** any purpose constraints. Constraints can include requirements for the following:
    - Secret type, such as signing key or encryption key
    - Cryptographic algorithm, such as Diffie-Hellman and RSA
    - Signature algorithm, such as ES256 and ES384

  Constraints are defined when the secret is generated, and cannot be added after.
- **Named secret**: A valid secret that a secret store can find by using a secret ID and stable ID.
- **Active secret**: One of the valid secrets that is considered eligible at the time of use. The way that the active secret is chosen is determined by the type of secret store. For more information, refer to Secrets object and secret stores,

Using keys and certificates

The examples in this doc set use self-signed certificates, but your deployment is likely to use certificates issued by a certificate authority (CA certificates).

The way to obtain CA certificates depends on the certificate authority that you are using, and is not described in this document. As an example, refer to Let's Encrypt.

Integrate CA certificates by using secret stores:

- For PEM files, use a FileSystemSecretStore and PemPropertyFormat
- For PKCS12 keystores, use a KeyStoreSecretStore

For examples, refer to Serve the same certificate for TLS connections to all server names.

Note the following points about using secrets:

- When IG starts up, it listens for HTTPS connections, using the ServerTlsOptions configuration in `admin.json`. The keys and certificates are fetched at startup.
- Keys and certificates must be present at startup.
- If keys or certificates change, you must to restart IG.

- When the `autoRefresh` property of FileSystemSecretStore or KeyStoreSecretStore is enabled, the secret store is automatically reloaded when the filesystem or keystore is changed.

For information about secret stores provided in IG, refer to Secrets object and secret stores.

Validating the signature of signed tokens

IG validates the signature of signed tokens as follows:

- Named secret resolution:
  - If the JWT contains a `kid`, IG queries the secret stores declared in `secretsProvider` or `secrets` to find a named secret, identified by a secret ID and stable ID.
  - If a named secret is found, IG then uses the named secret to try to validate the signature. If the named secret can't validate the signature, the token is considered as invalid.
  - If a named secret isn't found, IG tries valid secret resolution.
- Valid secret resolution:
  - IG uses the value of `verificationSecretId` as the secret ID, and queries the declared secret stores to find all secrets that match the provided secret ID.
  - All matching secrets are returned as valid secrets, in the order that the secret stores are declared, and for KeyStoreSecretStore and HsmSecretStore, in the order defined by the mappings.
  - IG tries to verify the signature with each valid secret, starting with the first valid secret, and stopping when it succeeds.
  - If no valid secrets are returned, or if none of the valid secrets can verify the signature, the token is considered as invalid.

For examples where a StatelessAccessTokenResolver uses a secret store to validate the signature of signed tokens, refer to the example sections of JwkSetSecretStore and KeyStoreSecretStore.

Using multiple secret stores in a configuration

When multiple secrets stores are provided in a configuration, the secrets stores are queried in the following order:

- Locally in the route, starting with the first secret store in the list, up to the last.
- In ascending parent routes, starting with the first secret store in each list, up to the last.

- In `config.json`, starting with the first secret store in the list, up to the last.
- If a secrets store is not configured in `config.json`, the secret is queried in a default SystemAndEnvSecretStore, and a base64-encoded value is expected.
- If a secret is not resolved, an error is produced.

Secrets stores defined in `admin.json` can be accessed only by heap objects in `admin.json`.

Algorithms for elliptic curve digital signatures

When the Elliptic Curve Digital Signature Algorithm (ECDSA) is used for signing, and both of the following conditions are met, JWTs are signed with a deterministic ECDSA:

- Bouncy Castle is installed.
- The system property `org.forgerock.secrets.preferDeterministicEcdsa` is `true`, which is its default value.

Otherwise, when ECDSA is used for signing, JWTs are signed with a non-deterministic ECDSA.

A non-deterministic ECDSA signature can be verified by the equivalent deterministic algorithm.

For information about deterministic ECDSA, refer to RFC 6979. For information about Bouncy Castle, refer to The Legion of the Bouncy Castle.

Routers and routes

The following sections provide an overview of how IG uses routers and routes to handle requests and their context. For information about creating routes in Studio, refer to the Studio guide.

Configure routers

The following `config.json` file configures a Router:

```
{
  "handler": {
    "type": "Router",
    "name": "_router",
    "baseURI": "http://app.example.com:8081",
    "capture": "all"
  },
  "heap": [
    {
      "name": "JwtSession",
      "type": "JwtSession"
    },
    {
      "name": "capture",
      "type": "CaptureDecorator",
      "config": {
        "captureEntity": true,
        "_captureContext": true
      }
    }
  ]
}
```

In this configuration, all requests are passed with the default settings to the Router. The Router scans `$HOME/.openig/config/routes` at startup, and rescans the directory every 10 seconds. If routes have been added, deleted, or changed, the router applies the changes.

The main router and any subrouters are used to build the monitoring endpoints. For information about monitoring endpoints, refer to Monitoring endpoints. For information about the parameters of a router, refer to Router.

Configure routes

Routes are JSON configuration files that handle requests and their context, and then hand off any request they accept to a handler. Another way to think of a route is like an independent dispatch handler, as described in DispatchHandler.

Routes can have a base URI to change the scheme, host, and port of the request.

For information about the parameters of routes, refer to Route.

Configure objects inline or in the heap

If you use an object only once in the configuration, you can declare it inline in the route and do not need to name it. However, when you need use an object multiple times, declare it in the heap, and then reference it by name in the route.

The following route shows an inline declaration for a handler. The handler is a router to route requests to separate route configurations:

```
{
  "handler": {
    "type": "Router"
  }
}
```

The following example shows a named router in the heap, and a handler references the router by its name:

```
{
  "handler": "My Router",
  "heap": [
    {
      "name": "My Router",
      "type": "Router"
    }
  ]
}
```

Notice that the heap takes an array. Because the heap holds all configuration objects at the same level, you can impose any hierarchy or order when referencing objects. Note that when you declare all objects in the heap and reference them by name, neither hierarchy nor ordering are obvious from the structure of the configuration file alone.

Set route conditions

When a route has a condition, it can handle only requests that meet the condition. When a route has no condition, it can handle any request.

A condition can be based on a characteristic of the request, context, or IG runtime environment, such as system properties or environment variables. Conditions are defined using IG expressions, as described in Expressions.

Because routes define the conditions on which they accept a request, the router does not have to know about specific routes in advance. In other words, you can configure the router first and then add routes while IG is running.

The following example shows a route with no condition. This route accepts any request:

```
{
  "name": "myroute",
  "handler": {
    "type": "ReverseProxyHandler"
  }
}
```

The following example shows the same route with a condition. This route accepts only requests whose path starts with `mycondition`:

```
{
  "name": "myroute",
  "handler": {
    "type": "ReverseProxyHandler"
  },
  "condition": "${find(request.uri.path, '^/mycondition')}"
}
```

The following table lists some of the conditions used in routes in this guide:

Example conditions and requests

Condition	Requests that meet the condition
`"${true}"`	All requests, because this expression always evaluates to `true`.
`"${find(request.uri.path, '^/login')}"`	`http://app.example.com/login`, ...
`"${request.uri.host == 'api.example.com'}"`	`http://api.example.com/`, `https://api.example.com/home`, `http://api.example.com:8080/home`, ...
`"${find(contexts.client.remoteAddress, '127.0.0.1')}"`	`http://localhost:8080/keygen`, `http://127.0.0.1:8080/keygen`, ... Where `/keygen` is not mandatory and could be anything else.
`"${find(request.uri.query, 'demo=simple')}"`	`http://ig.example.com:8080/login?demo=simple`, ... For information about URI query, refer to `query` in URI.
`"${request.uri.scheme == 'http'}"`	`http://ig.example.com:8080`, ...

Condition	Requests that meet the condition
`"${find(request.uri.path, '^/dispatch') or find(request.uri.path, '^/mylogin')}"`	`http://ig.example.com:8080/dispatch` , `http://ig.example.com:8080/mylogin`, ...
`"${request.uri.host == 'sp1.example.com' and not find(request.uri.path, '^/saml')}"`	`http://sp1.example.com:8080/`, `http://sp1.example.com/mypath`,... Not `http://sp1.example.com:8080/saml`, `http://sp1.example.com/saml`,...
`"condition": "${find (request.uri.path, '&{uriPath}')}"`	`http://ig.example.com:8080/hello`, when the following property is configured: `{ "properties": { "uriPath": "hello" } }` For information about including properties in the configuration, refer to Route properties.
`"condition": "${request.headers['X-Forwarded-Host'][0] == 'service.example.com'}"`	Requests with the header `X-Forwarded-Host`, whose first value is `service.example.com`.

Configure route names, IDs, and filenames

The filenames of routes have the extension `.json`, in lowercase.

The Router scans the routes folder for files with the `.json` extension, and uses the route's `name` property to order the routes in the configuration. If the route does not have a `name` property, the Router uses the route ID.

The route ID is managed as follows:

- When you add a route manually to the routes folder, the route ID is the value of the `_id` field. If there is no `_id` field, the route ID is the filename of the added route.
- When you add a route through the Common REST endpoint, the route ID is the value of the mandatory `_id` field.
- When you add a route through Studio, you can edit the default route ID.

CAUTION

The filename of a route cannot be `default.json`, and the route's `name` property and route ID cannot be `default`.

Create and edit routes through Common REST

NOTE

When IG is in production mode, you cannot manage, list, or even read routes through Common REST. For information about switching to development mode, refer to Switching from production mode to development mode.

Through Common REST, you can read, add, delete, and edit routes on IG without manually accessing the file system. You can also list the routes in the order that they are loaded in the configuration, and set fields to filter the information about the routes.

The following examples show some ways to manage routes through Common REST. For more information, refer to About ForgeRock Common REST.

Manage routes through common REST

Before you start, prepare IG as described in the Getting started.

1. Add the following route to IG:
    1. Linux
    2. Windows

```
$HOME/.openig/config/routes/00-crest.json
```

```
%appdata%\OpenIG\config\routes\00-crest.json
```

```
{
  "name": "crest",
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "headers": {
        "Content-Type": [ "text/plain; charset=UTF-8" ]
      },
      "entity": "Hello world!"
    }
  },
  "condition": "${find(request.uri.path, '^/crest')}"
}
```

To check that the route is working, access the route on: http://ig.example.com:8080/crest.

2. To read a route through Common REST:

   a. Enter the following command in a terminal window:

   ```
   $ curl -v
   http://ig.example.com:8080/openig/api/system/objects/_r
   outer/routes/00-crest\?_prettyPrint\=true
   ```

   The route is displayed. Note that the route `_id` is displayed in the JSON of the route.

3. To add a route through Common REST:

   a. Move `$HOME/.openig/config/routes/00-crest.json` to `/tmp/00-crest.json`.

   b. Check in `$HOME/.openig/logs/route-system.log` that the route has been removed from the configuration, where `$HOME/.openig` is the instance directory. To double check, go to http://ig.example.com:8080/crest. You should get an HTTP 404 error.

   c. Enter the following command in a terminal window:

   ```
   $ curl -X PUT
   http://ig.example.com:8080/openig/api/system/objects/_r
   outer/routes/00-crest \
          -d "@/tmp/00-crest.json" \
          --header "Content-Type: application/json"
   ```

   This command posts the file in `/tmp/00-crest.json` to the `routes` directory.

   d. Check in `$HOME/.openig/logs/route-system.log` that the route has been added to configuration, where `$HOME/.openig` is the instance directory. To double-check, go to http://ig.example.com:8080/crest. You should see the "Hello world!" message.

4. To edit a route through Common REST:

   a. Edit `/tmp/00-crest.json` to change the message displayed by the response handler in the route.

   b. Enter the following command in a terminal window:

   ```
   $ curl -X PUT
   http://ig.example.com:8080/openig/api/system/objects/_r
   outer/routes/00-crest \
          -d "@/tmp/00-crest.json" \
          --header "Content-Type: application/json" \
          --header "If-Match: *"
   ```

   This command deploys the route with the new configuration. Because the changes are persisted into the configuration, the existing `$HOME/.openig/config/routes/00-crest.json` is replaced with the edited version in `/tmp/00-crest.json`.

   c. Check in `$HOME/.openig/logs/route-system.log` that the route has been updated, where `$HOME/.openig` is the instance directory. To double-check, go to http://ig.example.com:8080/crest to confirm that the displayed message has changed.

5. To delete a route through Common REST:

   a. Enter the following command in a terminal window:

   ```
   $ curl -X DELETE
   http://ig.example.com:8080/openig/api/system/objects/_r
   outer/routes/00-crest
   ```

   b. Check in `$HOME/.openig/logs/route-system.log` that the route has been removed from the configuration, where `$HOME/.openig` is the instance directory. To double-check, go to http://ig.example.com:8080/crest. You should get an HTTP 404 error.

6. To list the routes deployed on the router, in the order that they are tried by the router:

   a. Enter the following command in a terminal window:

   ```
   $ curl
   "http://ig.example.com:8080/openig/api/system/objects/_
   router/routes?_queryFilter=true"
   ```

   The list of loaded routes is displayed.

Prevent the reload of routes

To prevent routes from being reloaded after startup, stop IG, edit the router `scanInterval`, and restart IG. When the interval is set to `disabled`, routes are loaded only at startup:

```
{
  "name": "Router",
  "type": "Router",
  "config": {
    "scanInterval": "disabled"
  }
}
```

The following example changes the location where the router looks for the routes:

```
{
  "name": "Router",
  "type": "Router",
  "config": {
    "directory": "/path/to/safe/routes",
    "scanInterval": "disabled"
  }
}
```

Access reserved routes

IG uses an `ApiProtectionFilter` to protect the reserved routes. By default, the filter allows access to reserved routes only from the loopback address. To override this behavior, declare a custom `ApiProtectionFilter` in the top-level heap. For an example, refer to the CORS filter described in Set up the UMA example.

Authentication

Single sign-on (SSO)

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

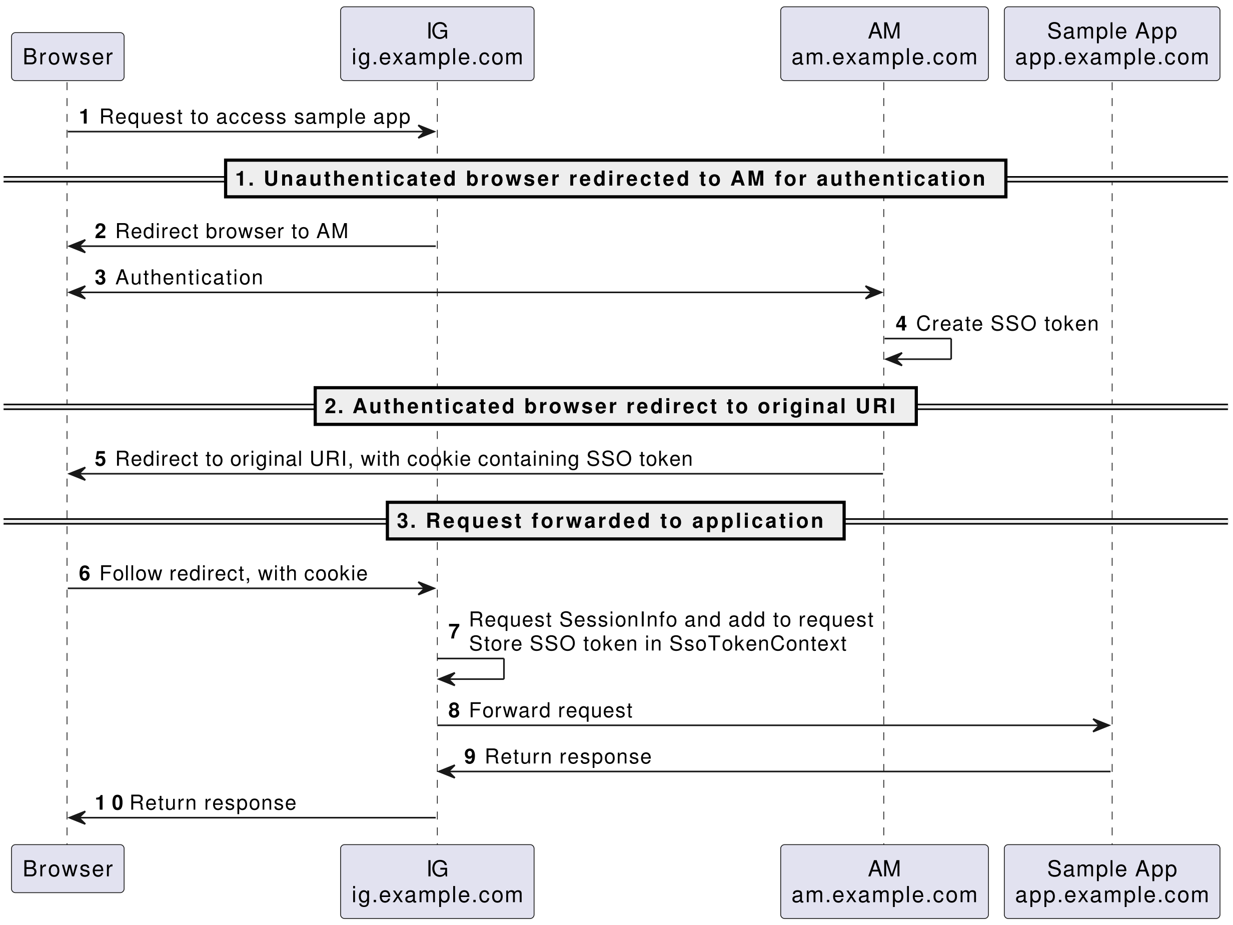
IMPORTANT

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 users to authenticate to a specific realm.

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

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


Browser
IG ig.example.com
AM 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 app.
- IG 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 IG.
- IG validates the token it gets from the cookie. It then adds the AM session info to the request, and stores the SSO token in the context for use by downstream filters and handlers.
- IG forwards the request to the sample app, and the sample app returns the requested resource to the browser.

SSO through the default AM authentication tree

This section gives an example of how to authenticate by using SSO and the default authentication service provided in AM.

Before you start, prepare AM, IG, and the sample application as described in Example installation for this guide.

1. Set up AM:

   a. (From AM 6.5.3) 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/*?*`

   b. Select **Applications** > **Agents** > **Identity Gateway** and register an IG agent with the following values:

      - **Agent ID**: `ig_agent`
      - **Password**: `password`

        For AM 6.5.x and earlier versions, register an agent as described in Register an IG agent in AM 6.5 and earlier.

        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 an IG agent to AM.

      IMPORTANT

      IG 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 IG:

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

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

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

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

1. Linux
2. Windows

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

```
%appdata%\OpenIG\config\routes\static-resources.json
```

```
{
  "name" : "sampleapp-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css')}",
  "handler": "ReverseProxyHandler"
}
```

c. Add the following route to IG:

1. Linux
2. Windows

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

```
%appdata%\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 IG route in Studio, refer to Policy enforcement in Structured Editor or Protecting a web app with Freeform Designer.

3. Test the setup:

   a. If you are logged in to AM, log out and clear any cookies.

   b. Go to http://ig.example.com:8080/home/sso, and log in to AM as user `demo`, password `Ch4ng31t`.

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

SSO through a specified AM authentication tree

This section gives an example of how to authenticate by using SSO and the example authentication tree provided in AM, instead of the default authentication tree.

1. Set up the example in Authenticate with SSO through the default authentication service.
2. Add the following route to IG:
    1. Linux
    2. Windows

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

```
%appdata%\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. If you are logged in to AM, log out and clear any cookies.

   b. Go to http://ig.example.com:8080/home/sso-authservice, and note that the login page is different to that returned in Authenticate with SSO through the default authentication service.

Cross-domain single sign-on (CDSSO)

The following sections describe how to set up CDSSO for requests in a different domain:

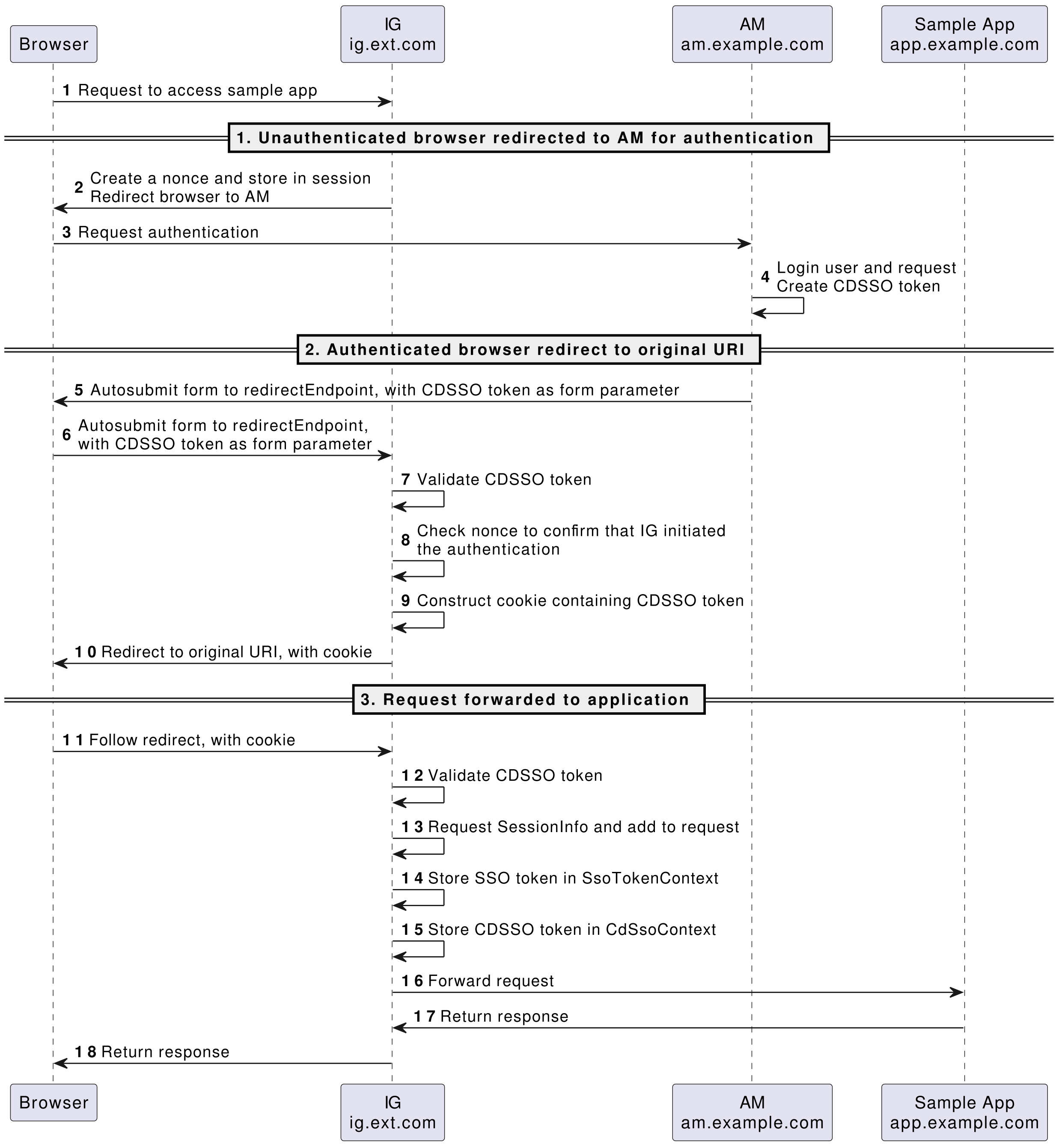
IMPORTANT

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. For an example, refer to Require users to authenticate to a specific realm.

The SSO mechanism described in Authenticating with SSO can be used when IG and AM are running in the same domain. When IG and AM are running in different domains, AM cookies are not visible to IG because of the same-origin policy.

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

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


Browser
IG ig.ext.com
AM 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. IG 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 IG callback URL for the redirect endpoint.

7. When `verificationSecretId` in CrossDomainSingleSignOnFilter is configured, IG verifies the AM session token signature.

WARNING

WARNING

For the security of your deployment, always configure `verificationSecretId` in CrossDomainSingleSignOnFilter.
When `verificationSecretId` is not configured, IG does not verify the signature of AM session tokens, increasing the risk of CDSSO token tampering.

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

9. IG 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 IG agent.

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

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

13-15. IG 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. IG forwards the request to the sample application, and the sample application returns the requested resource to the browser.

Before you start, prepare AM, IG, and the sample application, as described in Download and start IG.

1. Set up AM:
    a. Select **Applications** > **Agents** > **Identity Gateway** and register an IG agent with the following values:
        - **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.

        For AM 6.5.x and earlier versions, register an agent as described in Register an IG agent in AM 6.5 and earlier.
    b. (Optional) Authenticate the agent to AM as described in Authenticate an IG agent to AM.

IMPORTANT

IMPORTANT

IG 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. (From AM 6.5.3) 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 IG:

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

   b. Add the following `session` configuration to `admin.json`, to ensure that the browser passes the session cookie in the form-POST to the redirect endpoint (step 6 of Information flow during CDSSO):

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

   This step is required for the following reasons:

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

     For more information, refer to admin.json.

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

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

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

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

1. Linux
2. Windows

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

```
%appdata%\OpenIG\config\routes\static-resources.json
```

```
{
  "name" : "sampleapp-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css')}",
  "handler": "ReverseProxyHandler"
}
```

e. Add the following route to IG:

1. Linux
2. Windows

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

```
%appdata%\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",
            "verificationSecretId": "verify",
            "secretsProvider": {
              "type": "JwkSetSecretStore",
              "config": {
                "jwkUrl":
"http://am.example.com:8088/openam/oauth2/connect/jwk_u
ri"
              }
            }
          }
        }
      ],
      "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.
- The JwkSetSecretStore provides the URL to a JWK set on AM that contains verification keys identified by a `kid`.
- If the value of a `kid` in the JWK set matches a `kid` in the the signed AM session token, the JwkSetSecretStore verifies the signature of the AM session token.
- 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`.

> WARNING
>
> For the security of your deployment, always configure `verificationSecretId` in CrossDomainSingleSignOnFilter. When `verificationSecretId` is not configured, IG does not verify the signature of AM session tokens, increasing the risk of CDSSO token tampering.

3. Test the setup:

   a. If you are logged in to AM, log out and clear any cookies.

   b. Go to https://ig.ext.com:8443/home/cdsso.

   If you see warnings that the site is not secure, respond to the warnings to access the site.

   The CrossDomainSingleSignOnFilter redirects the request to AM for authentication.

   c. 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 from AM

Use IG with AM's password capture and replay to bring SSO to legacy web applications, without the need to edit, upgrade, or recode. This feature helps you to integrate legacy web applications with other applications using the same user identity.

For an alternative configuration using an AM policy agent instead of IG's CapturedUserPasswordFilter, refer to the documentation for earlier versions of IG.

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.

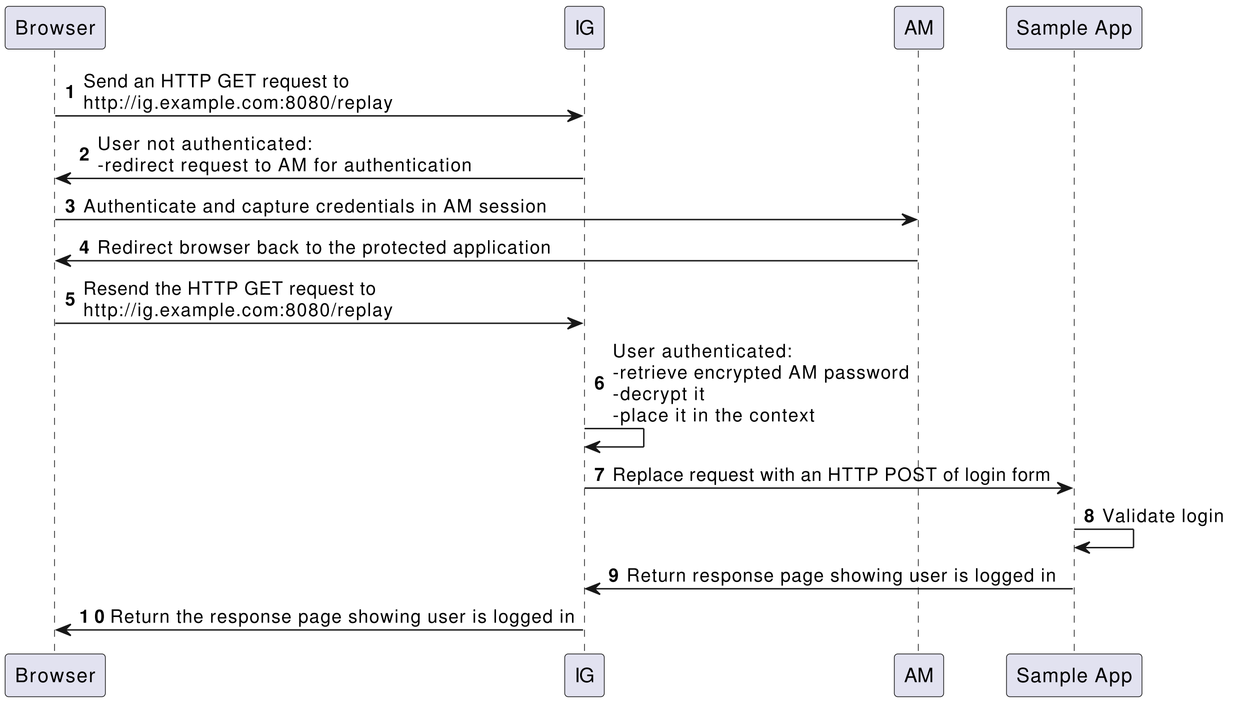

Browser
IG
AM
Sample App
1 Send an HTTP GET request to http://ig.example.com:8080/replay
2 User not authenticated: -redirect request to AM for authentication
3 Authenticate and capture credentials in AM session
4 Redirect browser back to the protected application
5 Resend the HTTP GET request to http://ig.example.com:8080/replay
6 User authenticated: -retrieve encrypted AM password -decrypt it -place it in the context
7 Replace request with an HTTP POST of login form
8 Validate login
9 Return response page showing user is logged in
10 Return the response page showing user is logged in

Figure 1. Data flow to log in to a protect appl

- IG intercepts the browser's HTTP GET request.
- Because the user is not authenticated, the SingleSignOnFilter redirects the user to AM for authentication.
- AM authenticates the user, capturing the login credentials, and storing the encrypted password in the user's AM session.
- AM redirects the browser back to the protected application.
- IG intercepts the browser's HTTP GET request again:
  - The user is now authenticated, so IG's SingleSignOnFilter passes the request to the CapturedUserPasswordFilter.
  - The CapturedUserPasswordFilter checks that the SessionInfoContext `${contexts.amSession.properties.sunIdentityUserPassword}` is available and not `null`. It then decrypts the password and stores it in the CapturedUserPasswordContext, at `${contexts.capturedPassword}`.
- The PasswordReplayFilter uses the username and decrypted password in the context to replace the request with an HTTP POST of the login form.
- The sample application validates the credentials.

- The sample application responds with the user's profile page.
- IG then passes the response from the sample application to the browser.

Before you start, prepare AM, IG, and the sample application as described in Example installation for this guide.

1. Generate an AES 256-bit key:

```
$ openssl rand -base64 32
loH...UFQ=
```

2. Set up AM:

   a. (From AM 6.5.3) 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/*?*`

   b. Select **Applications** > **Agents** > **Identity Gateway** and register an IG agent with the following values:

      - **Agent ID**: `ig_agent`
      - **Password**: `password`

      For AM 6.5.x and earlier versions, register an agent as described in Register an IG agent in AM 6.5 and earlier.

      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 an IG agent to AM.

      IMPORTANT

      IG 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. Update the Authentication Post Processing Classes for password replay:

      i. Select **Authentication** > **Settings** > **Post Authentication Processing**.

ii. In **Authentication Post Processing Classes**, add `com.sun.identity.authentication.spi.JwtReplayPassword`.

e. Add the AES 256-bit key to AM:

i. Select **DEPLOYMENT** > **Servers**, and then select the AM server name, `http://am.example.com:8088/openam`.

In earlier version of AM, select **Configuration** > **Servers and Sites**.

ii. Select **Advanced**, and add the following property:

- **PROPERTY NAME** : `com.sun.am.replaypasswd.key`
- **PROPERTY VALUE** : The value of the AES 256-bit key from step 1.

f. 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.

3. Set up IG:

a. Set environment variables for the value of the AES 256-bit key in step 1, and the IG agent password, and then restart IG:

```
$ export AES_KEY='AES 256-bit key'
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

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

1. Linux
2. Windows

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

```
%appdata%\OpenIG\config\routes\static-resources.json
```

```
{
  "name" : "sampleapp-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css')}",
  "handler": "ReverseProxyHandler"
}
```

c. Add the following route to IG:

1. Linux

2. Windows

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

```
%appdata%\OpenIG\config\routes\04-replay.json
```

```
{
  "name": "04-replay",
  "condition": "${find(request.uri.path, '^/replay')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore",
      "config": {
        "mappings": [
          {
            "secretId": "aes.key",
            "format": {
              "type": "SecretKeyPropertyFormat",
              "config": {
                "format": "BASE64",
                "algorithm": "AES"
              }
            }
          }
        ]
      }
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    },
    {
      "name": "CapturedUserPasswordFilter",
      "type": "CapturedUserPasswordFilter",
      "config": {
        "ssoToken": "${contexts.ssoToken.value}",
        "keySecretId": "aes.key",
        "keyType": "AES",
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "amService": "AmService-1"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        },
        {
          "type": "PasswordReplayFilter",
          "config": {
            "loginPage": "${true}",
            "credentials":
"CapturedUserPasswordFilter",
            "request": {
              "method": "POST",
              "uri":
"http://app.example.com:8081/login",
              "form": {
                "username": [
                  "${contexts.ssoToken.info.uid}"
                ],
                "password": [
                  "${contexts.capturedPassword.value}"
                ]
              }
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/replay`.
- The agent password for AmService is provided by a SystemAndEnvSecretStore in the heap.
- If the request does not have a valid AM session cookie, the SingleSignOnFilter redirects the request to AM for authentication.

  After authentication, the SingleSignOnFilter passes the request to the next filter, storing the cookie value in an `SsoTokenContext`.
- The PasswordReplayFilter uses the CapturedUserPasswordFilter declared in the heap to retrieve the AM password from AM session properties. The CapturedUserPasswordFilter uses the AES 256-bit key to decrypt the password, and then makes it available in a CapturedUserPasswordContext.

  The value of the AES 256-bit key is provided by the SystemAndEnvSecretStore.

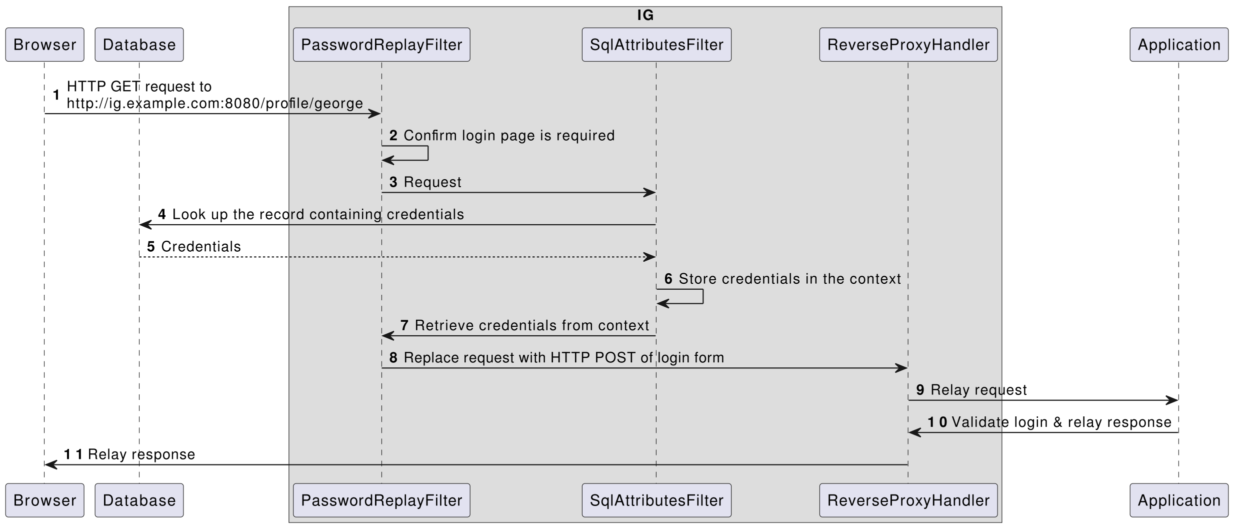
  The PasswordReplayFilter retrieves the username and password from the context. It replaces the browser's original HTTP GET request with an HTTP POST login request containing the credentials to authenticate to the sample application.

4. Test the setup:
   a. If you are logged in to AM, log out and clear any cookies.
   b. Go to http://ig.example.com:8080/replay. The SingleSignOnFilter redirects the request to AM for authentication.
   c. Log in to AM as user `demo`, password `Ch4ng31t`. The request is redirected to the sample application.

Password replay from a database

This section describes how to configure IG to get credentials from a database. This example is tested with H2 1.4.197.

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


IG
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

- IG 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 IG and the sample application as described in the Getting started.

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

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

```
C:\Temp\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
```

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

```
$ 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 IG configuration:

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

2. Set up IG:

a. Set an environment variable for the database password, and then restart IG:

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

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

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

1. Linux
2. Windows

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

```
%appdata%\OpenIG\config\routes\static-resources.json
```

```
{
  "name" : "sampleapp-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css')}",
  "handler": "ReverseProxyHandler"
}
```

c. Add the following route to IG:

1. Linux
2. Windows

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

```
%appdata%\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/~/test",
        "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"
                ],
                "target": "${attributes.sql}"
              }
            },
            "request": {
              "method": "POST",
              "uri":
"http://app.example.com:8081/login",
              "form": {
                "username": [
                  "${attributes.sql.USERNAME}"
                ],
                "password": [
                  "${attributes.sql.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, a parameter to pass into the statement, and where to store the search results in the request context attributes map.

  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. Go to http://ig.example.com:8080/profile.

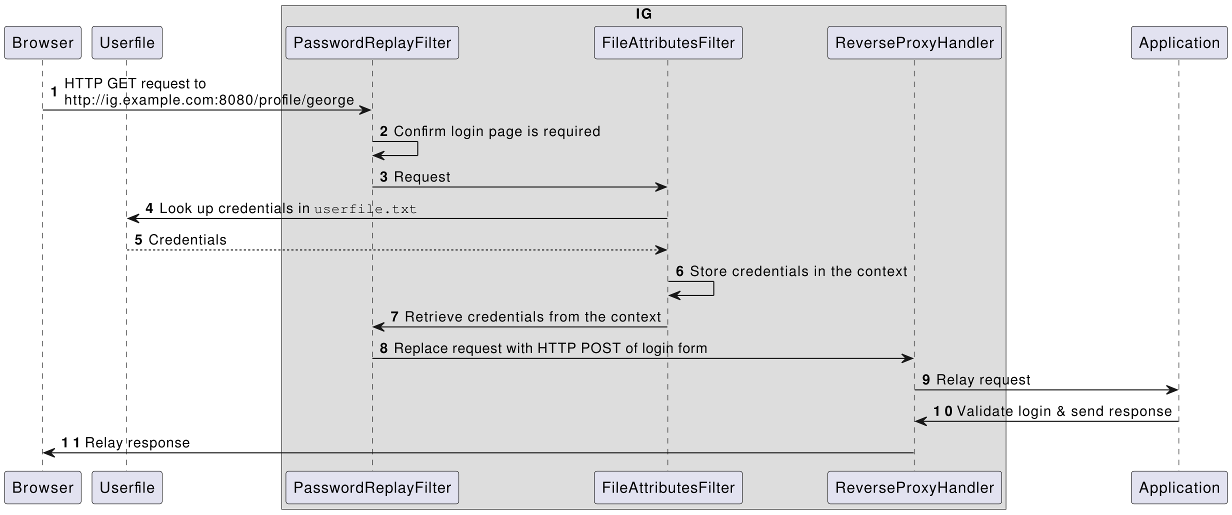
      Because the property `loginPage` resolves to `false`, the PasswordReplayFilter passes the request directly to the ReverseProxyHandler. The sample app returns the login page.

   b. Go to http://ig.example.com:8080/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 IG uses credentials in a file to log a user in to the sample application:


IG
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

- IG 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 IG and the sample application as described in the Getting started.

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

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

```
C:\Temp\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
```

2. Set up IG:

   a. Add the following route to IG, to serve .css and other static resources for the sample application:

      1. Linux

      2. Windows

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

```
%appdata%\OpenIG\config\routes\static-resources.json
```

```
{
  "name" : "sampleapp-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css')}",
  "handler": "ReverseProxyHandler"
}
```

   b. Add the following route to IG:

      1. Linux

      2. Windows

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

```
%appdata%\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",
                "target": "${attributes.credentials}"
              }
            },
            "request": {
              "method": "POST",
              "uri":
"http://app.example.com:8081/login",
              "form": {
                "username": [
                  "${attributes.credentials.username}"
                ],
                "password": [
                  "${attributes.credentials.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. Go to http://ig.example.com:8080/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.

   b. Go to http://ig.example.com:8080/profile/bob, or to any other URI starting with `http://ig.example.com:8080/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 IG, IG 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 and Cross-domain single sign-on 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 Authenticating with 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 IG:

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

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

4. Go to http://ig.example.com:8080/home/sso, and log in to AM as user `demo`, password `Ch4ng31t`.

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

6. Note that the IG system logs are updated with Websocket notifications about the logout. The following example uses AM 7.3:

```
... | TRACE | vert.x-eventloop-thread-2 |
o.f.o.t.n.w.SubscriptionService | @system | Received a
message: { "topic": "/agent/session.v2", "timestamp":
"...", "body": { "sessionuid": "58c...573", "eventType":
"LOGOUT" } }
... | TRACE | vert.x-eventloop-thread-2 |
o.f.o.t.n.w.SubscriptionService | @system | Received a
notification: { "topic": "/agent/session.v2", "timestamp":
"...", "body": { "sessionuid": "58c...573", "eventType":
"LOGOUT" } }
```

Policy enforcement

About policy enforcement

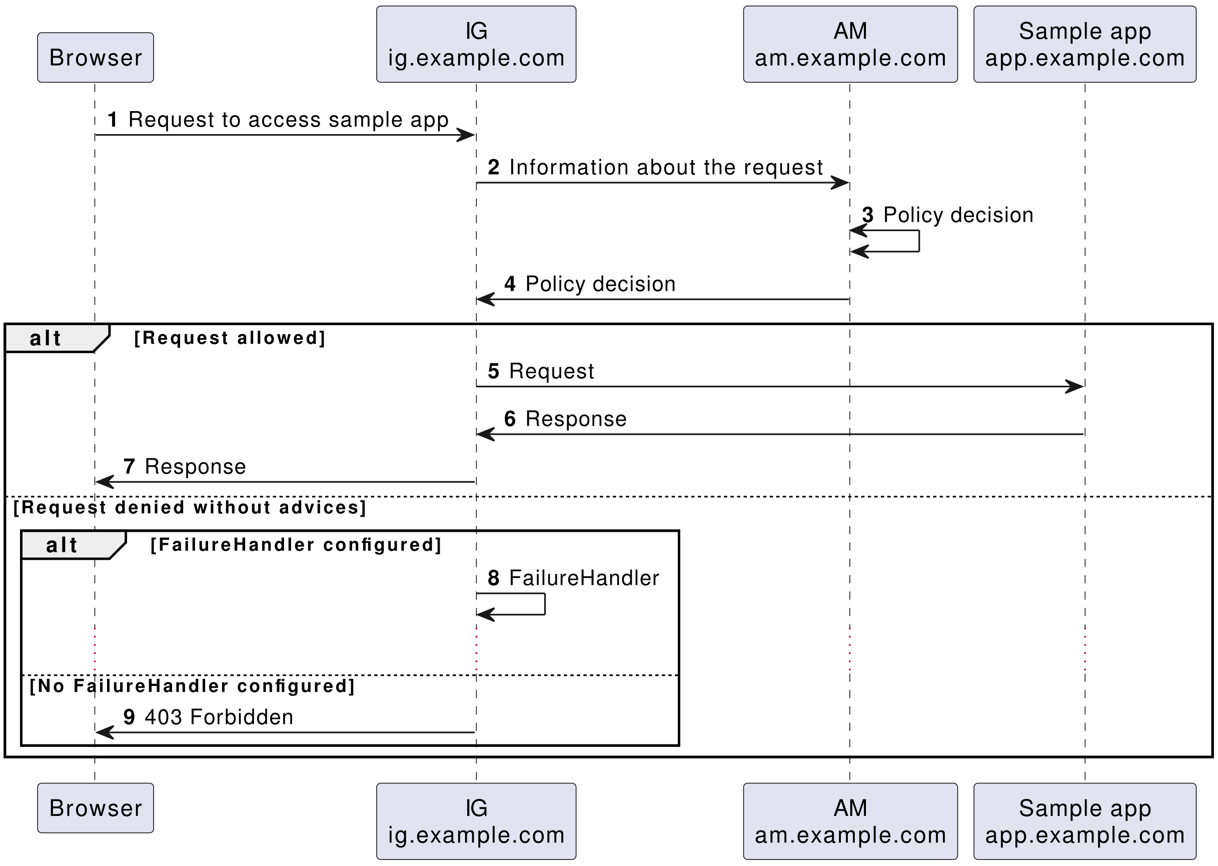
IG as a policy enforcement point (PEP) uses the PolicyEnforcementFilter to intercept requests for a resource and provide information about the request to AM.

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

For more information, refer to the PolicyEnforcementFilter and AM's Authentication and SSO guide.

Deny requests without advices

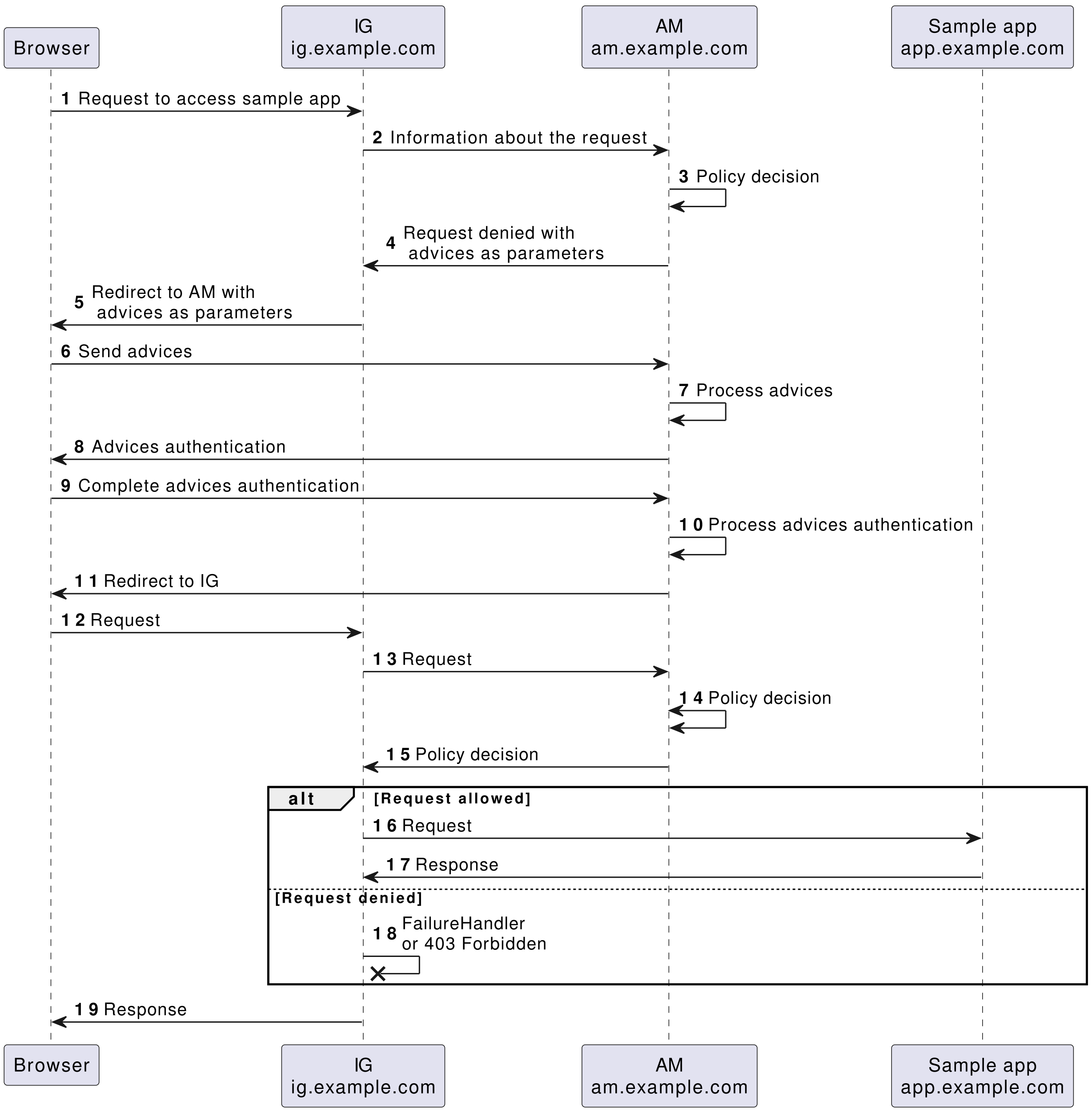
The following image shows a simplified flow of information when AM denies a request without advices.


Browser
IG ig.example.com
AM 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 advices as parameters in a redirect response

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

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


Browser
IG ig.example.com
AM 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 IG
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

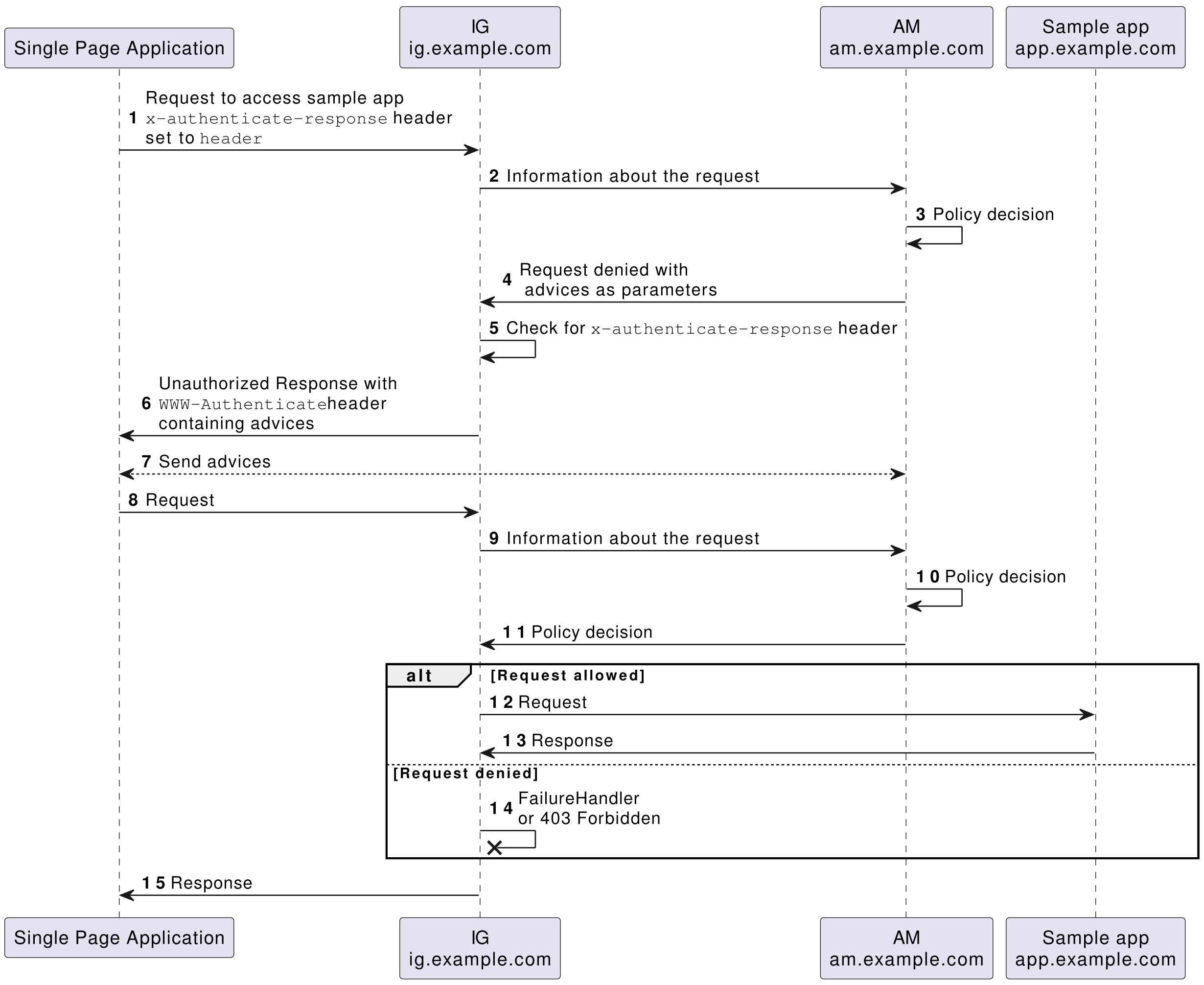
Deny requests with advices in a header

The following image shows a simplified flow of information when the request to IG includes an `x-authenticate-response` header with the value `header`. If the header has any other value, the flow in Deny requests with advices 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 advices, and IG sends the response with the advices in the `WWW-authenticate` header.

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



Single Page Application
IG ig.example.com
AM 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
1 0 Policy decision
1 1 Policy decision
alt
[Request allowed]
1 2 Request
1 3 Response
[Request denied]
1 4 FailureHandler or 403 Forbidden
1 5 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
```

IG returns a `WWW-Authenticate` header containing advices, as follows:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: SSOADVICE
realm="/",advices="eyJ...XX0=",am_uri="http://openam.example.com:
8080/am/"
transfer-encoding: chunked
connection: close
```

The advice decodes to a transaction condition advice:

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

Enforce policy decisions from AM

The following sections describe how to set up single sign on for requests in the same domain and in a different domain.

Enforce AM policy 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 IG and AM are in the same domain.

Before you start, set up and test the example in Authenticate with SSO through the default authentication service.

1. Set up AM:
   a. In the AM admin UI, 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. Set up IG:
   a. Set an environment variable for the IG agent password, and then restart IG:

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

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

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

1. Linux
2. Windows

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

```
%appdata%\OpenIG\config\routes\static-resources.json
```

```
{
  "name" : "sampleapp-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css')}",
  "handler": "ReverseProxyHandler"
}
```

c. Add the following route to IG:

1. Linux
2. Windows

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

```
%appdata%\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 IG 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. If you are logged in to AM, log out and clear any cookies.

   b. Go to http://ig.example.com:8080/home/pep-sso.

Because you have not 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 IG requests a policy decision using the AM session cookie.

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

Require users to authenticate to a specific 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. (From AM 6.5.3) 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/*?*`

   c. Select **Applications** > **Agents** > **Identity Gateway** and register an IG agent with the following values:

   - **Agent ID**: `ig_agent`
   - **Password**: `password`

     For AM 6.5.x and earlier versions, register an agent as described in Register an IG agent in AM 6.5 and earlier.

     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 an IG agent to AM.

   IMPORTANT

IMPORTANT

IG 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 IG:

a. Add the following route to IG:

1. Linux
2. Windows

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

```
%appdata%\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 a private browser, go to http://ig.example.com:8080/home/pep-sso-realm, and 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`.

Enforce AM policy 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 IG and AM are in different domains.

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

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. Set up IG:

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

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

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

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

   1. Linux
   2. Windows

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

   ```
   %appdata%\OpenIG\config\routes\static-resources.json
   ```

   ```
   {
     "name" : "sampleapp-resources",
     "baseURI" : "http://app.example.com:8081",
     "condition": "${find(request.uri.path,'^/css')}",
     "handler": "ReverseProxyHandler"
   }
   ```

   c. Add the following route to IG:

   1. Linux
   2. Windows

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

   ```
   %appdata%\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",
            "verificationSecretId": "verify",
            "secretsProvider": {
              "type": "JwkSetSecretStore",
              "config": {
                "jwkUrl":
"http://am.example.com:8088/openam/oauth2/connect/jwk_u
ri"
```

```
              }
            }
          }
        },
        {
          "name": "PolicyEnforcementFilter-1",
          "type": "PolicyEnforcementFilter",
          "config": {
            "application": "PEP-CDSSO",
            "ssoTokenSubject":
"${contexts.cdsso.token}",
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

WARNING

For the security of your deployment, always configure `verificationSecretId` in CrossDomainSingleSignOnFilter.
When `verificationSecretId` is not configured, IG does not verify the signature of AM session tokens, increasing the risk of CDSSO token tampering.

3. Test the setup:

   a. If you are logged in to AM, log out and clear any cookies.

   b. Go to https://ig.ext.com:8443/home/pep-cdsso.

      If you have warnings that the site is not secure respond to the warnings to access the site.

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

Enforce policy decisions using claimsSubject

This example is a variant of Enforce AM policy decisions in the same domain. It enforces a policy decision from AM, using `claimsSubject` instead of `ssoTokenSubject` to identify the subject.

1. Set up AM as described in Enforce AM policy decisions in the same domain.
2. In AM, select the policy you created in the previous step, and add a new resource:
   - **Resource Type**: `URL`
   - **Resource pattern**: `*://*:*/*`
   - **Resource value**: `http://app.example.com:8081/home/pep-claims`
3. In the same policy, add the following subject condition:
   - `Any of`
   - **Type** : `OpenID Connect/JwtClaim`
   - **claimName** : `iss`
   - **claimValue** : `am.example.com`
4. Set an environment variable for the IG agent password, and then restart IG:

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

   The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.
5. Add the following route to serve static resources, such as .css, for the sample application:
   1. Linux
   2. Windows

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

   ```
   %appdata%\OpenIG\config\routes\static-resources.json
   ```

   ```
   {
     "name" : "sampleapp-resources",
     "baseURI" : "http://app.example.com:8081",
     "condition": "${find(request.uri.path,'^/css')}",
     "handler": "ReverseProxyHandler"
   }
   ```
6. Add the following route to IG:

1. Linux
2. Windows

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

```
%appdata%\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"
    }
  }
}
```

7. Go to http://ig.example.com:8080/home/pep-claims.
8. Log in to AM as user `demo`, password `Ch4ng31t`.

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

Using WebSocket notifications to evict the policy cache

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

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

1. Set up and test the example in Enforce AM policy decisions in the same domain.
2. 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
}
```

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

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

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

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

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

5. Go to http://ig.example.com:8080/home/pep-sso, and log in to AM as user `demo`, password `Ch4ng31t`.

6. In a seperate 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`.

7. Note that the IG system logs are updated with Websocket notifications about the change:

   ```
   ... | TRACE | vert.x-eventloop-thread-2 |
   o.f.o.t.n.w.SubscriptionService | @system | Received a
   message: { "topic": "/agent/policy", "timestamp": ...,
   "body": { "realm": "/", "policy": "PEP-SSO", "policySet":
   "PEP-SSO", "eventType": "UPDATE" } }
   ... | TRACE | vert.x-eventloop-thread-2 |
   o.f.o.t.n.w.SubscriptionService | @system | Received a
   notification: { "topic": "/agent/policy", "timestamp":
   ..., "body": { "realm": "/", "policy": "PEP-SSO",
   "policySet": "PEP-SSO", "eventType": "UPDATE" } }
   ```

Harden authorization with advice from AM

To protect sensitive resources, AM policies can be configured with additional conditions to harden the authorization. When AM communicates these policy decisions to IG, the decision includes advices to indicate what extra conditions the user must meet.

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

The following sections build on the policies in Enforce policy decisions from AM to step up the authentication level:

Step up the authentication level for an AM session

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 section uses the policies you created in Enforce AM policy decisions in the same domain and Enforce AM policy decisions in different domains, adding an authorization policy with a Authentication by Service environment condition. 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 a verification code to meet the `AuthenticateToService` environment condition.

> Before you start, set up one of the following examples in Enforce AM policy decisions in the same domain or Enforce AM policy decisions in different domains.
>
> 1. In the AM admin UI, add an environment condition to the policy:
>   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** : `VerificationCodeLevel1`
> 2. Set up client-side and server-side scripts:
>   a. Select **Scripts** > **Scripted Module - Client Side**, and replace the default script with the following script:
>
> ```
> autoSubmitDelay = 60000;
>
> function callback() {
>     var parent = document.createElement("div");
>     parent.className = "form-group";
>
>     var label = document.createElement("label");
>     label.className = "sr-only separator";
>     label.setAttribute("for", "answer");
>     label.innerText = "Verification Code";
>     parent.appendChild(label);
> ```

```
        var input = document.createElement("input");
        input.className = "form-control input-lg";
        input.type = "text";
        input.placeholder = "Enter your verification code";
        input.name = "answer";
        input.id = "answer";
        input.value = "";
        input.oninput = function(event) {
            var element =
    document.getElementById("clientScriptOutputData");
            if (!element.value || element.value ==
    "clientScriptOutputData") element.value = "{}";
            var json = JSON.parse(element.value);
            json["answer"] = event.target.value;
            element.value = JSON.stringify(json);
        };
        parent.appendChild(input);

        var fieldset =
    document.forms[0].getElementsByTagName("fieldset")[0];
        fieldset.prepend(parent);
    }

    if (document.readyState !== 'loading') {
        callback();
    } else {
        document.addEventListener("DOMContentLoaded",
    callback);
    }
```

Leave all other values as default.

This client-side script adds a field to the AM form, in which the user is required to enter a verification code. The script formats the entered code as a JSON object, as required by the server-side script.

b. Select **</> Scripts** > **Scripted Module - Server Side**, and replace the default script with the following script:

```
username = 'demo'
logger.error('username: ' + username)

// Test whether the user 'demo' enters the correct
validation code
data = JSON.parse(clientScriptOutputData);
answer = data.answer;
```

```
if (answer !== '123456') {
  logger.error('Authentication Failed !!')
  authState = FAILED;
} else {
  logger.error('Authenticated !!')
  authState = SUCCESS;
}
```

Leave all other values as default.

This server-side script tests that the user `demo` has entered `123456` as the verification code.

3. Add an authentication module:
   a. Select **Authentication** > **Modules**, and add a module with the following settings:
      - **Name** : `VerificationCodeLevel1`
      - **Type** : `Scripted Module`
   b. In the authentication module, enable the option for client-side script, and select the following options:
      - **Client-side Script** : `Scripted Module - Client Side`
      - **Server-side Script** : `Scripted Module - Server Side`
      - **Authentication Level** : `1`
   c. Add the authentication module to an authentication chain:
      i. Select **Authentication** > **Chains**, and add a chain called `VerificationCodeLevel1` .
      ii. Add a module with the following settings:
         - **Select Module** : `VerificationCodeLevel1`
         - **Select Criteria** : `Required`
4. Test the setup:
   a. Log out of AM.
   b. Access the route:
      - For SSO, go to https://ig.example.com:8080/home/pep-sso.
      - For CDSSO, go to https://ig.ext.com:8443/home/pep-cdsso.

      If you have not previously authenticated to AM, the SingleSignOnFilter redirects the request to AM for authentication.
   c. Log in to AM as user `demo` , password `Ch4ng31t` .

AM creates a session with the default authentication level `0`, and IG 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`.

d. In the session upgrade window, enter the verification code `123456`.

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.

Increase authorization for a single transaction

Transactional authorization improves security by requiring a user to perform additional actions when trying to access a resource protected by an AM policy. For example, they must reauthenticate to an authentication module or respond to a push notification on their mobile device.

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 for an AM session, adding a simple authorization policy with a `Transaction` environment condition. Each time the user agent tries to access the protected resource, the user must reauthenticate to an authentication module by providing a verification code.

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

1. In the AM admin UI, add a new environment condition:
    a. Select the policy set:
        - For SSO, select **Authorization** > **Policy Sets** > **PEP-SSO**.
        - For CDSSO, select **Authorization** > **Policy Sets** > **PEP-CDSSO**.
    b. In the IG policy, select Environments and add another environment condition:
        - `All of`
        - **Type** : `Transaction`
        - **Authentication strategy** : `Authenticate To Module`
        - **Strategy specifier** : `TxVerificationCodeLevel5`
2. Set up client-side and server-side scripts:

a. Select **</> Scripts** > **New Script**, and add the following client-side script:

- **Name** : `Tx Scripted Module - Client Side`
- **Script Type** : `Client-side Authentication`

```
autoSubmitDelay = 60000;

function callback() {
    var parent = document.createElement("div");
    parent.className = "form-group";

    var label = document.createElement("label");
    label.className = "sr-only separator";
    label.setAttribute("for", "answer");
    label.innerText = "Verification Code";
    parent.appendChild(label);

    var input = document.createElement("input");
    input.className = "form-control input-lg";
    input.type = "text";
    input.placeholder = "Enter your TX code";
    input.name = "answer";
    input.id = "answer";
    input.value = "";
    input.oninput = function(event) {
        var element =
document.getElementById("clientScriptOutputData");
        if (!element.value || element.value ==
"clientScriptOutputData") element.value = "{}";
        var json = JSON.parse(element.value);
        json["answer"] = event.target.value;
        element.value = JSON.stringify(json);
    };
    parent.appendChild(input);

    var fieldset =
document.forms[0].getElementsByTagName("fieldset")
[0];
    fieldset.prepend(parent);
}

if (document.readyState !== 'loading') {
    callback();
} else {
    document.addEventListener("DOMContentLoaded",
callback);
}
```

This client-side script adds a field to the AM form, in which the user is required to enter a TX code. The script formats the entered code as a JSON object, as required by the server-side script.

b. Select </> **Scripts** > **New Script**, and add the following server-side script:

- **Name** : `Tx Scripted Module - Server Side`
- **Script Type** : `Server-side Authentication`

```
username = 'demo'
logger.error('username: ' + username)

// Test whether the user 'demo' enters the correct
validation code
data = JSON.parse(clientScriptOutputData);
answer = data.answer;

if (answer !== '789') {
  logger.error('Authentication Failed !!')
  authState = FAILED;
} else {
  logger.error('Authenticated !!')
  authState = SUCCESS;
}
```

This server-side script tests that the user `demo` has entered `789` as the verification code.

3. Add an authentication module:

a. Select **Authentication** > **Modules**, and add a module with the following settings:

- **Name** : `TxVerificationCodeLevel5`
- **Type** : `Scripted Module`

b. In the authentication module, enable the option for client-side script, and select the following options:

- **Client-side Script** : `Tx Scripted Module - Client Side`
- **Server-side Script** : `Tx Scripted Module - Server Side`
- **Authentication Level** : `5`

4. Test the setup:

a. Log out of AM.

b. Access your route:
   - For SSO, go to http://ig.example.com:8080/home/pep-sso.
   - For CDSSO, go to https://ig.ext.com:8443/home/pep-cdsso.

     If you have not previously authenticated to AM, the SingleSignOnFilter redirects the request to AM for authentication.

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

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

d. Enter the verification code `123456` to upgrade the authorization level for the session to `1`.

   The authentication module you configured for transactional authorization requires authentication level `5`, so AM issues a `TransactionConditionAdvice`.

e. In the transaction upgrade window, enter the verification code `789`.

   AM upgrades the authentication level for this policy evaluation to `5`, and then returns a policy decision that grants a one-time access to the sample application. If you try to access the sample application again, you must enter the code again.

OAuth 2.0

OAuth 2.0 includes the following entities:

- *Resource owner* : A user who owns protected resources on a resource server. For example, a resource owner can store photos in a web service.
- *Resource server* : A service that gives authorized client applications access to the resource owner's protected resources. In OAuth 2.0, an authorization server grants authorization to a client application, based on the resource owner's consent. For example, a resource server can be a web service that holds a user's photos.
- *Client* : An application that requests access to the resource owner's protected resources, on behalf of the resource owner. For example, a client can be a photo printing service requesting access to a resource owner's photos stored on a web service, after the resource owner gives the client consent to download the photos.
- *Authorization server* : A service responsible for authenticating resource owners, and obtaining their consent to allow client applications to access their resources. For example, AM can act as the OAuth 2.0 authorization server to authenticate resource owners and obtain their consent. Other services, such as Google and Facebook can provide OAuth 2.0 authorization services.

IG as an OAuth 2.0 client

IG 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 does not act on behalf of the resource owner.

IG 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 IG as the resource server:

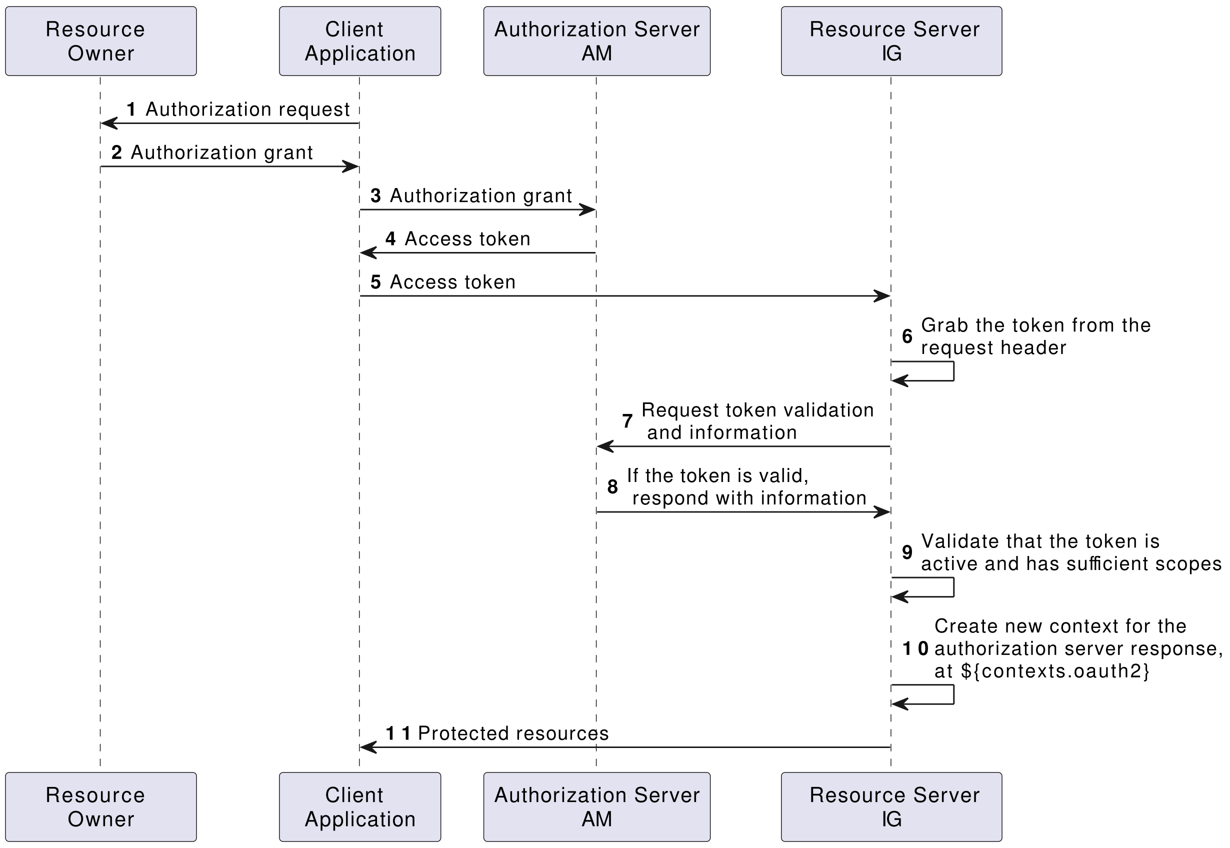

Resource Owner
Client Application
Authorization Server AM
Resource Server IG
1 Authorization request
2 Authorization grant
3 Authorization grant
4 Access token
5 Access token
6 Grab the token from the request header
7 Request token validation and information
8 If the token is valid, respond with information
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 AM
Resource Server IG

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

- The application obtains an *authorization grant*, representing the resource owner's consent. For information about the different OAuth 2.0 grant mechanisms supported by AM, refer to OAuth 2.0 grant flows in AM's *OAuth 2.0 guide*.
- 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. For information, refer to Auditing your deployment and Audit framework.

Validate stateful or stateless access tokens through the introspection endpoint

This section sets up IG as an OAuth 2.0 resource server, using the introspection endpoint.

For more information about configuring AM as an OAuth 2.0 authorization service, refer to AM's OAuth 2.0 guide.

IMPORTANT

This procedure uses the *Resource Owner Password Credentials* grant type. According to information in the The OAuth 2.0 Authorization Framework, minimize use of this grant type and utilize other grant types whenever possible.

Before you start, prepare AM, IG, and the sample application as described in Example installation for this guide.

1. Set up AM:
   a. Select **Applications** > **Agents** > **Identity Gateway**, and register an IG agent with the following values:
      - **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.

b. (Optional) Authenticate the agent to AM as described in Authenticate an IG agent to AM.

IMPORTANT

IG 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. (From AM 6.5) On the **Advanced** tab, select the following value:

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

2. Set up IG

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

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

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

b. Add the following route to IG:

1. Linux
2. Windows

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

```
%appdata%\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>"
        }
      }
    }
  }
}
```

For information about how to set up the IG route in Studio, see 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 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 IG does not 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&sc
   ope=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 http://ig.example.com:8080/rs-introspect --
   header "Authorization: Bearer ${mytoken}"
   ```

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

Define required scopes with a script

This example builds on the example in Validate access tokens through the introspection endpoint 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`.

1. Set up and test the example in Validate access tokens through the introspection endpoint.
2. Add the following route to IG:
   1. Linux
   2. Windows

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

```
%appdata%\OpenIG\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>"
        }
      }
    }
  }
}
```

3. 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&sc
ope=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 http://ig.example.com:8080/rs-dynamicscope --
header "Authorization: Bearer ${mytoken}"

{
  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 http://ig.example.com:8080/rs-
dynamicscope/employee --header "Authorization: Bearer
${mytoken}"
```

4. 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&sc
ope=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 http://ig.example.com:8080/rs-
dynamicscope/employee --header "Authorization: Bearer
${mytoken}"
```

Validate stateless access tokens with the StatelessAccessTokenResolver

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 sections provide examples of how to validate signed and encrypted access tokens:

Validate signed access tokens with the StatelessAccessTokenResolver and JwkSetSecretStore

This section provides examples of how to validate signed access tokens with the StatelessAccessTokenResolver, using a JwkSetSecretStore. For more information about JwkSetSecretStore, refer to JwkSetSecretStore.

IMPORTANT

This procedure uses the *Resource Owner Password Credentials* grant type. According to information in the The OAuth 2.0 Authorization Framework, minimize use of this grant type and utilize 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 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 Whitelist** : `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. (From AM 6.5) On the **Advanced** tab, select the following values:

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

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

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

2. Set up IG:

a. Add the following route to IG:

1. Linux
2. Windows

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

```
%appdata%\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_u
ri"
            }
          }
        ]
      }
    }
  ],
  "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. For more information, refer to OAuth2Context.
- If there is no access token in a request, or token validation does not complete successfully, the filter returns an HTTP error status to the user agent, and IG does not 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&sc
   ope=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 http://ig.example.com:8080/rs-stateless-
signed --header "Authorization: Bearer ${mytoken}"

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

Validate signed access tokens with the StatelessAccessTokenResolver and KeyStoreSecretStore

This section provides examples of how to validate signed access tokens with the StatelessAccessTokenResolver, using a KeyStoreSecretStore. For more information about KeyStoreSecretStore, refer to KeyStoreSecretStore.

Set Up Keys for Signing

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`
   - IG 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 IG:

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 IG 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 IG configuration, set an environment variable for the keystore password:

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

d. Restart IG.

Validate signed access tokens with the StatelessAccessTokenResolver and KeyStoreSecretStore

IMPORTANT

This procedure uses the *Resource Owner Password Credentials* grant type. According to information in the The OAuth 2.0 Authorization Framework, minimize use of this grant type and utilize 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 ID** : `keystore.pass`
         - **Entry password secret ID** : `keystore.pass`

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

         - **Secret ID** : `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 Whitelist** : `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. (From AM 6.5) On the **Advanced** tab, select the following values:

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

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

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

2. Set up IG:

a. Add the following route to IG, and replace the path to IG_keystore.p12:

1. Linux
2. Windows

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

```
%appdata%\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. For more information, refer to OAuth2Context.
- If there is no access token in a request, or if the token validation does not complete successfully, the filter returns an HTTP error status to the

user agent, and IG 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&sc
   ope=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 http://ig.example.com:8080/rs-stateless-
   signed-ksss --header "Authorization: Bearer ${mytoken}"

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

Validating encrypted access tokens with the StatelessAccessTokenResolver and KeyStoreSecretStore

Set up keys for encryption

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`
   - IG 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 IG:

a. Import `encryption-key` into the IG 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 IG 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 IG configuration, set an environment variable for the keystore password:

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

d. Restart IG.

Validate encrypted access tokens with the StatelessAccessTokenResolver and KeyStoreSecretStore

1. Set up AM:

   a. Set up AM as described in Validate signed access tokens with the StatelessAccessTokenResolver and KeyStoreSecretStore.

   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 ID** : `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-Based Tokens**.

2. Set up IG:

   a. Add the following route to IG, and replace ig_keystore_directory:

      1. Linux

      2. Windows

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

```
%appdata%\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`, used in: Validate signed access tokens with the StatelessAccessTokenResolver and KeyStoreSecretStore:

- 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.

Test the setup for an encrypted access token

1. 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")
```

2. Display the token:

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

   Note that the token is structured as an encrypted token.

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

```
$ curl -v http://ig.example.com:8080/rs-stateless-
encrypted --header "Authorization: Bearer ${mytoken}"

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

Validate certificate-bound access tokens

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.

When a client requests an access token from AM through mTLS, AM can use a *confirmation key* to bind the access token to the presented client certificate. The confirmation key is the certificate *thumbprint*, computed as `base64url-`

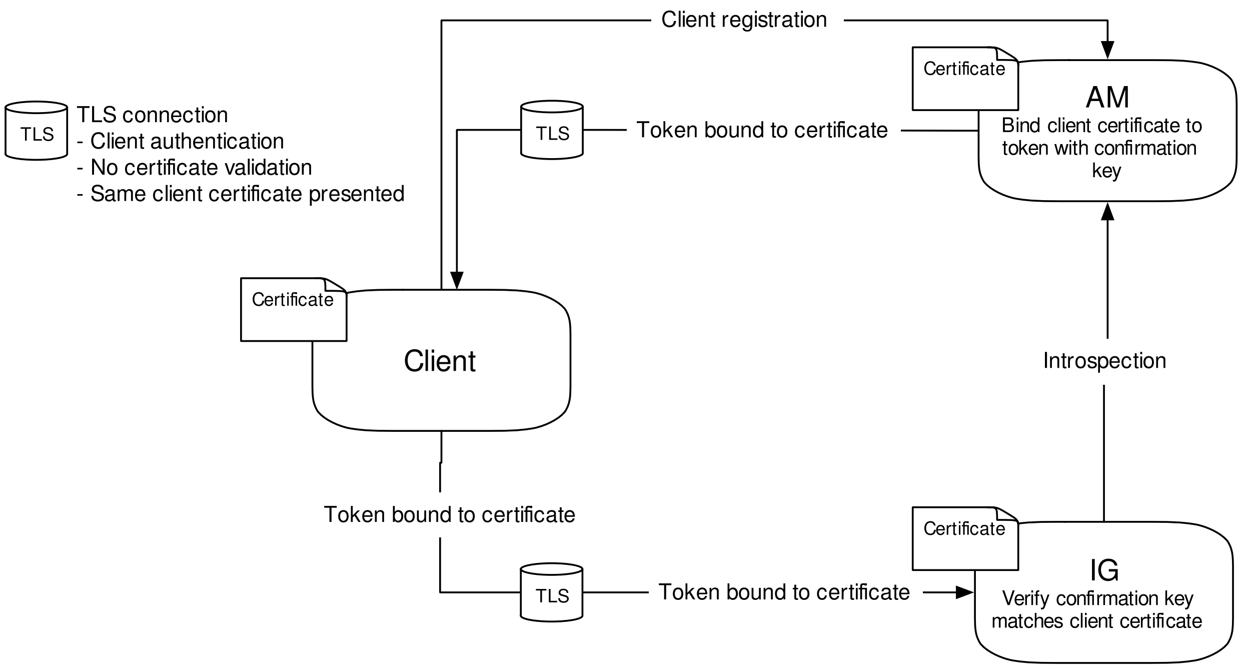
`encode(sha256(der(certificate)))` . The access token is then *certificate-bound*. For more information, refer to Mutual TLS in AM's *OAuth 2.0 guide*.

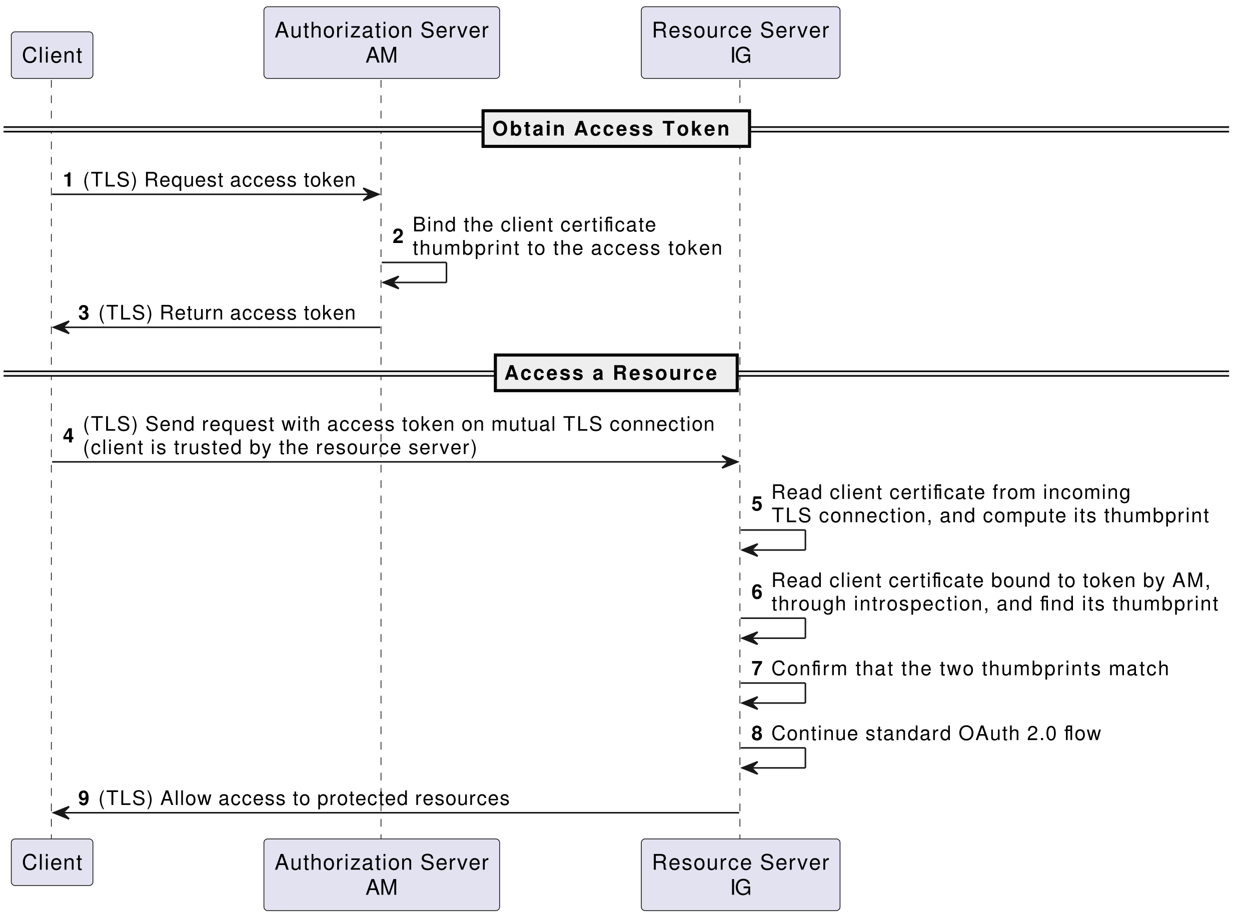
When the client connects to IG by using that certificate, IG can verify that the confirmation key corresponds to the presented certificate. This proof-of-possession interaction ensures that only the client in possession of the key corresponding to the certificate can use the access token to access protected resources.

mTLS using standard TLS client certificate authentication

IG 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 IG through a TLS connection, for which IG is the TLS termination point. If TLS is terminated at a reverse proxy or load balancer before IG, use the example in mTLS Using Trusted Headers.


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


Client
Authorization Server AM
Resource Server IG
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 AM
Resource Server IG

Perform the procedures in this section to set up and test mTLS using standard TLS client certificate authentication:

Set up keystores and truststores

1. Locate the following keystore directories, and in a terminal create variables for them:
   - `oauth2_client_keystore_directory`
   - `am_keystore_directory`
   - `ig_keystore_directory`
2. Create self-signed RSA key pairs for AM, IG, 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 openig-server \
-keyalg RSA \
-keysize 2048 \
-keystore $ig_keystore_directory/keystore.p12 \
-storepass changeit \
-storetype PKCS12 \
-keypass changeit \
-validity 360 \
-dname CN=ig.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
```

3. Export the certificates to .pem so that the **`curl`** client can verify the identity of the AM and IG 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
```

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

Certificate stored in file openig-server.cert.pem
```

4. 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
   ```

   You can now delete the client keystore.

5. 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
   ```

Set up AM for HTTPS (server-side) in Tomcat

This procedure sets up AM for HTTPS in Tomcat. For more information, see Secure connections to the AM container in AM's *Installation guide*.

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.p
   12"
                    truststorePassword="changeit"
                    truststoreType="PKCS12">
       <Certificate
   certificateKeystoreFile="am_keystore_directory/keystore.p1
   2"
                    certificateKeystorePassword="changeit"
                    certificateKeystoreType="PKCS12"/>
     </SSLHostConfig>
   </Connector>
   ```

   The `optionalNoCA` property allows the presentation of client certificates to be optional. Tomcat does not check them against the list of trusted CAs.

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`.

Set up IG for HTTPS (server-side)

This procedure sets up IG for HTTPS. Before you start, install IG as described in Download and start IG.

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

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

NOTE

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

2. Add the following configuration to IG, replacing instances of ig_keystore_directory and oauth2_client_keystore_directory with your path:

   1. Linux

   2. Windows

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

```
%appdata%\OpenIG\config\admin.json
```

```
{
  "mode": "DEVELOPMENT",
  "connectors": [
    {
      "port": 8080
    },
    {
      "port": 8443,
      "tls": {
        "type": "ServerTlsOptions",
        "config": {
          "alpn": {
            "enabled": true
          },
          "clientAuth": "REQUEST",
          "keyManager": {
            "type": "SecretsKeyManager",
            "config": {
              "signingSecretId": "key.manager.secret.id",
              "secretsProvider": {
                "type": "KeyStoreSecretStore",
                "config": {
                  "file": "
<ig_keystore_directory>/keystore.p12",
                  "storePasswordSecretId":
"keystore.pass",
                  "secretsProvider": "SecretsPasswords",
                  "mappings": [
                    {
                      "secretId": "key.manager.secret.id",
                      "aliases": [
                        "openig-server"
                      ]
                    }
                  ]
                }
              }
            }
          },
          "trustManager": {
            "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"
                      ]
                    }
                  ]
                }
              }
            }
          }
        }
      }
    }
  ],
  "heap": [
    {
      "name": "SecretsPasswords",
      "type": "FileSystemSecretStore",
      "config": {
        "directory": "<ig_keystore_directory>",
        "format": "PLAIN"
      }
    }
  ]
}
```

Notice the following features of the configuration:

- IG starts on port `8080`, and on `8443` over TLS.
- IG's private keys for TLS are managed by the SecretsKeyManager, which references the KeyStoreSecretStore that holds the keys.
- The password of the KeyStoreSecretStore is provided by the FileSystemSecretStore.
- The KeyStoreSecretStore maps the keystore alias to the secret ID for retrieving the private signing keys.

3. Start IG:

   1. Linux
   2. Windows

```
$ /path/to/identity-gateway/bin/start.sh

...
... started in 1234ms on ports : [8080 8443]
```

```
C:\path\to\identity-gateway\bin\start.bat
```

By default, IG configuration files are located under `$HOME/.openig` (on Windows `%appdata%\OpenIG`). For information about how to use a different location, refer to Change the base location of the IG configuration.

Set up AM as an authorization server with mTLS

1. In a the AM admin UI, select **Applications** > **Agents** > **Identity Gateway**, and register an IG 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.

2. (Optional) Authenticate the agent to AM as described in Authenticate an IG agent to AM.

   IMPORTANT

   IG 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. Configure an OAuth 2.0 Authorization Server:
   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
4. 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. 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**: `tls_client_auth`

   c. On the **signing and Encryption** tab, select the following values:
      - **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 any valid certificate trusted by the trust manager.
      - **Use Certificate-Bound Access Tokens**: Enabled

5. Set up AM secret stores to trust the client certificate:
   a. Select **Secret Stores**, and add a store with the following values:
      - **Secret Store ID**: `trusted-ca-certs`
      - **Store Type**: `Keystore`
      - **File**: `$oauth2_client_keystore_directory/cacerts.p12`
      - **Keystore type**: `PKCS12`
      - **Store password secret ID**: `passwordSecretId`
   b. Select **Mappings** and add the following mapping:
      - **Secret ID**: `am.services.oauth2.tls.client.cert.authentication`
      - **Aliases**: `client-cert`

   When the token endpoint authentication method is `tls_client_auth`, this secret is used to validate the client certificate. Add an alias in this list for each client that uses `tls_client_auth`. For certificates signed by a CA, add the CA certificate to the list.

Set up IG as a resource server with mTLS

1. Set an environment variable for the IG agent password, and then restart IG:

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

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

2. Add the following route to IG:
   1. Linux
   2. Windows

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

```
%appdata%\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 that 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.

Test the setup

1. Get an access token from AM, over TLS:

```
$ mytoken=$(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&grant_type=client_credentials&scope=test' \
https://am.example.com:8445/openam/oauth2/access_token |
jq -r .access_token)
```

2. Introspect the access token on AM:

```
$ curl --request POST \
-u ig_agent:password \
--header 'content-type: application/x-www-form-urlencoded'
\
--data token=${mytoken} \
http://am.example.com:8088/openam/oauth2/realms/root/intro
spect | jq

{
  "active": true,
  "scope": "test",
  "realm": "/",
  "client_id": "client-application",
  "user_id": "client-application",
  "token_type": "Bearer",
  "exp": 155...833,
  "sub": "(age!client-application)",
  "subname": "client-application",
  "iss": "http://am.example.com:8088/openam/oauth2",
  "cnf": {
    "x51...156": "T4u...R9Q"
  },
  "authGrantId": "dfE...2vk",
  "auditTrackingId": "e36...524"
}
```

The `cnf` property indicates the value of the confirmation code, as follows:

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

3. Access the IG route to validate the token's confirmation thumbprint with the ConfirmationKeyVerifierAccessTokenResolver:

```
$ curl --request POST \
--cacert $ig_keystore_directory/openig-server.cert.pem \
--cert $oauth2_client_keystore_directory/client.cert.pem \
--key $oauth2_client_keystore_directory/client.key.pem \
--header "authorization: Bearer ${mytoken}" \
https://ig.example.com:8443/mtls-certificate

mTLS
  Valid token: 2Bp...s_k
  Confirmation keys: {
  ...
  }
```

The validated token and confirmation keys are displayed.

mTLS using trusted headers

IG 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 IG. IG 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 IG.
- If the connection is not over TLS, the connection presents no client certificate.

If the client is connected directly to IG through a TLS connection, for which IG is the TLS termination point, use the example in mTLS Using Standard TLS Client Certificate Authentication.

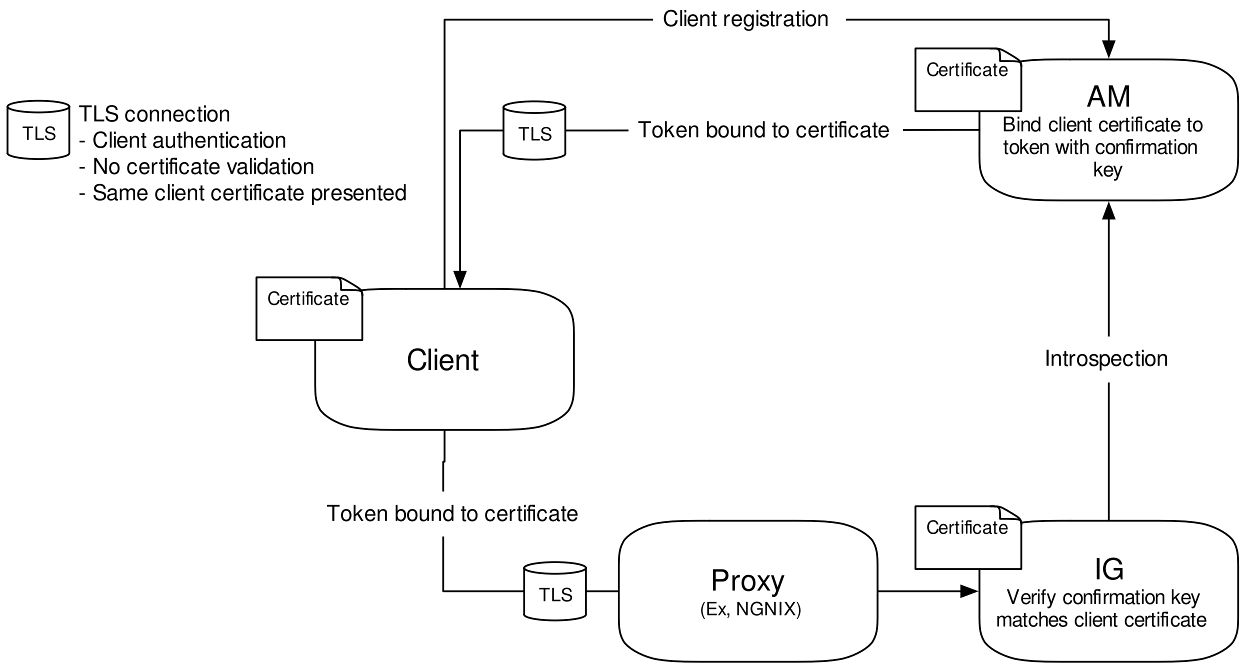
Configure the proxy or load balancer to:

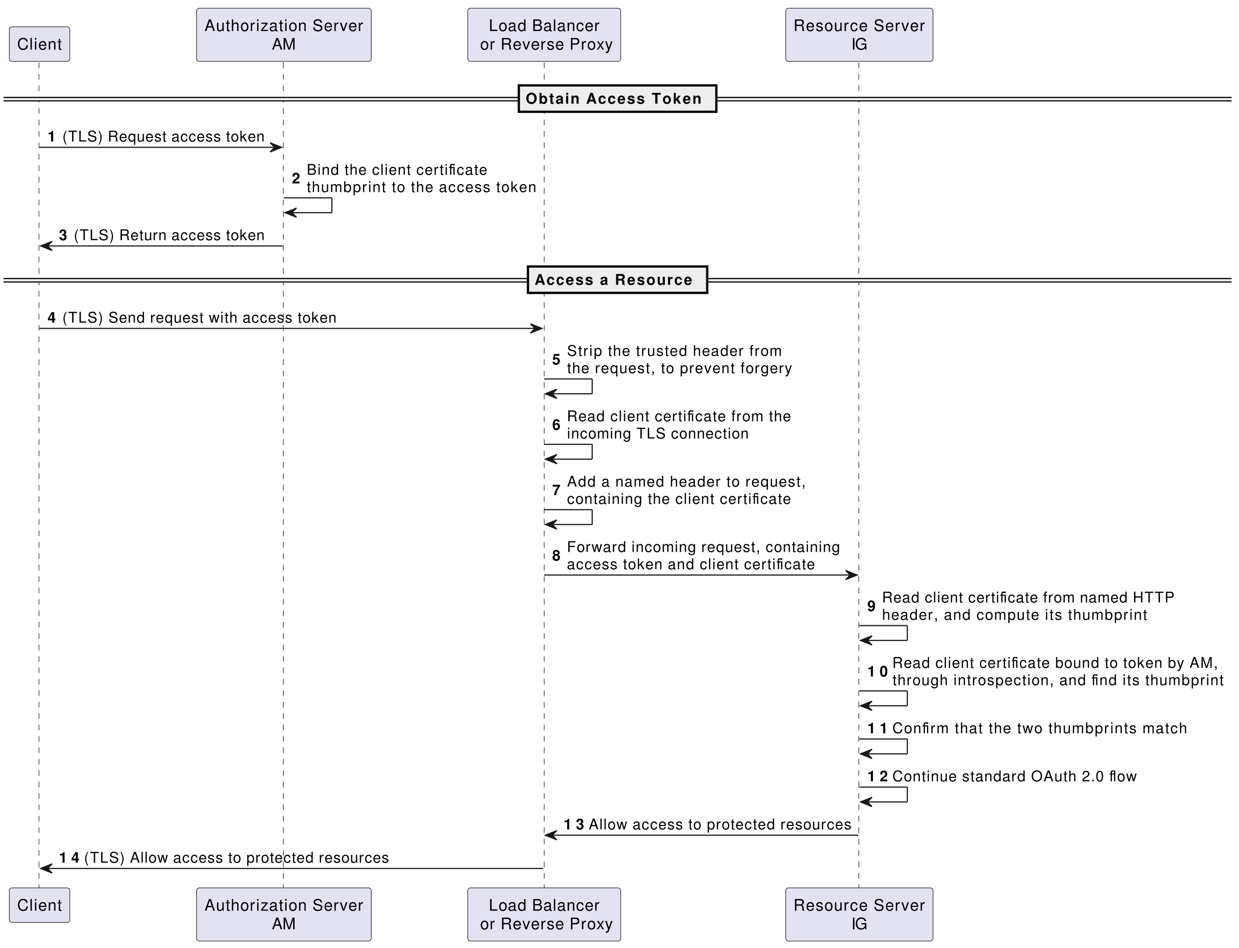
- Forward the encoded certificate to IG in the trusted header. Encode the certificate in an HTTP-header compatible format that can convey a full certificate, so that IG can rebuild the certificate.
- Strip the trusted header from incoming requests, and change the default header name to something an attacker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

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

Use this example when the IG instance is running behind a load balancer or other ingress point. If the IG instance is running behind the TLS termination point, consider the example in mTLS Using Standard TLS Client Certificate Authentication.

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


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


Client
Authorization Server AM
Load Balancer or Reverse Proxy
Resource Server IG
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 AM
Load Balancer or Reverse Proxy
Resource Server IG

Set up mTLS using trusted headers

1. Set up the keystores, truststores, AM, and IG as described in mTLS Using Standard TLS Client Certificate Authentication.
2. Base64-encode the value of `$oauth2_client_keystore_directory/client.cert.pem`. The value is used in the final POST.
3. Add the following route to IG:
    1. Linux
    2. Windows

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

```
%appdata%\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(decodeBase64(request.headers['ssl_client
_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 a Java certificate from the trusted header, computes the SHA-256 thumbprint of that certificate, and makes the thumbprint available for the ConfirmationKeyVerifierAccessTokenResolver.

4. Test the setup:

   a. Get an access token from AM, over TLS:

```
$ mytoken=$(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&grant_type=client_credentials&scope=test' \
https://am.example.com:8445/openam/oauth2/access_token
| jq -r .access_token)
```

   b. Introspect the access_token on AM:

```
$ curl --request POST \
-u ig_agent:password \
--header 'content-type: application/x-www-form-
urlencoded' \
--data token=${mytoken} \
http://am.example.com:8088/openam/oauth2/realms/root/in
trospect | jq

{
  "active": true,
  "scope": "test",
  "realm": "/",
  "client_id": "client-application",
  "user_id": "client-application",
  "token_type": "Bearer",
  "exp": 157...994,
  "sub": "(age!client-application)",
  "subname": "client-application",
  "iss": "http://am.example.com:8088/openam/oauth2",
  "cnf": {
    "x51...156": "1QG...Wgc"
  },
  "authGrantId": "lto...8vw",
  "auditTrackingId": "119...480"
}
```

The `cnf` property indicates the value of the confirmation code, as follows:

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

c. Access the IG route to validate the confirmation key, using the base64-encoded value of `$oauth2_client_keystore_directory/client.cert.pem` :

```
$ curl --request POST \
--header "authorization:Bearer $mytoken" \
--header 'ssl_client_cert:base64-encoded-cert'
http://ig.example.com:8080/mtls-header

Valid token: zw5...Sj1
  Confirmation keys: {
```

```
  ...
  }
```

The validated token and confirmation keys are displayed.

Use the OAuth 2.0 context to log in to the sample application

The introspection returns scopes in the context. This section contains an example route that retrieves the scopes, assigns them as the IG 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.

1. Set up AM:

   a. Set up AM as described in Validate access tokens through the introspection endpoint.

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

   c. 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", "{amDemoPw}")
   ```

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

   Do not 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 IG:

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

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

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

b. Add the following route to IG:

   1. Linux

   2. Windows

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

```
%appdata%\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&sc
ope=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 http://ig.example.com:8080/rs-pwreplay --
header "Authorization: Bearer ${mytoken}"
```

HTML for the sample application is displayed.

Cache access tokens

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

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

(From AM 6.5.3.) 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.

1. Set up AM as described in Validate access tokens through the introspection endpoint.
2. Set up IG:
   a. Set an environment variable for the IG agent password, and then restart IG:

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

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

   b. Add the following route to IG:
      1. Linux
      2. Windows

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

```
%appdata%\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 through the introspection endpoint:

- 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 IG’s interactions with AM.

3. 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&sc
ope=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 http://ig.example.com:8080/rs-introspect-cache -
-header "Authorization: Bearer ${mytoken}"

{
 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 IG calls AM to introspect the access token:

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

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

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

4. 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/t
oken/revoke"
```

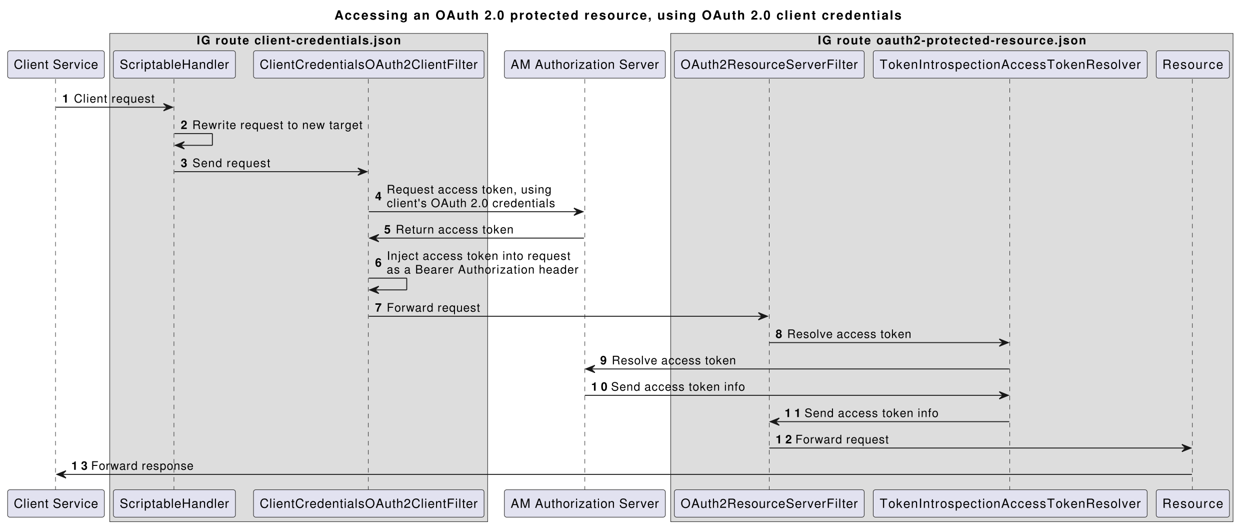
b. Access the route, using the access token returned in the previous step, and and note that the request is not authorized because the token is revoked:

```
$ curl -v http://ig.example.com:8080/rs-introspect-
cache --header "Authorization: Bearer ${mytoken}"
```

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

Using OAuth 2.0 client credentials

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
IG route oauth2-protected-resource.json
Client Service
ScriptableHandler
ClientCredentialsOAuth2ClientFilter
AM 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. Select **Applications** > **Agents** > **Identity Gateway**, and register an IG 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 an IG agent to AM.

      IMPORTANT

      IG 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. (From AM 6.5) On the **Advanced** tab, select the following value:

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

2. Set up IG:

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

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

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

b. Add the following route to IG:

1. Linux
2. Windows

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

```
%appdata%\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.

c. Add the following route to IG:

1. Linux

2. Windows

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

```
%appdata%\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);" ]
    }
  }
}
```

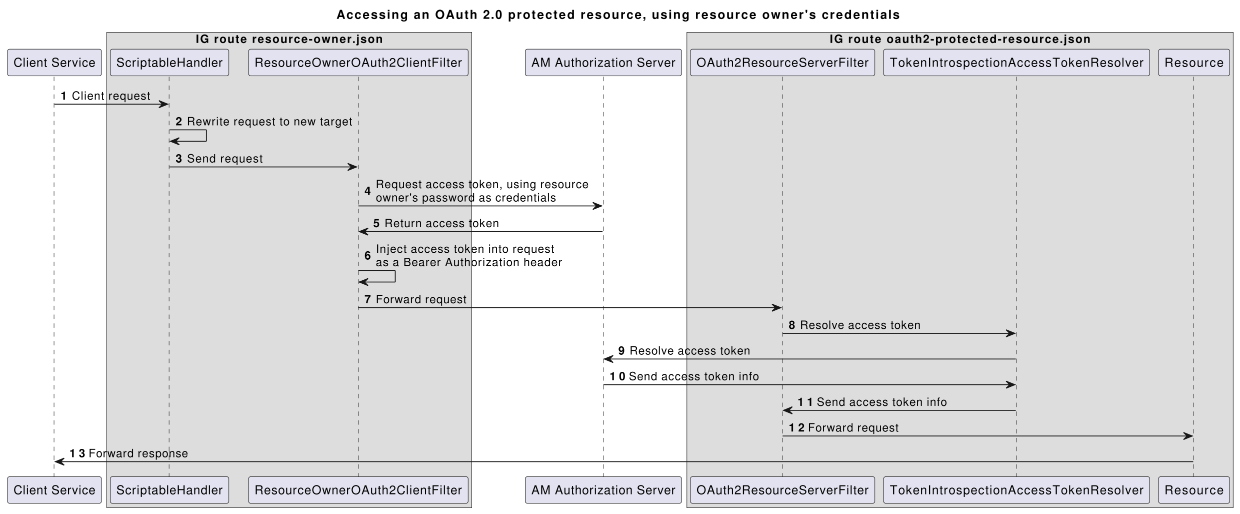
Note the following features of the route:

- 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 by accessing the route on http://ig.example.com:8080/client-credentials. A message shows that access is granted.

Using OAuth 2.0 resource owner password credentials

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
AM 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. According to information in the The OAuth 2.0 Authorization Framework, minimize use of this grant type and utilize other grant types whenever possible.

1. Set up the AM as an authorization server:

   a. Select **Applications** > **Agents** > **Identity Gateway**, and register an IG 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 an IG agent to AM.

      IMPORTANT

      IG 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. (From AM 6.5) On the **Advanced** tab, select the following value:

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

2. Set up IG:

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

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

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

   b. Add the following route to IG:

      1. Linux
      2. Windows

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

```
%appdata%\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.

c. Add the following route to IG:

1. Linux

2. Windows

```
$HOME/.openig/config/routes/resource-owner.json
```

```
%appdata%\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 by accessing the route on http://ig.example.com:8080/resource-owner. A message shows that access is granted.

OpenID Connect

The following sections provide an overview of how IG supports OpenID Connect 1.0, and examples of to set up IG as an OpenID Connect relying party in different deployment scenarios:

About IG with OpenID Connect

IG supports OpenID Connect 1.0, an authentication layer built on OAuth 2.0. OpenID Connect 1.0 is a specific implementation of OAuth 2.0, where the identity provider holds the protected resource that the third-party application wants to access. For more information, refer to OpenID Connect.

OpenID Connect refers 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 in order to present the correct inbox.

  As another example, an online shopping site needs to know which end user is accessing the site in order to present the right offerings, account, and shopping cart.

- *OpenID Provider* (OP): An OAuth 2.0 authorization server and also 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 OpenID Connect 1.0 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.

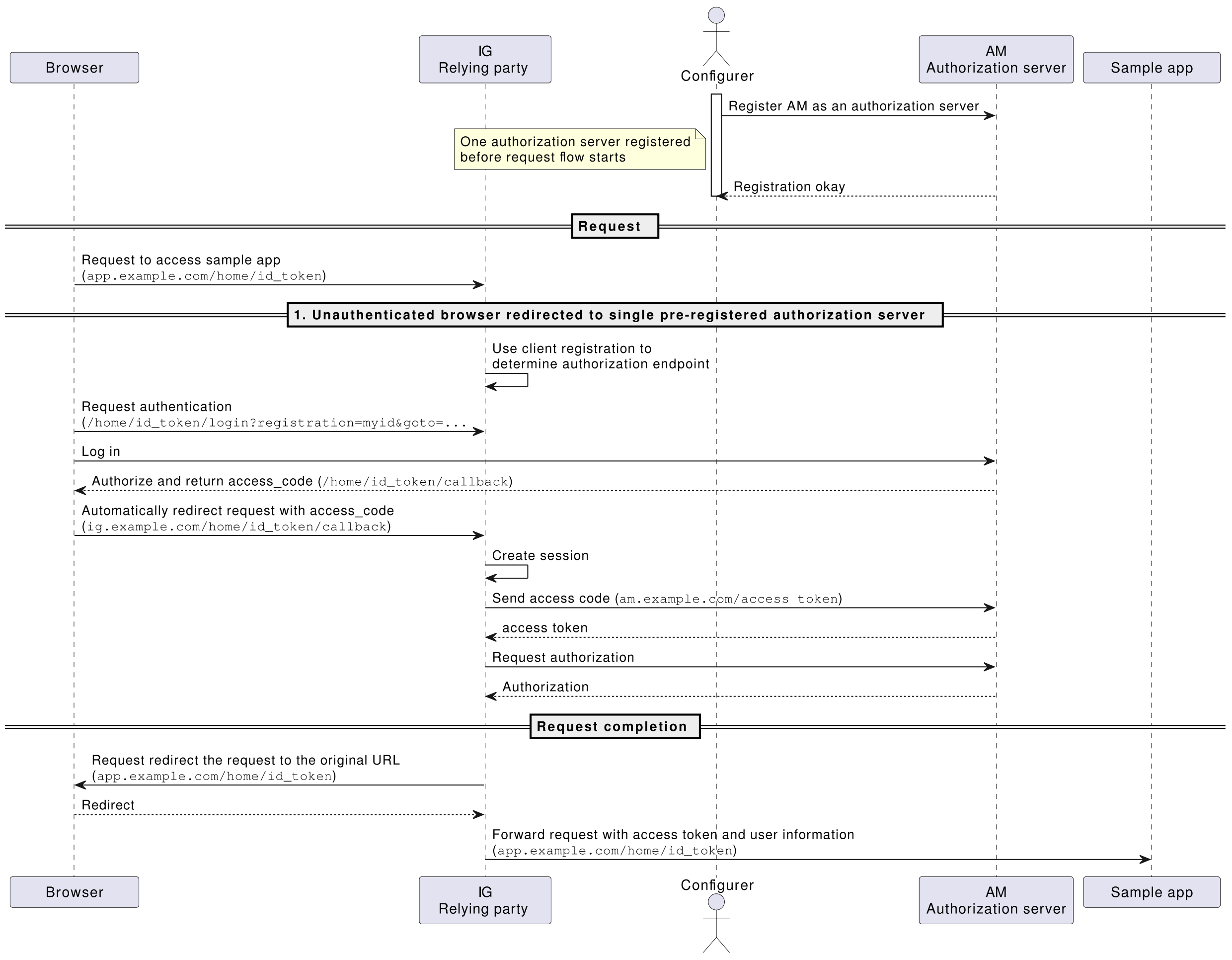
When IG acts as an OpenID Connect relying party, its role is to retrieve user information from the OpenID provider, and then to inject that information into the context for use by subsequent filters and handlers.

Use AM as a single OpenID Connect provider

This section gives an example of how to set up AM as an OpenID Connect identity provider, and IG 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, using AM as a single, preregistered OpenID Connect identity provider, and IG as a relying party:

Information flow for requests using AM as a single OpenID Connect identity provider



Browser
IG Relying party
Configurer
AM 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
Request redirect the request to the original URL (app.example.com/home/id_token)
Redirect
Forward request with access token and user information (app.example.com/home/id_token)

Before you start, prepare AM, IG, and the sample application as described in Example installation for this guide.

1. Set Up AM as an OpenID Connect provider:
   a. (From AM 6.5.3) 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/*?*`
   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**: `http://ig.example.com:8080/home/id_token/callback`
- **Scope(s)**: `openid`, `profile`, and `email`

iii. (From AM 6.5) On the **Advanced** tab, select the following values:

- **Grant Types**: `Authorization Code`

iv. On the **Signing and Encryption** tab, change **ID Token Signing Algorithm** to `HS256`, `HS384`, or `HS512`. The algorithm must be HMAC.

2. Set up IG:

a. Set an environment variable for `oidc_client`, and then restart IG:

```
$ export OIDC_SECRET_ID='cGFzc3dvcmQ='
```

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

1. Linux
2. Windows

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

```
%appdata%\OpenIG\config\routes\static-resources.json
```

```
{
  "name" : "sampleapp-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css')}",
  "handler": "ReverseProxyHandler"
}
```

c. Add the following route to IG:

1. Linux
2. Windows

```
$HOME/.openig/config/routes/07-openid.json
```

```
%appdata%\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 IG 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 IG to act as a relying party. It uses a single client registration that is defined inline and refers to the AM server configured in Use AM As a single OpenID Connect provider.
- 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 (OpenID Connect provider). This is why the redirect URI in the client profile in AM is set to `http://ig.example.com:8080/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. If you are logged in to AM, log out and clear any cookies.
   b. Go to http://ig.example.com:8080/home/id_token.

      The AM login page is displayed.
   c. 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.

Use multiple OpenID Connect providers

This section builds on the example in Use AM As a single OpenID Connect provider to give an example of using OpenID Connect with two identity providers.

The client registration for the AM provider is declared in the heap, and a second client registration defines Google as an alternative identity provider. The Nascar page helps the user to choose an identity provider.

1. Set up AM as the first OpenID Connect provider, as described in Use AM As a single OpenID Connect provider.
2. Set up Google as the second OpenID Connect identity provider, using the following hints:
    a. Go to https://console.cloud.google.com/apis/credentials.
    b. Create credentials for an OAuth 2.0 client ID with the following options:
        - **Application type**: `Web application`
        - **Authorized redirect URI**: `http://ig.example.com:8080/home/id_token/callback`
    c. Make a note of the ID and password for the Google identity provider.

3. Set up IG:

   a. Add the following route to IG, to serve .css and other static resources for the sample application:

      1. Linux

      2. Windows

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

```
%appdata%\OpenIG\config\routes\static-resources.json
```

```
{
  "name" : "sampleapp-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css')}",
  "handler": "ReverseProxyHandler"
}
```

   b. Add the following route to IG:

      1. Linux

      2. Windows

```
$HOME/.openig/config/routes/07-openid-nascar.json
```

```
%appdata%\OpenIG\config\routes\07-openid-nascar.json
```

```
{
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "openam",
      "type": "ClientRegistration",
      "config": {
        "clientId": "oidc_client",
        "clientSecretId": "oidc.secret.id",
        "issuer": {
          "name": "Issuer",
          "type": "Issuer",
          "config": {
            "wellKnownEndpoint":
"http://am.example.com:8088/openam/oauth2/.well-
known/openid-configuration"
          }
        },
        "scopes": [
          "openid",
          "profile",
          "email"
        ],
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "tokenEndpointAuthMethod":
"client_secret_basic"
      }
    },
    {
      "name": "google",
      "type": "ClientRegistration",
      "config": {
        "clientId": "googleClientId",
        "clientSecretId": "google.secret.id",
        "issuer": {
          "name": "accounts.google.com",
          "type": "Issuer",
          "config": {
            "wellKnownEndpoint":
"https://accounts.google.com/.well-known/openid-
configuration"
          }
        },
        "scopes": [
          "openid",
          "profile"
        ],
        "secretsProvider": "SystemAndEnvSecretStore-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=Issuer&goto=${urlEncode
QueryParameterNameOrValue('http://ig.example.com:8080/h
ome/id_token')}'>AM Login</a></p>",
          "    <p><a href='/home/id_token/login?
registration=googleClientId&issuer=accounts.google.com&
goto=${urlEncodeQueryParameterNameOrValue('http://ig.ex
ample.com:8080/home/id_token')}'>Google Login</a></p>",
          "  </body>",
          "</html>"
        ]
      }
    }
  ],
  "name": "07-openid-nascar",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path,
'^/home/id_token')}",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type":
"AuthorizationCodeOAuth2ClientFilter",
          "config": {
            "clientEndpoint": "/home/id_token",
            "failureHandler": {
              "type": "StaticResponseHandler",
              "config": {
                "comment": "Trivial failure handler for
debugging only",
                "status": 500,
                "headers": {
                  "Content-Type": [ "text/plain;
charset=UTF-8" ]
                },
                "entity": "${attributes.openid}"
              }
            },
            "loginHandler": "NascarPage",
            "registrations": [ "openam", "google" ],
            "requireHttps": false,
            "cacheExpiration": "disabled"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Consider the differences with `07-openid.json`:

- The heap objects `openam` and `google` define two client registrations to authenticate IG to identity providers.
- The heap object `NascarPage` is a StaticResponseHandler that provides links to the two client registrations.
- The AuthorizationCodeOAuth2ClientFilter uses a `loginHandler` that refers to `NascarPage` to allow users to choose from the two client registrations.

c. In the route, replace both occurrences of `googleClientId` by the Google identity provider ID.

d. Set environment variables for the identity providers' passwords:

i. Set an environment variable for the password of the AM identity provider, `oidc_client`:

```
$ export OIDC.SECRET.ID='cGFzc3dvcmQ='
```

ii. Set an environment variable for the password of the Google identity provider:

```
$ export GOOGLE.SECRET.ID='base64-encoded-google-
client-password'
```

The passwords are retrieved by the SystemAndEnvSecretStore, and must be base64-encoded.

4. Test the setup:

a. Log out of AM.

b. Go to http://ig.example.com:8080/home/id_token.

The Nascar page offers the choice of identity provider.

c. Select a provider, log in with your credentials, and then allow the application to access user information.

For AM, use the following credentials: username `demo`, password `Ch4ng31t`. For the Google identity provider, use the Google credentials.

The home page of the sample application is displayed.

Discover and dynamically register with OpenID Connect providers

OpenID Connect defines mechanisms for discovering and dynamically registering with an identity provider that is not 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 do not persist when IG is restarted.

This section builds on the example in Use AM As a single OpenID Connect provider to give an example of discovering and dynamically registering with an identity provider that is not 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 Use AM As a single OpenID Connect provider.

   b. 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.

   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 Whitelist**: `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 IG:

   a. In the IG configuration, set an environment variable for the keystore password, and then restart IG:

```
$ export KEYSTORE_SECRET_ID='a2V5c3RvcmU='
```

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

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

      1. Linux
      2. Windows

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

```
%appdata%\OpenIG\config\routes\static-resources.json
```

```
{
  "name" : "sampleapp-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css')}",
  "handler": "ReverseProxyHandler"
}
```

c. Add the following script to IG:

1. Linux
2. Windows

```
$HOME/.openig/scripts/groovy/discovery.groovy
```

```
%appdata%\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.router.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 IG. 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 IG, replacing `/path/to/secrets/keystore.p12` with your path:

   1. Linux
   2. Windows

```
$HOME/.openig/config/routes/07-discovery.json
```

```
%appdata%\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": "${attributes.openid}"
               }
             },
             "loginHandler": "DiscoveryPage",
             "discoverySecretId":
"private.key.jwt.signing.key",
             "tokenEndpointAuthMethod":
"private_key_jwt",
             "secretsProvider": "SecretsProvider-1",
             "metadata": {
               "client_name": "My Dynamically Registered
Client",
               "redirect_uris": [
"http://ig.example.com:8080/discovery/callback" ],
               "scopes": [ "openid", "profile", "email"
]
             }
           }
         },
         {
           "type": "StaticRequestFilter",
           "config": {
             "method": "POST",
             "uri": "http://app.example.com:8081/login",
             "form": {
               "username": [
                 "${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, IG uses the `discovery` value as the `resource` for a WebFinger request using OpenID Connect Discovery protocol.
- IG uses the `discovery` parameter to find an identity provider. IG 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 IG to the end user's browser, using the `goto` parameter, fter the process is complete and IG has injected the OpenID Connect 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.

Passing 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 **Applications** > **Agents** > **Identity Gateway** and register an IG agent with the following values:

      - **Agent ID**: `ig_agent`
      - **Password**: `password`

      For AM 6.5.x and earlier versions, register an agent as described in Register an IG agent in AM 6.5 and earlier.

      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 an IG agent to AM.

      IMPORTANT

      IG 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. (From AM 6.5.3) 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 IG:

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

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

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

b. Add the following route to IG:

1. Linux
2. Windows

```
$HOME/.openig/config/routes/user-profile-sso.json
```

```
%appdata%\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. Go to http://ig.example.com:8080/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 that is available in `rawInfo`:

```
username: demo
rawInfo: {_id=demo, _rev=273001616, employeeNumber=
[123], 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 **Applications** > **Agents** > **Identity Gateway** and register an IG agent with the following values:
      - **Agent ID**: `ig_agent`
      - **Password**: `password`

      For AM 6.5.x and earlier versions, register an agent as described in Register an IG agent in AM 6.5 and earlier.

      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 an IG agent to AM.

      IMPORTANT

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

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

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

   b. Add the following route to IG:

1. Linux
2. Windows

```
$HOME/.openig/config/routes/user-profile-ses-info.json
```

```
%appdata%\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, retrieve an SSO token:

```
$ curl --request POST \
--url
http://am.example.com:8088/openam/json/realms/root/auth
enticate \
--header 'accept-api-version: resource=2.0' \
--header 'content-type: application/json' \
--header 'x-openam-username: demo' \
--header 'x-openam-password: Ch4ng31t' \
--data '{}'

{"tokenId":"AQIC5wM2LY . . .
Dg5AAJTMQAA*","successUrl":"/openam/console"}
```

b. Access the route, providing the token ID retrieved in the previous step, where iPlanetDirectoryPro is the name of the AM session cookie:

```
$ curl --cookie 'iPlanetDirectoryPro=tokenID'
http://ig.example.com:8080/user-profile-ses-info | jq .

{
  "username": "demo",
  "user_profile": {
    "_id": "demo",
    "_rev": "123...456",
    "employeeNumber": ["123"],
    "mail": ["demo@example.com"],
    "username": "demo"
  }
}
```

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 that is available in `asJsonValue()`.

Retrieving 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.

1. Set up AM as described in Validate access tokens through the introspection endpoint.
2. Set up IG:
   a. Set an environment variable for the IG agent password, and then restart IG:

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

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

b. Add the following route to IG:

1. Linux
2. Windows

```
$HOME/.openig/config/routes/user-profile-oauth.json
```

```
%appdata%\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()} }"
        }
      }
    }
  }
}
```

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&sc
ope=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 http://ig.example.com:8080/user-profile-oauth
 --header "Authorization: Bearer ${mytoken}" | 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. (From AM 6.5.3) 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/*?*`

    b. Select **Applications** > **Agents** > **Identity Gateway** and register an IG agent with the following values:

    - **Agent ID**: `ig_agent`
    - **Password**: `password`

      For AM 6.5.x and earlier versions, register an agent as described in Register an IG agent in AM 6.5 and earlier.

      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 an IG agent to AM.

    IMPORTANT

    IG 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 IG:

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

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

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

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

    1. Linux
    2. Windows

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

```
%appdata%\OpenIG\config\routes\static-resources.json
```

```
{
  "name" : "sampleapp-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css')}",
  "handler": "ReverseProxyHandler"
}
```

c. Add the following route to IG, replacing value of the property `secretsDir` with the directory for the PEM file:

1. Linux
2. Windows

```
$HOME/.openig/config/routes/jwt-builder-sign-pem.json
```

```
%appdata%\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 does not 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. If you are logged in to AM, log out and clear any cookies.
   b. Go to http://ig.example.com:8080/jwt-builder-sign-pem.
   c. Log in to AM as user `demo`, password `Ch4ng31t`, or as another user. The sample app displays the signed JWT along with its header and payload.
   d. 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. (From AM 6.5.3) 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/*?*`

b. Select **Applications** > **Agents** > **Identity Gateway** and register an IG agent with the following values:

- **Agent ID**: `ig_agent`
- **Password**: `password`

For AM 6.5.x and earlier versions, register an agent as described in Register an IG agent in AM 6.5 and earlier.

> 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 an IG agent to AM.

> IMPORTANT
>
> IG 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 IG:

   a. In IG, 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='ZW5jcnlwdGVkcGFzc3dvc
   mQ='
   ```

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

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

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

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

      1. Linux
      2. Windows

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

   ```
   %appdata%\OpenIG\config\routes\static-resources.json
   ```

   ```
   {
     "name" : "sampleapp-resources",
     "baseURI" : "http://app.example.com:8081",
   ```

```
  "condition": "${find(request.uri.path,'^/css')}",
  "handler": "ReverseProxyHandler"
}
```

d. Add the following route to IG, replacing the value of `secretsDir` with your secrets directory:

   1. Linux

   2. Windows

```
$HOME/.openig/config/routes/jwtbuilder-sign-then-
encrypt.json
```

```
%appdata%\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 IG 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. If you are logged in to AM, log out and clear any cookies.

b. Go to http://ig.example.com:8080/jwtbuilder-sign-then-encrypt.

c. 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.

d. 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.

e. 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. (From AM 6.5.3) 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/*?*`

   b. Select **Applications** > **Agents** > **Identity Gateway** and register an IG agent with the following values:

      - **Agent ID**: `ig_agent`

- **Password**: `password`

  For AM 6.5.x and earlier versions, register an agent as described in Register an IG agent in AM 6.5 and earlier.

  > 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 an IG agent to AM.

> IMPORTANT
>
> IG 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 IG:

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

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

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

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

1. Linux
2. Windows

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

```
%appdata%\OpenIG\config\routes\static-resources.json
```

```
{
  "name" : "sampleapp-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css')}",
  "handler": "ReverseProxyHandler"
}
```

c. Add the following route to IG, replacing the value of the property `secretsDir` with your value:

1. Linux
2. Windows

```
$HOME/.openig/config/routes/jwtbuilder-encrypt-
symmetric.json
```

```
%appdata%\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. If you are logged in to AM, log out and clear any cookies.
   b. Go to http://ig.example.com:8080/jwtbuilder-encrypt-symmetric.
   c. Log in to AM as user `demo`, password `Ch4ng31t`, or as another user. The JWT is displayed in the sample app.
   d. 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. (From AM 6.5.3) 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/*?*`

   b. Select **Applications** > **Agents** > **Identity Gateway** and register an IG agent with the following values:

      - **Agent ID**: `ig_agent`
      - **Password**: `password`

      For AM 6.5.x and earlier versions, register an agent as described in Register an IG agent in AM 6.5 and earlier.

      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 an IG agent to AM.

IMPORTANT

IMPORTANT

IG 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 IG:

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

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

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

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

      1. Linux
      2. Windows

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

   ```
   %appdata%\OpenIG\config\routes\static-resources.json
   ```

   ```
   {
     "name" : "sampleapp-resources",
     "baseURI" : "http://app.example.com:8081",
     "condition": "${find(request.uri.path,'^/css')}",
     "handler": "ReverseProxyHandler"
   }
   ```

   c. Add the following route to IG, replacing value of the property `secretsDir` with the directory for the PEM file:

      1. Linux
      2. Windows

   ```
   $HOME/.openig/config/routes/jwtbuilder-encrypt-
   asymmetric.json
   ```

   ```
   %appdata%\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. If you are logged in to AM, log out and clear any cookies.

   b. Go to http://ig.example.com:8080/jwtbuilder-encrypt-asymmetric.

   c. Log in to AM as user `demo`, password `Ch4ng31t`, or as another user. The JWT is displayed in the sample app.

   d. 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

The IG 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 Identity Cloud or AM.
- *Service Provider* (SP): The service that a user wants to access. IG 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.

SAML assertions can be signed and encrypted. ForgeRock recommends using *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 in deployments with multiple instances of IG

IG uses the federation libraries from AM (also referred to as the Fedlet) to implement SAML. When IG acts as a SAML service provider, the session information is stored in the fedlet, not the session cookie. In deployments that use multiple instances of IG as a SAML service provider, it is therefore 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*.

Federation using the SamlFederationFilter

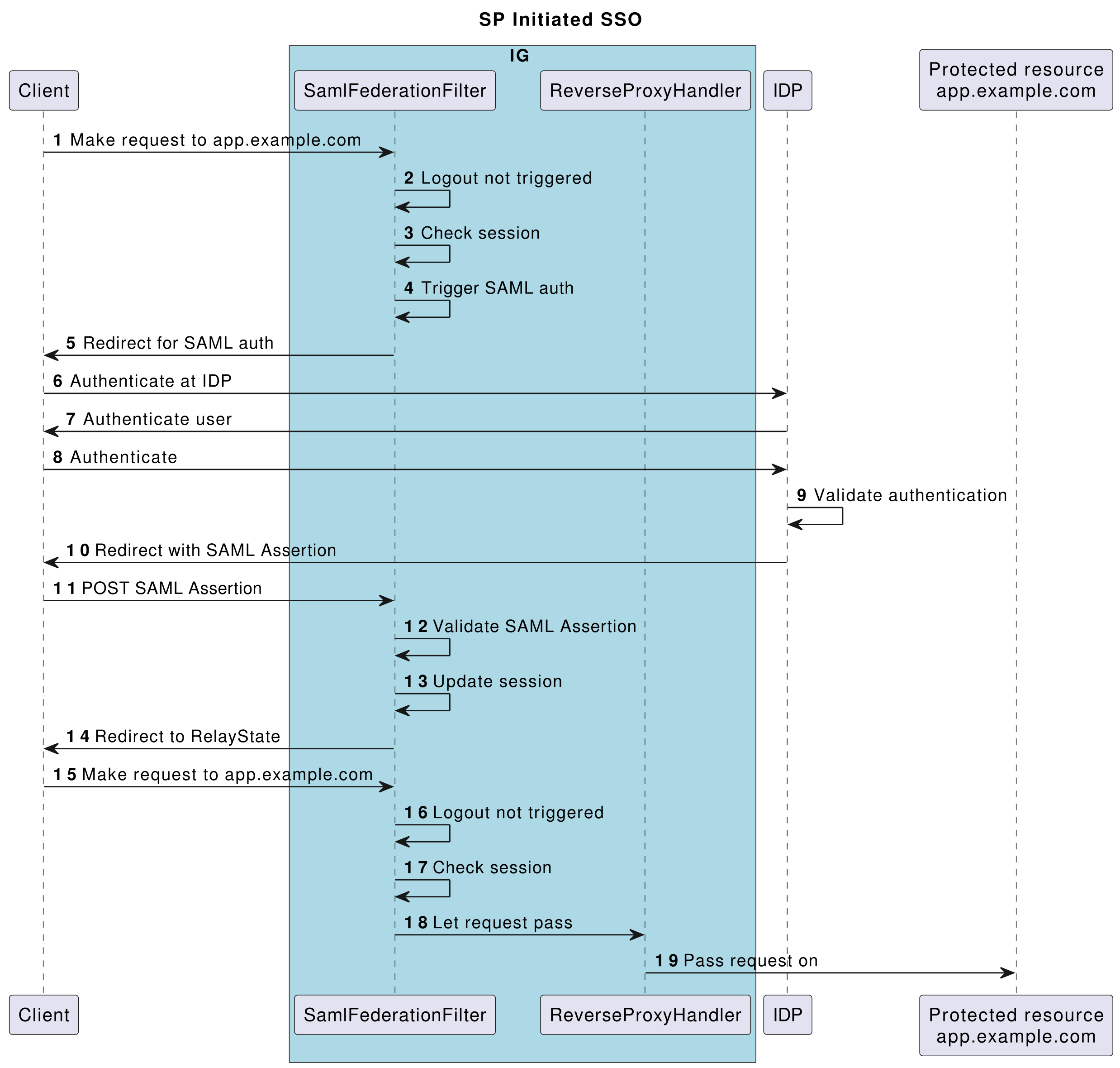
About SP-initiated SSO with the SamlFederationFilter

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.

For the SamlFederationFilter, SP-initiated SSO is the preferred 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 IG acts as a SAML 2.0 SP:



Set up federation with unsigned/unencrypted assertions with the SamlFederationFilter

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 IG, 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-7.3.0.zip`, delivered with the AM installation, into a local directory.

```
$ unzip $HOME/openam/Fedlet-7.3.0.zip

Archive:  Fedlet-7.3.0.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/home/saml`
`fedletcot` and `FEDLET_COT`	`Circle of Trust`

Replace this	With this
`sp.example.com:8080/home/saml/fedletapplication`	`sp.example.com:8080/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 Use a 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/export
metadata.jsp?entityid=openam"
```

The `idp.xml` file is created locally.

4. Set up IG:

   a. Copy the edited fedlet files, and the exported `idp.xml` file into the IG 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. Add the following route to IG, to serve .css and other static resources for the sample application:

      1. Linux
      2. Windows

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

```
%appdata%\OpenIG\config\routes\static-resources.json
```

```
{
  "name" : "sampleapp-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css')}",
  "handler": "ReverseProxyHandler"
}
```

c. Add the following route to IG:

1. Linux

2. Windows

```
$HOME/.openig/config/routes/saml-filter.json
```

```
%appdata%\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.

d. Restart IG.

5. Test the setup:

   a. If you are logged in to AM, log out and clear any cookies.

   b. Go to http://sp.example.com:8080/home.

   c. Log in to AM as user `demo`, password `Ch4ng31t`. The request is redirected to the sample application.

Set up federation with signed/encrypted assertions with the SamlFederationFilter

1. Set up the example in saml.adoc#federation-setup-filter.
2. Set up the SAML keystore:

   a. Find the values of AM's default SAML keypass and storepass:

```
$ more /path/to/am/security/secrets/default/.keypass
$ more /path/to/am/security/secrets/default/.storepass
```

   b. Copy the SAML keystore from the AM configuration to IG:

```
$ cp /path/to/am/security/keystores/keystore.jceks
/path/to/ig/keystore.jceks
```

WARNING

Legacy keystore types such as JKS and JCEKS are supported but are not secure. Consider using the PKCS#12 keystore type.

3. Configure the Fedlet in IG:

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 IG
   a. In the IG configuration, set environment variables for the following secrets, and then restart IG:

   ```
   $ 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:
      1. Linux
      2. Windows

   ```
   $HOME/.openig/config/routes/saml-filter-secure.json
   ```

   ```
   %appdata%\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 IG.

6. Test the setup:

   a. If you are logged in to AM, log out and clear any cookies.

   b. Go to http://sp.example.com:8080/home.

   c. Log in to AM as user `demo`, password `Ch4ng31t`. The request is redirected to the sample application.

Federation using the SamlFederationHandler (deprecated)

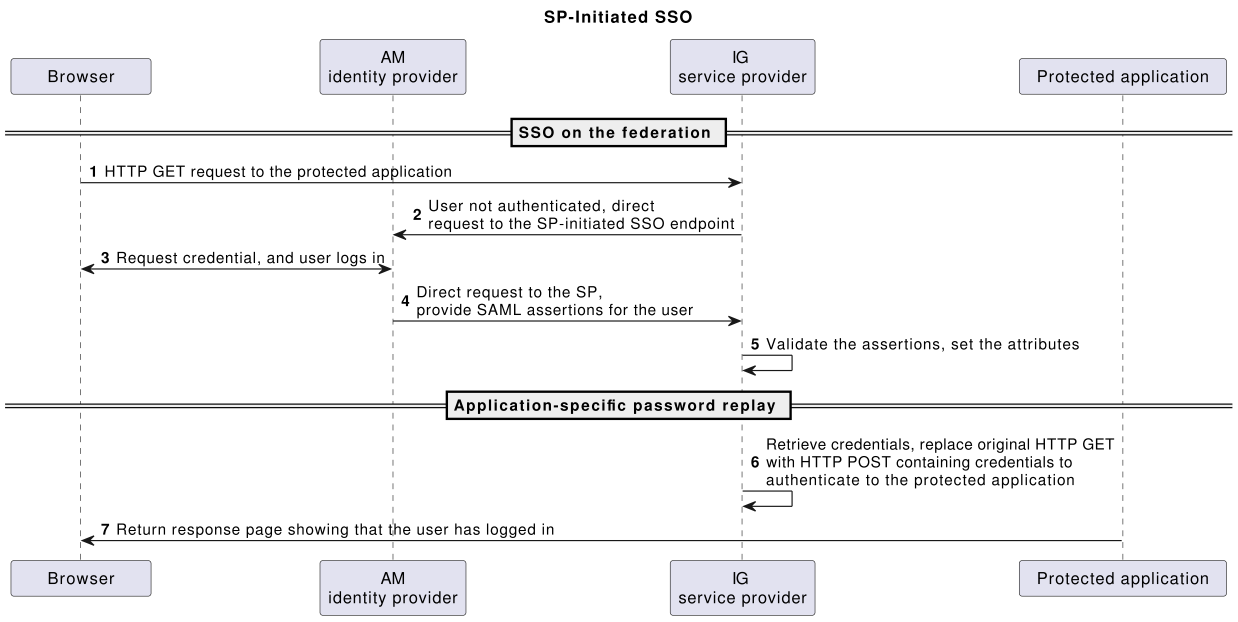
IMPORTANT

The SamlFederationHandler is deprecated; use the SamlFederationFilter instead. For more information, refer to the Deprecated section of the *Release Notes*.

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 IG acts as a SAML 2.0 SP:


SP-Initiated SSO
Browser
AM identity provider
IG 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
Browser
AM identity provider
IG service provider
Protected application

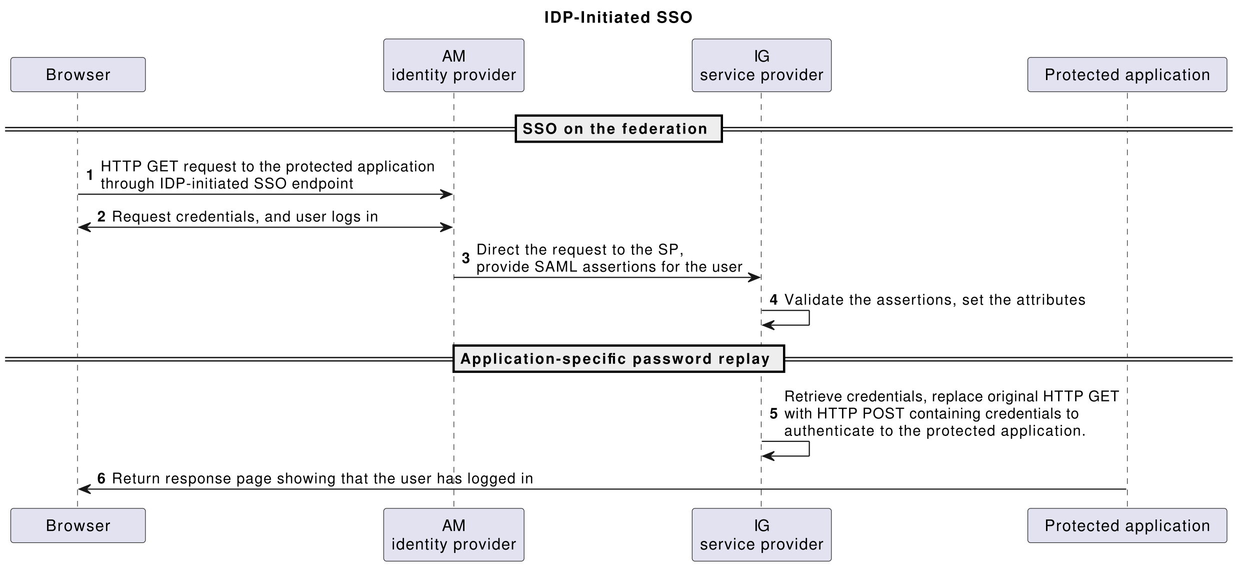
About IDP-initiated SSO the the SamlFederationHandler

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 IG acts as a SAML 2.0 SP:


IDP-Initiated SSO
Browser
AM identity provider
IG 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

Set up federation with unsigned/unencrypted assertions with the SamlFederationHandler

For examples of the federation configuration files, refer to Example fedlet files. To set up multiple SPs, work through this section, and then 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 IG, 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-7.3.0.zip`, delivered with the AM installation, into a local directory.

```
$ unzip $HOME/openam/Fedlet-7.3.0.zip

Archive:  Fedlet-7.3.0.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 Use a 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/export
metadata.jsp?entityid=openam"
```

      The `idp.xml` file is created locally.

4. Set up IG:

   a. Copy the edited fedlet files, and the exported `idp.xml` file into the IG 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 IG, to serve .css and other static resources for the sample application:

1. Linux
2. Windows

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

```
%appdata%\OpenIG\config\routes\static-resources.json
```

```
{
  "name" : "sampleapp-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css')}",
  "handler": "ReverseProxyHandler"
}
```

d. Add the following route to IG:

1. Linux
2. Windows

```
$HOME/.openig/config/routes/saml-handler.json
```

```
%appdata%\OpenIG\config\routes\saml-handler.json
```

```
{
  "name": "saml-handler",
  "condition": "${find(request.uri.path, '^/saml')}",
  "session": "JwtSession",
  "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.
- The route uses the `JwtSession` implementation, meaning it stores encrypted session information in a browser cookie. The name is a reference to the `JwtSession` object defined in `config.json`. For information, see JwtSession.

e. Add the following route to IG:

1. Linux
2. Windows

```
$HOME/.openig/config/routes/federate-handler.json
```

```
%appdata%\OpenIG\config\routes\federate-handler.json
```

```
{
  "name": "federate-handler",
  "condition": "${find(request.uri.path,
'^/home/federate')}",
  "session": "JwtSession",
  "baseURI": "http://app.example.com:8081",
  "handler": {
    "type": "DispatchHandler",
    "config": {
      "bindings": [
        {
          "condition": "${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.
- The route uses the `JwtSession` implementation, meaning it stores encrypted session information in a browser cookie. The name is a reference to the `JwtSession` object defined in `config.json`. For information, see JwtSession.

f. Restart IG.

1. 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`.

   IG returns the response page showing that the the demo user has logged in.

TIP

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.o
riginalUri)}"
  ]
}
```

Set up federation with signed/encrypted assertions with the SamlFederationHandler

1. Set up the example in saml.adoc#federation-setup-handler.
2. Set up the SAML keystore:
   a. Find the values of AM's default SAML keypass and storepass:

   ```
   $ more /path/to/am/security/secrets/default/.keypass
   $ more /path/to/am/security/secrets/default/.storepass
   ```

   b. Copy the SAML keystore from the AM configuration to IG:

   ```
   $ cp /path/to/am/security/keystores/keystore.jceks
   /path/to/ig/keystore.jceks
   ```

   WARNING

   Legacy keystore types such as JKS and JCEKS are supported but are not secure. Consider using the PKCS#12 keystore type.

3. Configure the Fedlet in IG:

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 IG:
   a. In the IG configuration, set environment variables for the following secrets, and then restart IG:

   ```
   $ 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:
      1. Linux
      2. Windows

   ```
   $HOME/.openig/config/routes/saml-handler-secure.json
   ```

   ```
   %appdata%\OpenIG\config\routes\saml-handler-secure.json
   ```

   ```
   {
     "name": "saml-handler-secure",
     "condition": "${find(request.uri.path, '^/saml')}",
   ```

```
  "session": "JwtSession",
  "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 IG.

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`.

IG returns the response page showing that the the demo user has logged in.

SAML 2.0 and multiple applications with the SamlFederationHandler

The chapter extends the example in SAML with the service provider `sp`, 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 IG, configure the service provider files for `sp2`, using the files you created in Configure a Java 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 IG.

3. In AM, set up a remote service provider for `sp2`, as described in Set up federation with unsigned/unencrypted assertions:

    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 IG:

    1. Linux

    2. Windows

```
$HOME/.openig/config/routes/saml-handler-sp2.json
```

```
%appdata%\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"
    }
  }
}
```

1. Linux
2. Windows

```
$HOME/.openig/config/routes/federate-handler-sp2.json
```

```
%appdata%\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`.

      IG returns the response page showing that the user has logged in.

Use a non-transient NameID format

By default, AM as an IDP uses the NameID format `urn:oasis:names:tc:SAML:2.0:nameid-format:transient`. For more information, refer to Hosted identity provider configuration properties in AM's *SAML v2.0 guide*.

When the IDP uses another NameID format, configure IG 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 IG supports the `persistent` NameID format, IG does not 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 login 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, then AM defaults to `transient`, and IG defaults to `persistent`.

Example fedlet files

File	Description
`FederationConfig.properties`	Fedlet properties
`fedlet.cot`	Circle of trust for IG and the IDP
`idp.xml`	Standard metadata for the IDP
`idp-extended.xml`	Metadata extensions for the IDP
`sp.xml`	Standard metadata for the IG SP
`sp-extended.xml`	Metadata extensions for the IG SP

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 2016-2023 ForgeRock AS.

# 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.p
lugin.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.identi
ty.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.o
penam.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.
impl.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.
session.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.identit
y.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.identit
y.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.identit
y.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.i
dentity.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/keys
tores/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.
fedlet.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=1
6384

# 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%/de
bug

# 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.ut
il.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.sha
red.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.JCEEn
cryption

# 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 IG 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:protoc
ol">
        <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/m
etaAlias/idp"/>
        <SingleLogoutService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect"
Location="http://am.example.com:8088/openam/IDPSloRedirect/met
aAlias/idp"
ResponseLocation="http://am.example.com:8088/openam/IDPSloRedi
rect/metaAlias/idp"/>
        <SingleLogoutService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
Location="http://am.example.com:8088/openam/IDPSloPOST/metaAli
as/idp"
ResponseLocation="http://am.example.com:8088/openam/IDPSloPOST
/metaAlias/idp"/>
        <SingleLogoutService
```

```
Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP"
Location="http://am.example.com:8088/openam/IDPSloSoap/metaAli
as/idp"/>
        <ManageNameIDService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect"
Location="http://am.example.com:8088/openam/IDPMniRedirect/met
aAlias/idp"
ResponseLocation="http://am.example.com:8088/openam/IDPMniRedi
rect/metaAlias/idp"/>
        <ManageNameIDService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
Location="http://am.example.com:8088/openam/IDPMniPOST/metaAli
as/idp"
ResponseLocation="http://am.example.com:8088/openam/IDPMniPOST
/metaAlias/idp"/>
        <ManageNameIDService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP"
Location="http://am.example.com:8088/openam/IDPMniSoap/metaAli
as/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/metaAl
ias/idp"/>
        <SingleSignOnService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
Location="http://am.example.com:8088/openam/SSOPOST/metaAlias/
idp"/>
        <SingleSignOnService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP"
Location="http://am.example.com:8088/openam/SSOSoap/metaAlias/
idp"/>
```

```
        <NameIDMappingService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP"
Location="http://am.example.com:8088/openam/NIMSoap/metaAlias/
idp"/>
        <AssertionIDRequestService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP"
Location="http://am.example.com:8088/openam/AIDReqSoap/IDPRole
/metaAlias/idp"/>
        <AssertionIDRequestService
Binding="urn:oasis:names:tc:SAML:2.0:bindings:URI"
Location="http://am.example.com:8088/openam/AIDReqUri/IDPRole/
metaAlias/idp"/>
    </IDPSSODescriptor>
</EntityDescriptor>
```

idp-extended.xml

The following example of `$HOME/.openig/SAML/idp-extended.xml` defines an AM-specific 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 ForgeRock AS.
-->
<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, 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" />
```

The following example of `$HOME/.openig/SAML/sp.xml` defines a SAML configuration file for the IG 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 ForgeRock AS.
-->
<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:protoc
ol">
        <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/fedletSloRed
irect"/>
        <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/fedletSloPOS
```

```
T"/>
        <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/me
taAlias/sp"/>
        <AssertionConsumerService index="1"
Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Artifact"
Location="http://sp.example.com:8080/saml/fedletapplication/me
taAlias/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:protoc
ol">
    </XACMLAuthzDecisionQueryDescriptor>
</EntityDescriptor>
```

sp-extended.xml

The following example of `$HOME/.openig/SAML/sp-extended.xml` defines an AM-specific 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 ForgeRock AS.
-->
<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.DefaultLibrarySPAccountM
apper</Value>
        </Attribute>
        <Attribute name="spAttributeMapper">

<Value>com.sun.identity.saml2.plugins.DefaultSPAttributeMapper
</Value>
        </Attribute>
        <Attribute name="spAuthncontextMapper">

<Value>com.sun.identity.saml2.plugins.DefaultSPAuthnContextMap
per</Value>
        </Attribute>
        <Attribute name="spAuthncontextClassrefMapping">

<Value>urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtecte
dTransport|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>
        </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>
```

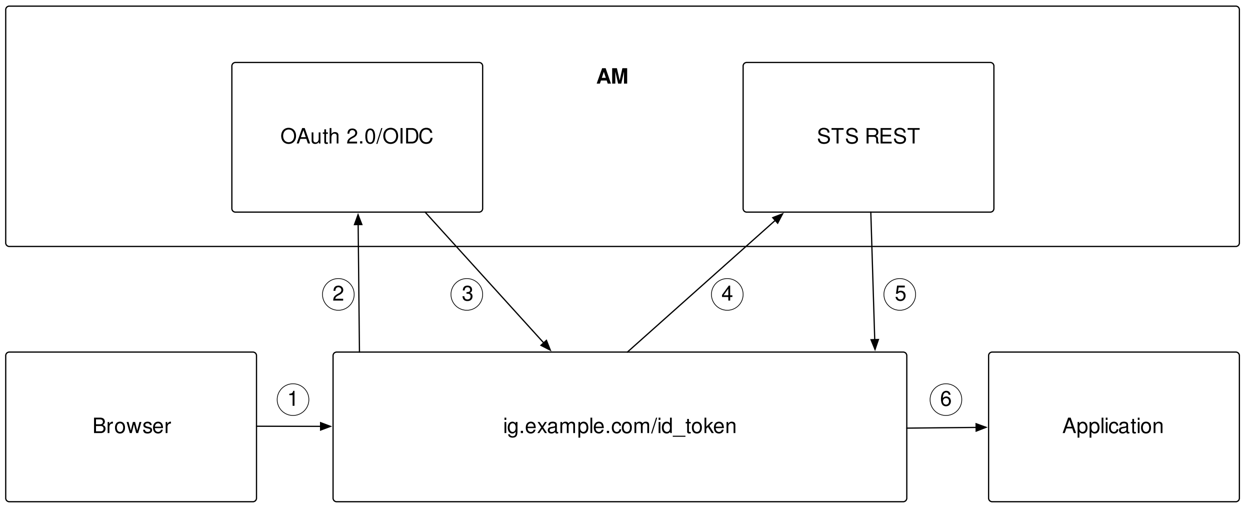
Token transformation

Transform OpenID Connect ID tokens into SAML assertions

This chapter builds on the example in OpenID Connect to transform OpenID Connect ID tokens into SAML 2.0 assertions.

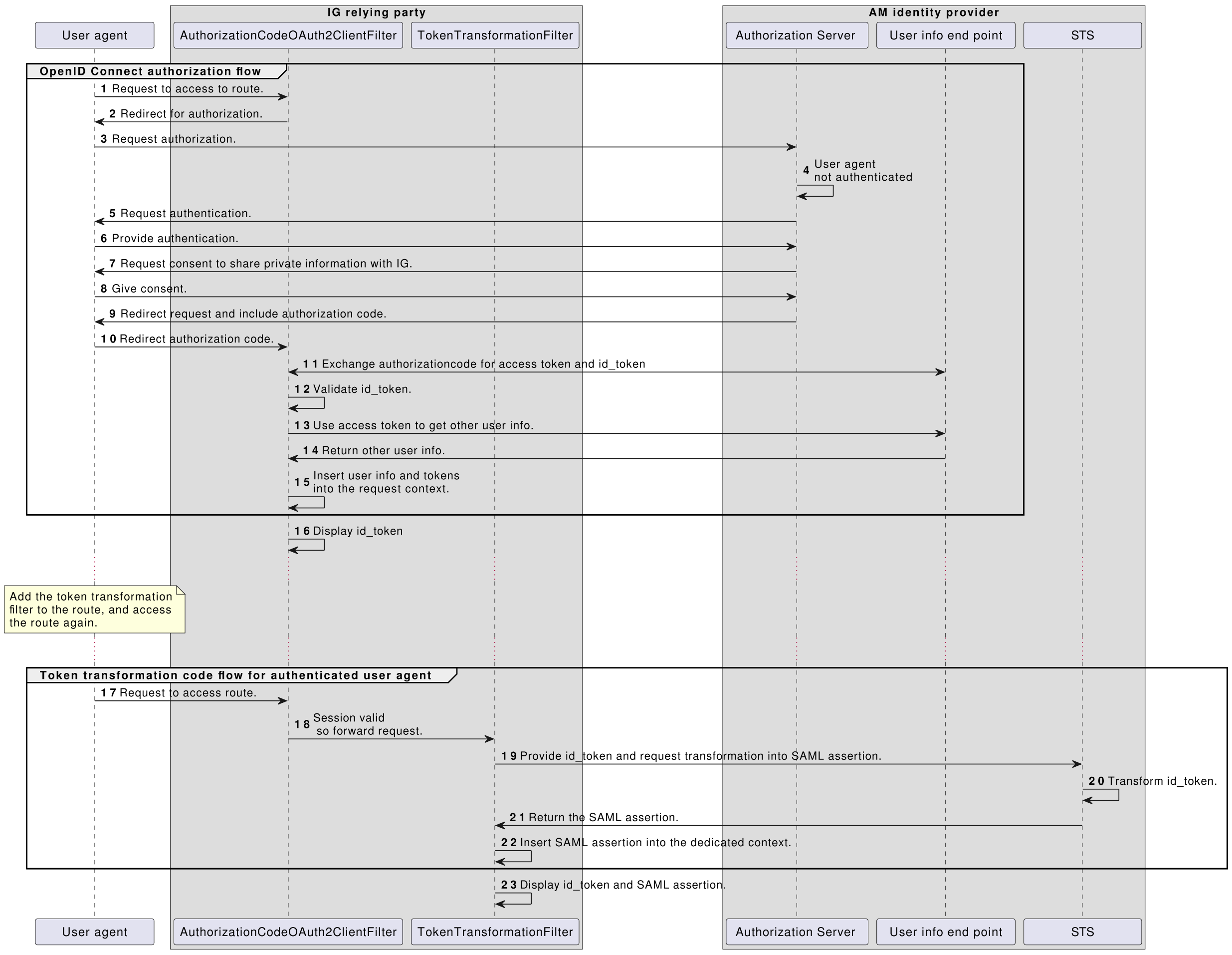
Many enterprises use existing or legacy, SAML 2.0-based SSO, but many mobile and social applications are managed by OpenID Connect. Use the IG TokenTransformationFilter to bridge the gap between OpenID Connect and SAML 2.0 frameworks.

The following figure illustrates the data flow:



1. A user tries to access to a protected resource.
2. If the user is not authenticated, the AuthorizationCodeOAuth2ClientFilter redirects the request to AM. After authentication, AM asks for the user's consent to give IG 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 that the user issued the transaction. The STS then issues a SAML assertion to IG on behalf of the user.
6. The TokenTransformationFilter makes the result of the token transformation available to downstream handlers in the `issuedToken` property of the `${contexts.sts}` context.

The following sequence diagram shows a more detailed view of the flow:


IG 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 IG.
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.

1. Set up an AM Security Token Service (STS), where the subject confirmation method is Bearer. For more information about setting up a REST STS instance, see AM's Security Token Service (STS) guide.

   a. Set up AM as described in

      - Use AM As a single OpenID Connect provider

   b. Select **Applications** > **Agents** > **Identity Gateway** and register an IG agent with the following values:

      - **Agent ID**: `ig_agent`
      - **Password**: `password`

      For AM 6.5.x and earlier versions, register an agent as described in Register an IG agent in AM 6.5 and earlier.

      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 an IG agent to AM.

IMPORTANT

IMPORTANT

IG 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. Create a Bearer Module:

   i. In the top level realm, select 👤 **Authentication** > **Modules**, and add a module with the following values:

      - **Module name** : `oidc`
      - **Type** : `OpenID Connect id_token bearer`

   ii. In the configuration page, enter the following values:

      - **OpenID Connect validation configuration type** : `Client Secret`
      - **OpenID Connect validation configuration value** : `password`

        This is the password of the OAuth 2.0/OpenID Connect client.
      - **Client secret** : `password`
      - **Name of OpenID Connect ID Token Issuer** : `http://am.example.com:8088/openam/oauth2`
      - **Audience name** : `oidc_client`

        This is the name of the OAuth 2.0/OpenID Connect client.
      - **List of accepted authorized parties** : `oidc_client`

        Leave all other values as default, and save your settings.

e. Create an instance of STS REST.

   i. In the top level realm, select **STS**, and add a Rest STS instance with the following values:

      - **Deployment URL Element** : `openig`

        This value identifies the STS instance and is used by the `instance` parameter in the TokenTransformationFilter.
      - **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** : Select `urn:oasis:names:tc:SAML:2.0:nameid-format:transient`
- **OpenID Connect Token**
  - **OpenID Connect Token Provider Issuer Id** : `oidc`
  - **Token signature algorithm** : Enter a value that is consistent with Use AM As a single OpenID Connect provider, for example, `HMAC SHA 256`
  - **Client Secret** : `password`
  - **Issued Tokens Audience** : `oidc_client`

ii. On the **SAML 2 Token** tab, add the following **Attribute Mappings**:

- **Key** : `userName` , **Value** : `uid`
- **Key** : `password` , **Value** : `mail`

f. Log out of AM.

2. Set up IG:

a. Set an environment variable for `oidc_client` and `ig_agent` , and then restart IG:

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

b. Add the following route to IG:

1. Linux
2. Windows

```
$HOME/.openig/config/routes/50-idtoken.json
```

```
%appdata%\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"
                    }
                  },
                  "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.issuedTok
en}\"}"
        }
      }
    }
  }
}
```

For information about how to set up the IG route in Studio, refer to 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 IG to act as an OpenID Connect relying party:
  - The client endpoint is set to `/home/id_token`, so the service URIs for this filter on the IG server are `/home/id_token/login`, `/home/id_token/logout`, and `/home/id_token/callback`.
  - 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` is true.
  - The target for storing authorization state information is `${attributes.openid}`. Subsequent filters and handlers can find access tokens and user information at this target.

  - The ClientRegistration holds configuration provided in Use AM As a single OpenID Connect provider, and used by IG to connect with AM.
  - The TokenTransformationFilter transforms an id_token into a SAML assertion:
    - The `id_token` parameter defines where this filter gets the id_token created by the `AuthorizationCodeOAuth2ClientFilter`.

      The TokenTransformationFilter makes the result of the token transformation available to downstream handlers in the `issuedToken` property of the `${contexts.sts}` context.
    - The `instance` parameter must match the `Deployment URL Element` for the REST STS instance.

      Errors that occur during token transformation cause an error response to be returned to the client and an error message to be logged for the IG administrator.
  - When the request succeeds, a StaticResponseHandler retrieves and displays the id_token from the target `{attributes.openid.id_token}`.

3. Test the setup:
   a. Go to http://ig.example.com:8080/home/id_token.

      The AM login screen is displayed.

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

      An OpenID Connect request to access private information is displayed.

   c. Select **Allow**.

      The id_token and SAML assertions are displayed:

```
{"id_token": "eyA . . ."}

{"saml_assertions": "<\"saml:Assertion
xmlns:saml=\"urn:oasis:names:tc:SAML:2.0:assertion\"
Version= . . ."}
```

OAuth 2.0 token exchange

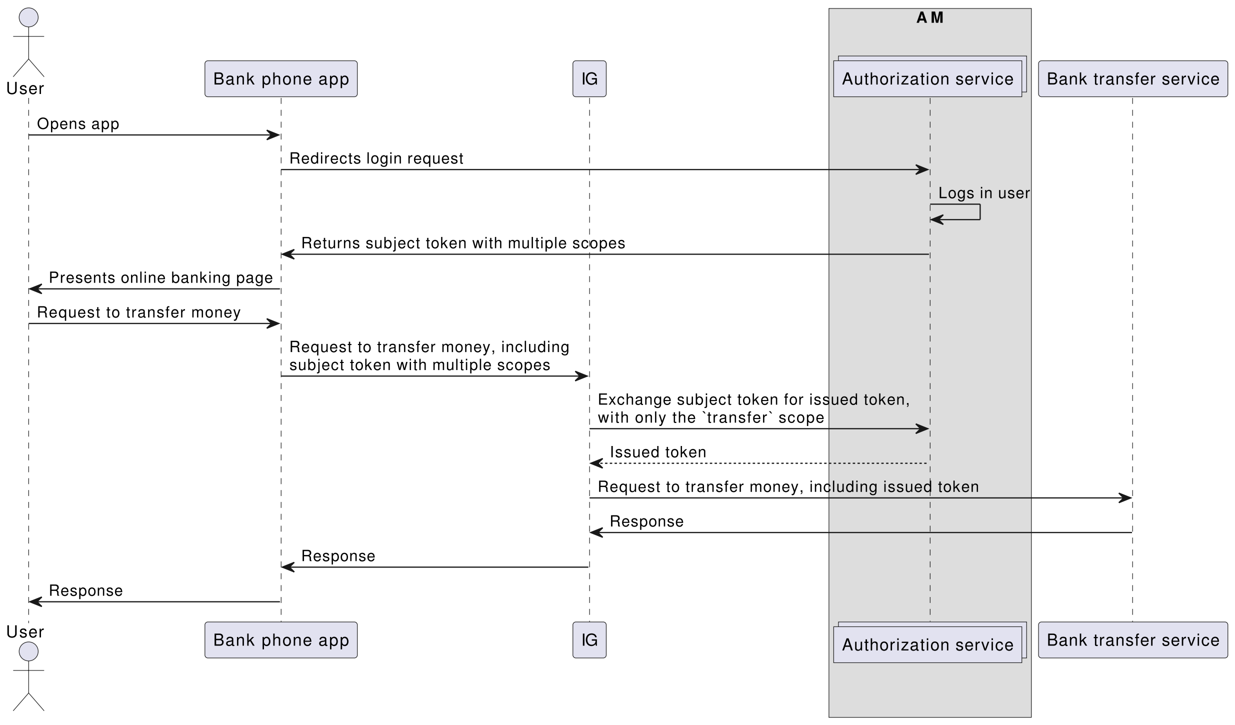
The following sections describe how to exchange an OAuth 2.0 access token for another access token, with AM as an authorization server. Other authorization providers can be used instead of AM.

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 AM's *OAuth 2.0 guide*.

The following sequence diagram shows the flow of information during token exchange between IG and AM:


AM
User
Bank phone app
IG
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

This procedure uses the *Resource Owner Password Credentials* grant type. According to information in the The OAuth 2.0 Authorization Framework, minimize use of this grant type and utilize other grant types whenever possible.

Before you start, prepare AM, IG, and the sample application as described in Example installation for this guide.

1. Set up AM:

   a. (From AM 6.5.3) 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/*?*`

b. Select **Applications** > **Agents** > **Identity Gateway**, and register an IG 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.

c. (Optional) Authenticate the agent to AM as described in Authenticate an IG agent to AM.

   IMPORTANT

   IG 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 IG:

a. Set an environment variable for the serviceConfidentialClient password:

```
$ export CLIENT_SECRET_ID='cGFzc3dvcmQ='
```

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

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

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

c. Add the following route to IG to exchange the access token:

1. Linux
2. Windows

```
$HOME/.openig/config/routes/token-exchange.json
```

```
%appdata%\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`

- IG 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&sc
ope=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/i
ntrospect' \
--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=$(curl \
--location \
--request POST 'http://ig.example.com:8080/token-
exchange' \
--header 'Content-Type: application/x-www-form-
urlencoded' \
--data "subject_token=${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/i
ntrospect' \
--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, IG routes protect resources (such as a websites or applications) from all requests on the route's condition path. Some parts of the resource, however, do not need to be protected. For example, it can be okay for unauthenticated requests to access the welcome page of a web site, or an image or favicon.

The following sections give examples of routes that do not enforce authentication for a specific request URL or URL pattern, but enforce authentication for other request URLs:

Implement not-enforced URIs with a SwitchFilter

Before you start:

- Prepare IG and the sample app as described in the Getting started
- 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:

     1. Linux
     2. Windows

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

```
C:\Temp\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
```

2. Set up AM:

   a. (From AM 6.5.3) 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/*?*`

   b. Select **Applications** > **Agents** > **Identity Gateway** and register an IG agent with the following values:

      - **Agent ID**: `ig_agent`
      - **Password**: `password`

        For AM 6.5.x and earlier versions, register an agent as described in Register an IG agent in AM 6.5 and earlier.

        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 an IG agent to AM.

      IMPORTANT

      IG 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 IG:

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

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

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

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

1. Linux
2. Windows

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

```
%appdata%\OpenIG\config\routes\static-
resources.json
```

```
{
  "name" : "sampleapp-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css')}",
  "handler": "ReverseProxyHandler"
}
```

c. Add the following route to IG:

1. Linux
2. Windows

```
$HOME/.openig/config/routes/not-enforced-
switch.json
```

```
%appdata%\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",
                "target":
"${attributes.credentials}"
              }
            },
            "request": {
              "method": "POST",
              "uri":
"http://app.example.com:8081/login",
              "form": {
                "username": [

"${attributes.credentials.username}"
                ],
                "password": [

"${attributes.credentials.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 does not 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 logged in to AM, log out 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. Log in 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.

Implement not-enforced URIs with a DispatchHandler

To use a DispatchHandler for not-enforced URIs, replace the route in Implement not-enforced URIs 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",
                        "target": "${attributes.credentials}"
                      }
                    },
                    "request": {
                      "method": "POST",
                      "uri": "http://app.example.com:8081/login",
                      "form": {
                        "username": [
                          "${attributes.credentials.username}"
                        ],
                        "password": [
                          "${attributes.credentials.password}"
                        ]
                      }
                    }
                  }
                }
              ],
              "handler": "ReverseProxyHandler"
            }
          }
        }
      ]
    }
  }
}
```

POST data preservation

The DataPreservationFilter triggers POST data preservation when an unauthenticated client posts HTML form data to a protected resource.

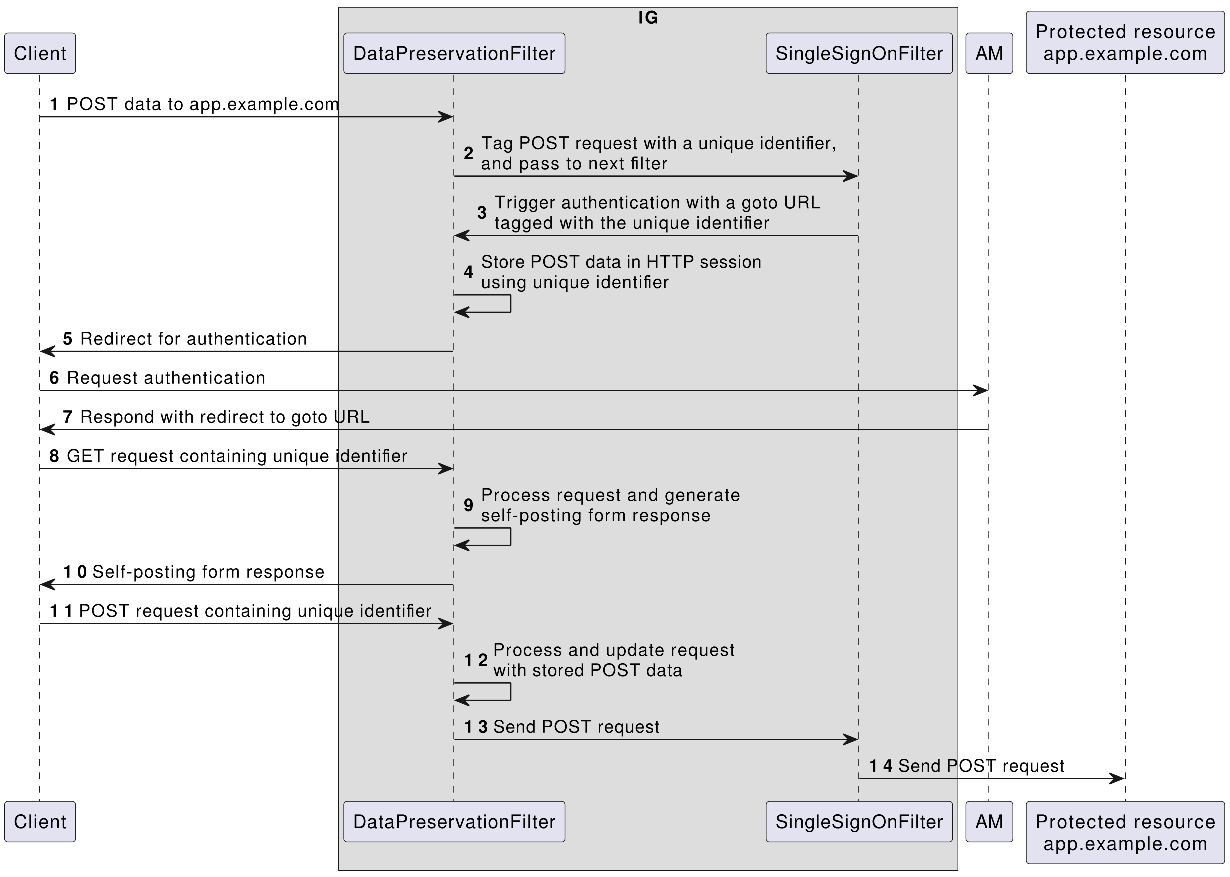
When an authentication redirect is triggered, the filter stores the data in the HTTP session, and redirects the client for authentication. After authentication, the filter generates an empty self-submitting form POST to emulate the original POST. It then replays the stored data into the request before 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 IG instances.

The following image shows a simplified data flow for POST data preservation:



IG
Client
DataPreservationFilter
SingleSignOnFilter
AM
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
14 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, IG 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

1. Set up AM and IG as described in Authentication, and test the example. This example extends that example.
2. Replace `cdsso.json` with the following route:
    1. Linux
    2. Windows

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

```
%appdata%\OpenIG\config\routes\pdp.json
```

```
{
  "name": "pdp",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path,
'^/home/cdsso')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "http://am.example.com:8088/openam",
        "realm": "/",
        "agent": {
          "username": "ig_agent_cdsso",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "sessionCache": {
          "enabled": false
        }
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "DataPreservationFilter",
          "type": "DataPreservationFilter"
        },
        {
          "name": "CrossDomainSingleSignOnFilter-1",
          "type": "CrossDomainSingleSignOnFilter",
          "config": {
            "redirectEndpoint": "/home/cdsso/redirect",
            "authCookie": {
              "path": "/home",
              "name": "ig-token-cookie"
            },
            "amService": "AmService-1",
            "verificationSecretId": "verify",
            "secretsProvider": {
              "type": "JwkSetSecretStore",
              "config": {
                "jwkUrl":
"http://am.example.com:8088/openam/oauth2/connect/jwk_uri"
              }
            },
            "logoutExpression": "${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.

WARNING

For the security of your deployment, always configure `verificationSecretId` in CrossDomainSingleSignOnFilter. When `verificationSecretId` is not configured, IG does not verify the signature of AM session tokens, increasing the risk of CDSSO token tampering.

3. Add the following route to IG:
    1. Linux
    2. Windows

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

```
%appdata%\OpenIG\config\routes\pdp.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>
```

4. Test the setup:
    a. If you are logged in to AM, log out and clear any cookies.
    b. Go to https://ig.ext.com:8443/form.

       If you see warnings that the site is not secure, respond to the warnings to access the site.

       The script in the StaticResponseHandler entity of `form.json` creates a button to demonstrate form posting.
    c. 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 are 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.

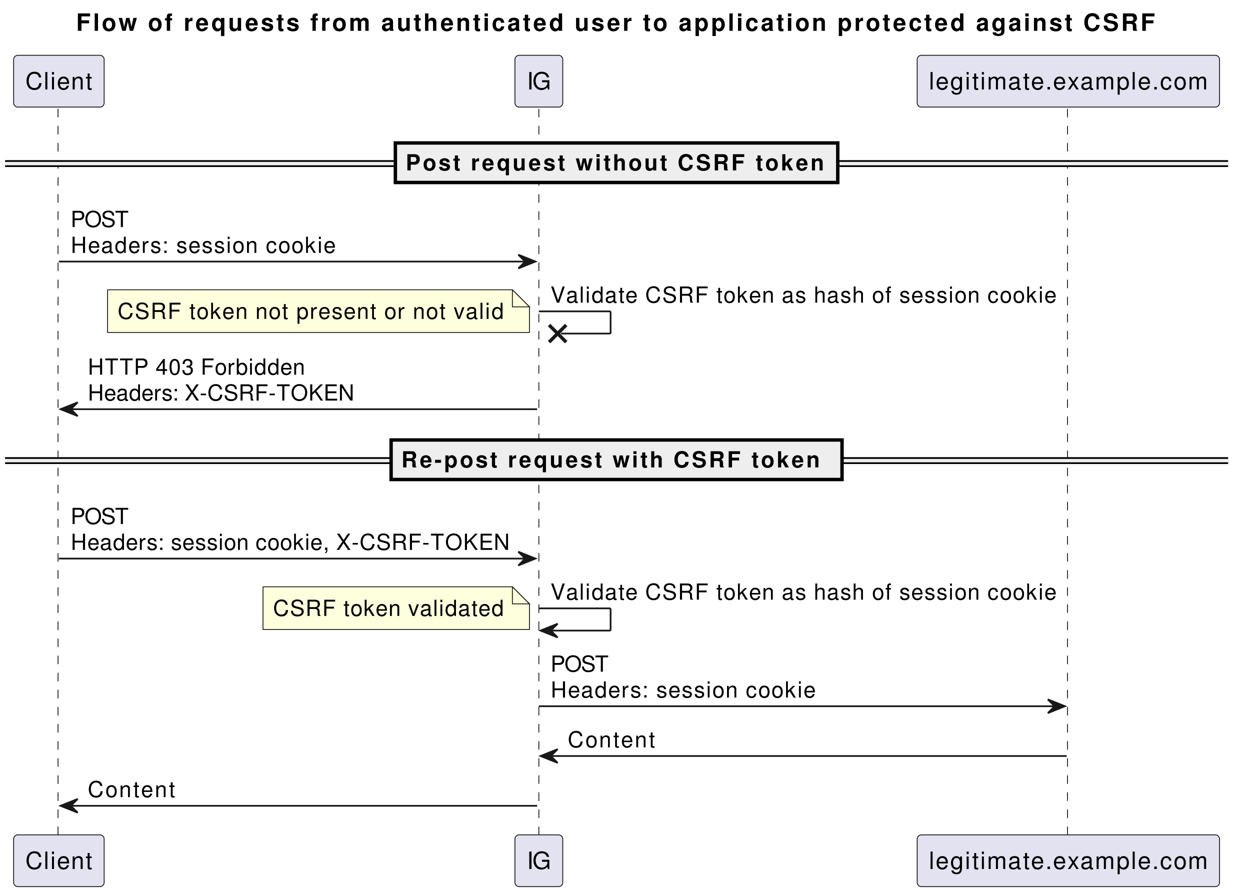
CSRF attacks interact with HTTP requests as follows:

- 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.

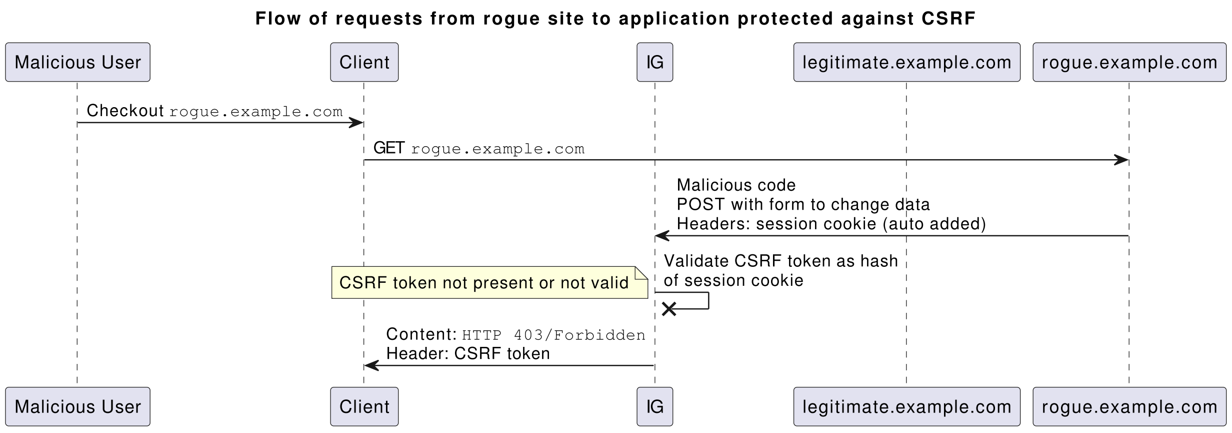
When IG processes POST, PUT, and DELETE requests, it checks a custom HTTP header in the request. If a CSRF token is not present in the header or not valid, IG rejects the request and returns a valid CSRF token in the response.

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 cannot, 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
IG
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
Client
IG
legitimate.example.com

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
IG
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
Malicious User
Client
IG
legitimate.example.com
rogue.example.com

1. Set up SSO, so that AM authenticates users to the sample app through IG:
   a. Set up AM and IG as described in Authenticate with SSO through the default authentication service.
   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 http://ig.example.com:8080/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.

   When you access this page, a hidden HTML form is automatically submitted to transfer $1000 to the rogue user, using the IG session cookie to authenticate to the bank.

   In the bank transaction history, note that $1000 has been debited.

3. Test the setup with CSRF protection:

   a. In IG, 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 is not present in the header or not valid, the filter rejects the request and provides a valid CSRF token in the header.

b. Go to http://ig.example.com:8080/bank/index, and send a bank transfer of $10 to Alice.

   Because there is no CSRF token, IG 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.

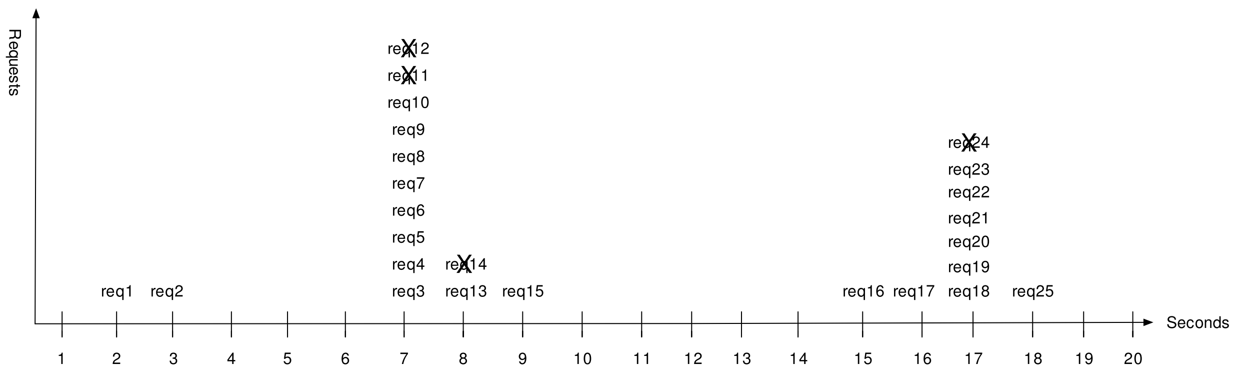
   Because there is no CSRF token, IG rejects the request and provides the CSRF token. However, because the rogue site is in a different domain to `ig.example.com` it can’t access the CSRF token.

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 IG manages requests for a throttling rate of 10 requests/10 seconds:



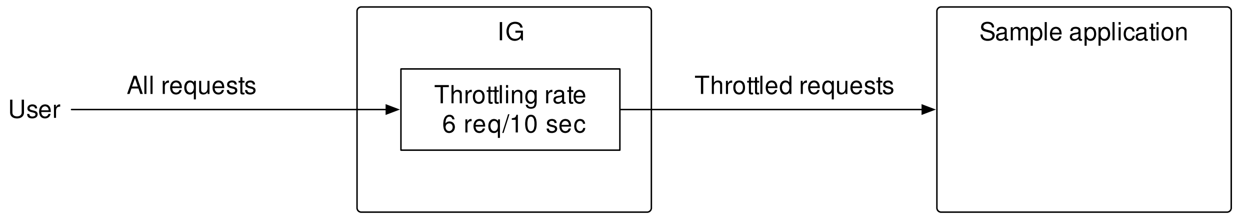
- At 7 seconds, 2 requests have previously passed when there is a burst of 9 requests. IG allows 8 requests, but disregards the 9th because the throttling rate for the 10-second throttling period has been reached.
- At 8 and 9 seconds, although 10 requests have already passed in the 10-second throttling period, IG allows 1 request each second.
- At 17 seconds, 4 requests have passed in the previous 10-second throttling period, and IG allows another burst of 6 requests.

When the throttling rate is reached, IG 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 http://ig.example.com:8080/home/throttle-scriptable HTTP/1.1
. . .

HTTP/1.1 429 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. Add the following route to IG:
    1. Linux
    2. Windows

```
$HOME/.openig/config/routes/00-throttle-simple.json
```

```
%appdata%\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 IG 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.

2. Test the setup:

   a. With IG and the sample application running, use **`curl`**, a bash script, or another tool to access the following route in a loop: http://ig.example.com:8080/home/simple-throttle.

   Accessing the route in a loop runs the request multiple times in quick succession, allowing you to test the throttling rate.

   ```
   $ curl -v http://ig.example.com:8080/home/throttle-simple/\[01-10\] \
   > /tmp/throttle-simple.txt 2>&1
   ```

   b. Search the output file for the result:

   ```
   $ grep "< HTTP/1.1" /tmp/throttle-simple.txt | sort | uniq -c

   6 < HTTP/1.1 200 OK
   4 < HTTP/1.1 429 Too Many Requests
   ```

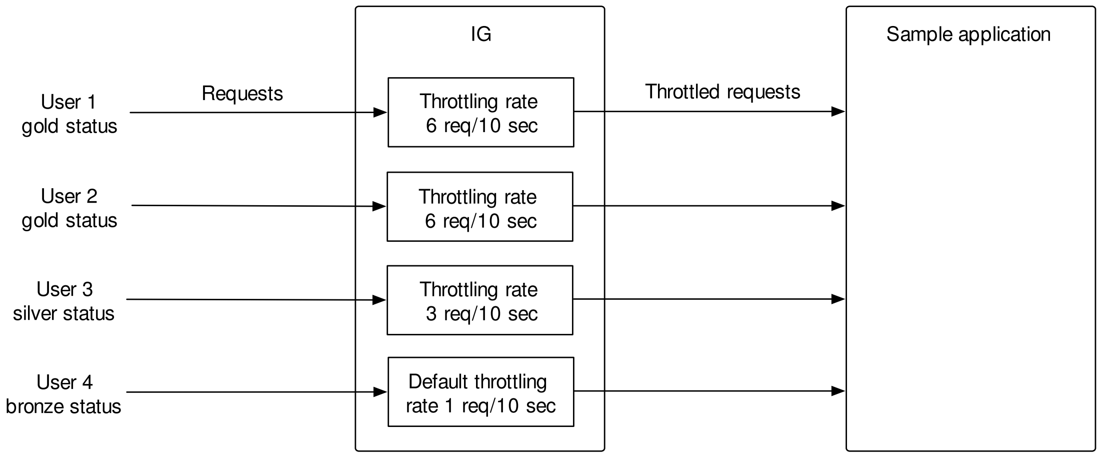
   Notice that the first six requests returned a success response, and the following four requests returned an HTTP 429 `Too Many Requests`. This result demonstrates that the throttling filter has allowed only six requests to access the application, and has blocked the other requests.

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.



1. Set up AM:

   a. Set up AM as described in Validate access tokens through the introspection endpoint.

   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

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. Set up IG:

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

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

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

   b. Add the following route to IG:

      1. Linux
      2. Windows

   ```
   $HOME/.openig/config/routes/00-throttle-mapped.json
   ```

   ```
   %appdata%\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 IG 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 the following to retrieve an access token:

   ```
   $ mytoken=$(curl -s \
   --user "client-application:password" \
   --data
   "grant_type=password&username=demo&password=Ch4ng31t&sc
   ope=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 http://ig.example.com:8080/home/throttle-
   mapped/\[01-10\] \
          --header "Authorization:Bearer ${mytoken}" \
          > /tmp/throttle-mapped.txt 2>&1
   ```

   c. Search the output file for the result:

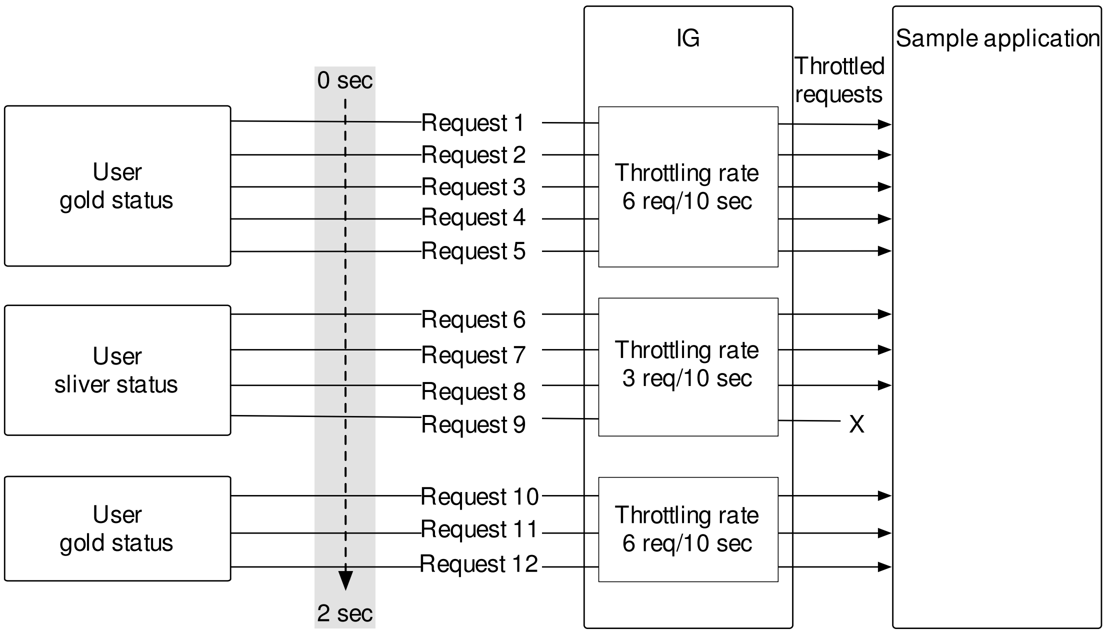
   ```
   $ grep "< HTTP/1.1" /tmp/throttle-mapped.txt | sort |
   uniq -c

   6 < HTTP/1.1 200
   4 < HTTP/1.1 429
   ```

   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.

1. Set up AM as described in Configure mapped throttling.
2. Set up IG:
    a. Set an environment variable for the IG agent password, and then restart IG:

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

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

    b. Add the following route to IG:

        1. Linux
        2. Windows

    ```
    $HOME/.openig/config/routes/00-throttle-scriptable.json
    ```

    ```
    %appdata%\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 IG 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.

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={amDemoUn}&password=
{amDemoPw}&scope=mail%20employeenumber" \
http://am.example.com:8088/openam/oauth2/access_token | jq -r
".access_token")
```

a. 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 http://ig.example.com:8080/home/throttle-
scriptable/\[01-10\] --header "Authorization:Bearer
${mytoken}" > /tmp/throttle-script.txt 2>&1
```

b. Search the output file for the result:

```
$ grep "< HTTP/1.1" /tmp/throttle-script.txt | sort | uniq
-c

6 < HTTP/1.1 200
4 < HTTP/1.1 429
```

Notice that with a `gold` status, the user can access the route 6 times in 10 seconds.

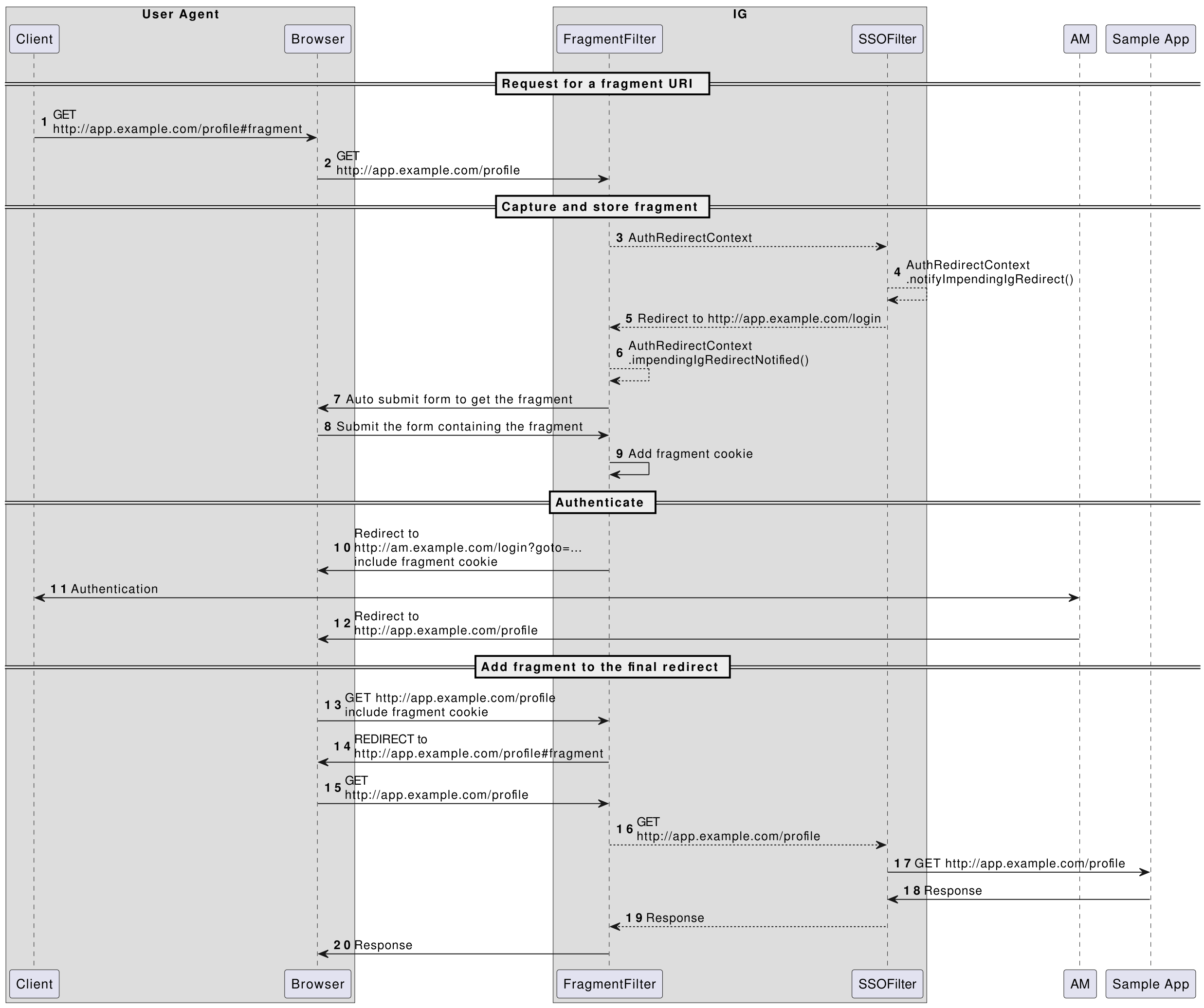
c. In AM, change the user’s email to `demo@other.com`, and then run the last two steps again to find how the access is reduced.

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 does not send the fragment. The fragment is lost during the authentication flow.

IG provides a FragmentFilter to track the fragment part of a URI when a request triggers a login redirect. The following image shows the flow of information when the FragmentFilter is included in the SSO authentication flow:


User Agent
IG
Client
Browser
FragmentFilter
SSOFilter
AM
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.

1. Set up the example in Authenticate with SSO through the default authentication service.
2. Add the following route to IG:
   1. Linux
   2. Windows

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

   ```
   %appdata%\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.

3. Test the setup:

   a. If you are logged in to AM, log out and clear any cookies.

   b. Go to http://ig.example.com:8080/home/sso#fragment.

   The SingleSignOnFilter redirects the request to AM for authentication.

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

The SingleSignOnFilter passes the request to sample app, which returns the home page. Note that the URL of the page has preserved the fragment: `http://ig.example.com:8080/home/sso?_ig=true#fragment`

d. Remove the FragmentFilter from the route and test the route again.

Note that this time the URL of the page has not preserved the fragment.

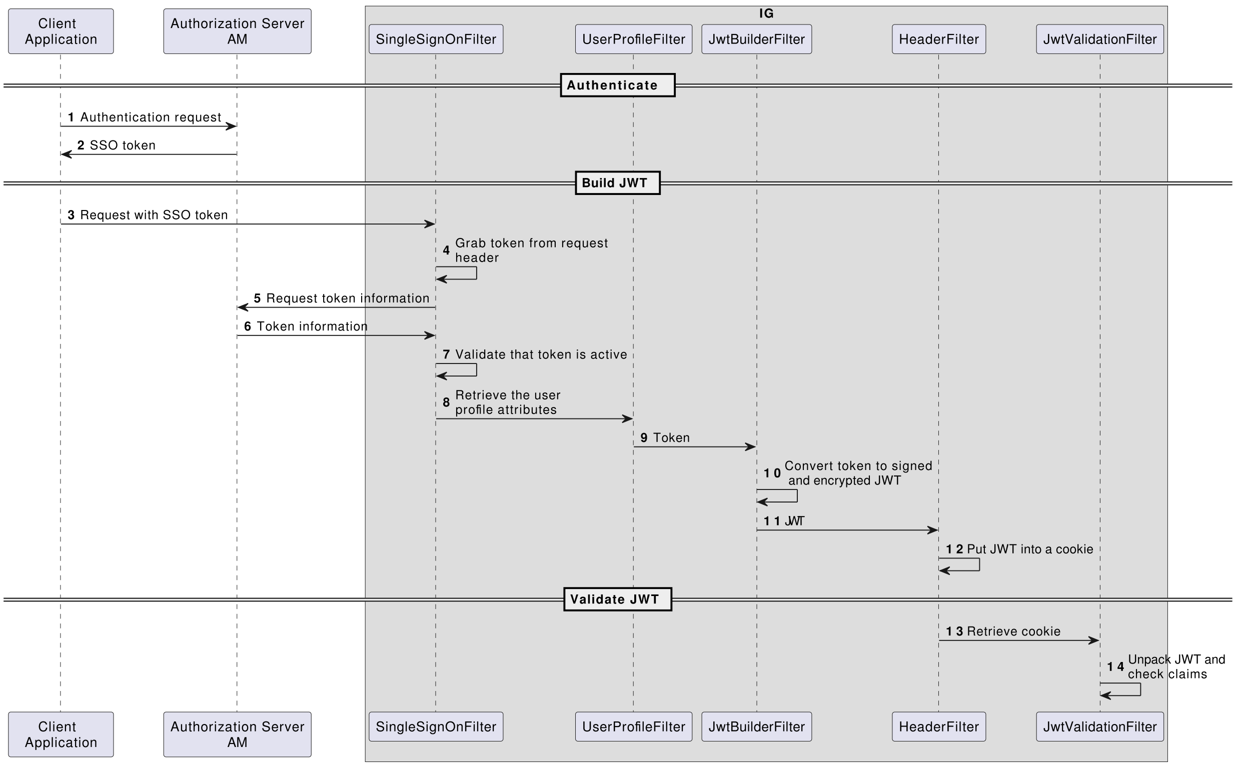
JWT validation

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:



1. Create a signed then encrypted JWT as described in Pass runtime data in a JWT signed with PEM then encrypted with a symmetric key.

2. Add the following route to IG, replacing value of the property `secretsDir` with the directory for the PEM files:

    1. Linux

    2. Windows

```
$HOME/.openig/config/routes/jwt-validate.json
```

```
%appdata%\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.descript
ion}",
                "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.

3. Test the setup:
    a. If you are logged in to AM, log out and clear any cookies.
    b. Go to http://ig.example.com:8080/jwtbuilder-sign-then-encrypt to build a JWT, and log in to AM as user `demo`, password `Ch4ng31t`. The sample app displays the signed JWT along with its header and payload.
    c. Go to http://ig.example.com:8080/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 is not validated, and an error is displayed.

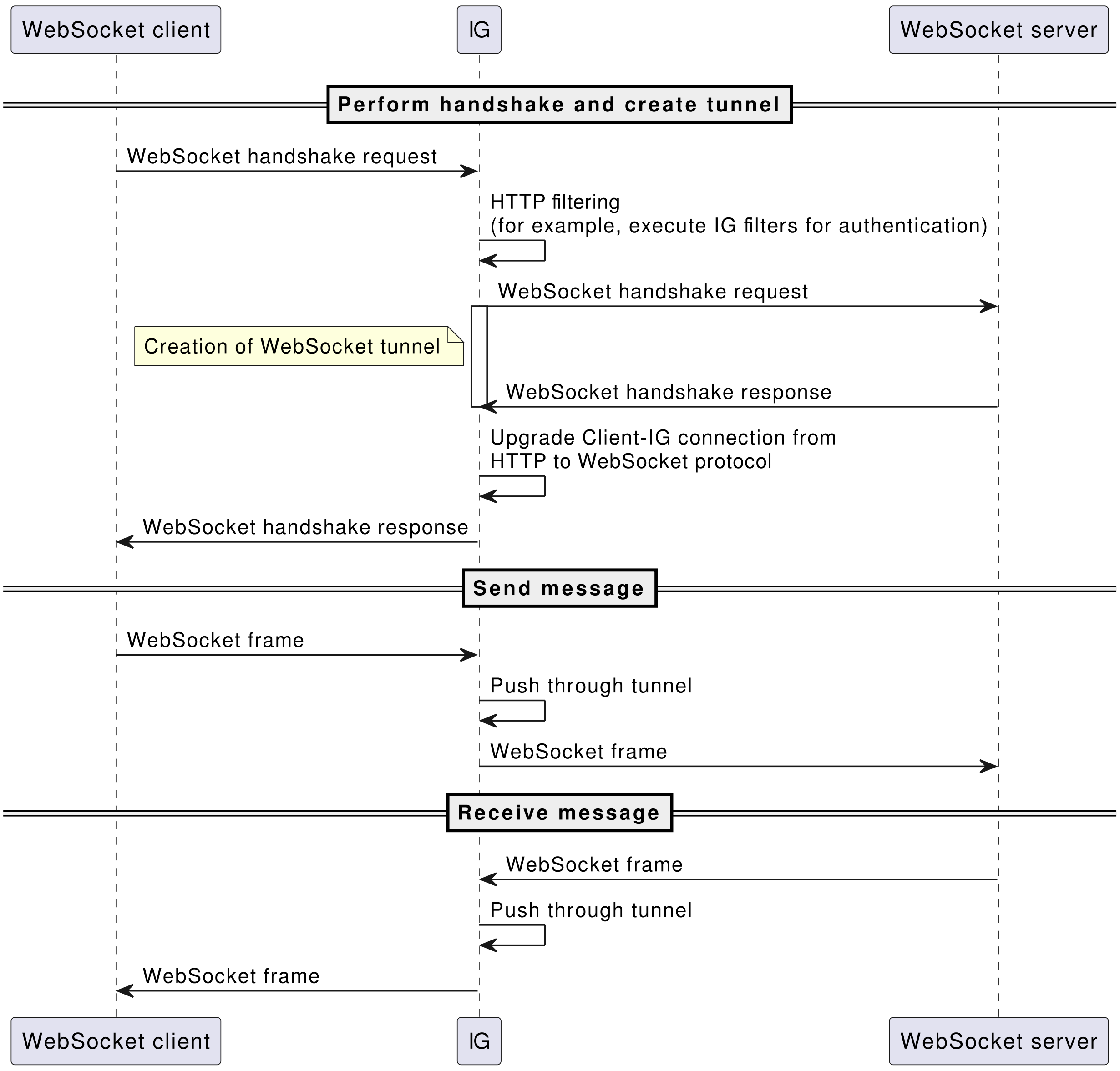
WebSocket traffic

When a user agent requests an upgrade from HTTP or HTTPS to the WebSocket protocol, IG detects the request and performs an HTTP handshake request between the user agent and the protected application.

If the handshake is successful, IG upgrades the connection and provides a dedicated tunnel to route WebSocket traffic between the user agent and the protected application. IG does not intercept messages to or from the WebSocket server.

The tunnel remains open until it is closed by the user agent or protected application. When the user agent closes the tunnel, the connection between IG and the protected application is automatically closed.

The following sequence diagram shows the flow of information when IG proxies WebSocket traffic:


WebSocket client
IG
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-IG 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
IG
WebSocket server

To configure IG to proxy WebSocket traffic, configure the `websocket` property of ReverseProxyHandler. By default, IG does not proxy WebSocket traffic.

Proxy WebSocket traffic

1. Set up AM:

   a. (From AM 6.5.3) 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/*?*`

   b. Select **Applications** > **Agents** > **Identity Gateway** and register an IG agent with the following values:

      - **Agent ID**: `ig_agent`
      - **Password**: `password`

For AM 6.5.x and earlier versions, register an agent as described in Register an IG agent in AM 6.5 and earlier.

> 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 an IG agent to AM.

> IMPORTANT
>
> IG 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 IG:

a. Add the following route to IG, to serve .css and other static resources for the sample application:

1. Linux
2. Windows

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

```
%appdata%\OpenIG\config\routes\static-resources.json
```

```
{
  "name" : "sampleapp-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css')}",
  "handler": "ReverseProxyHandler"
}
```

b. Add the following route to IG:

1. Linux
2. Windows

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

```
%appdata%\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
        }
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SingleSignOnFilter-1",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
```

```
      }
    }
```

For information about how to set up the route in Studio, refer to Proxy for WebSocket traffic in Structured Editor.

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 IG to proxy WebSocket traffic, and, after IG upgrades the HTTP connection to the WebSocket protocol, passes the messages to the WebSocket server.

  1. Test the setup:

c. If you are logged in to AM, log out and clear any cookies.

d. Go to http://ig.example.com:8080/websocket. The SingleSignOnFilter redirects the request to AM for authentication.

e. Log in 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`.

f. Enter text on the WebSocket echo screen, and note that the text is echoed back.

Vert.x-specific configuration for WebSocket connections

Configure Vert.x-specific configuration for WebSocket connections, where IG does not provide its own first-class configuration. Vert.x options are described in HttpClientOptions.

The following example configures Vert.x options for Websocket connections:

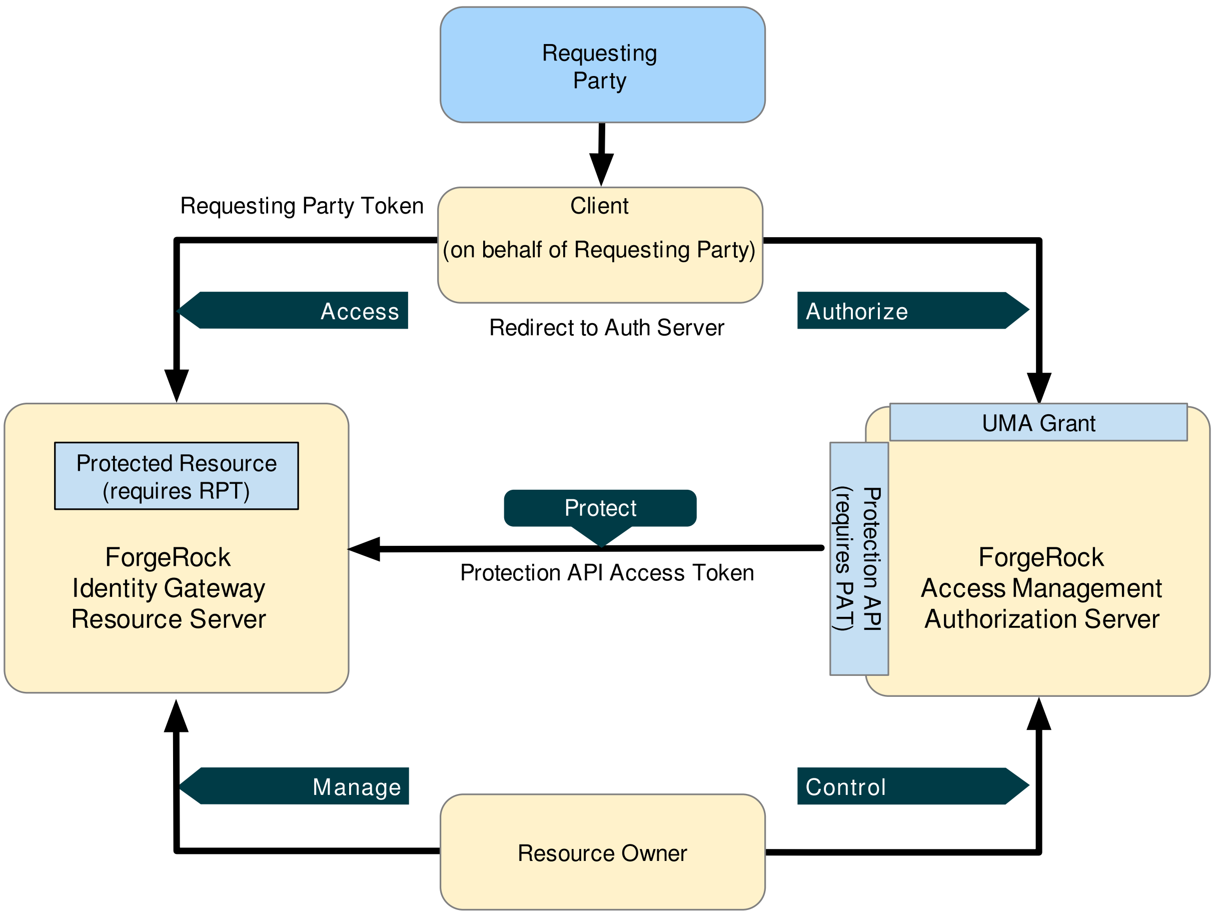
```
{
  "type": "ReverseProxyHandler",
  "config": {
    "websocket": {
      "enabled": true,
      "vertx": {
        "maxWebSocketFrameSize": 200000000,
        "maxWebSocketMessageSize": 200000000,
        "tryUsePerMessageWebSocketCompression": true
      }
    }
  }
}
```

UMA support

IG includes support for User-Managed Access (UMA) 2.0 Grant for OAuth 2.0 Authorization specifications.

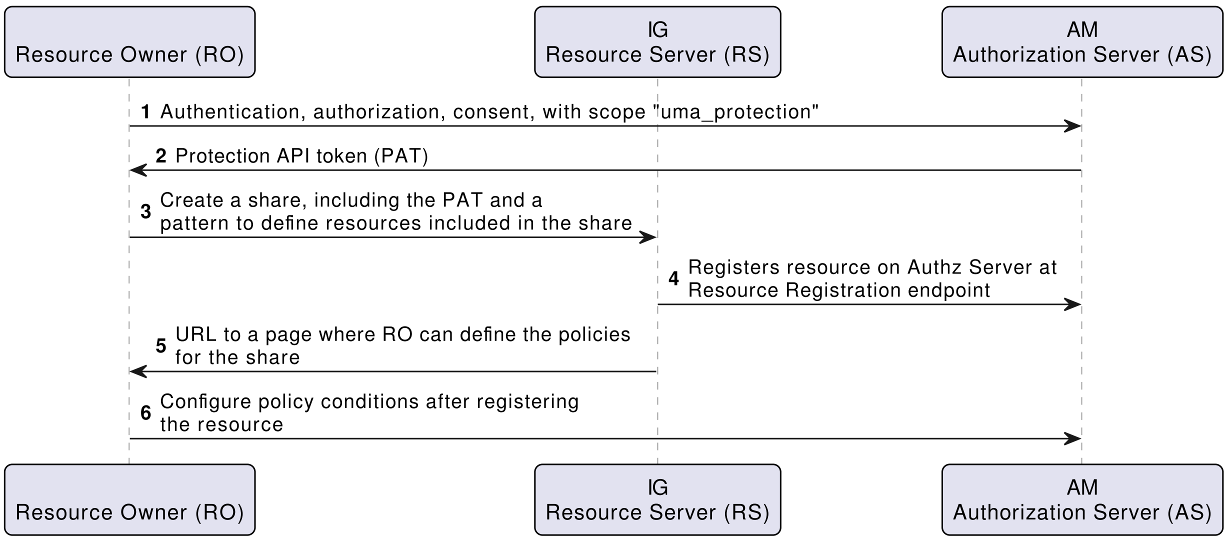
About IG as an UMA resource server

The following figure shows an UMA environment, with IG protecting a resource, and AM acting as an authorization server. For information about UMA, refer to AM's User-Managed Access (UMA) 2.0 guide.

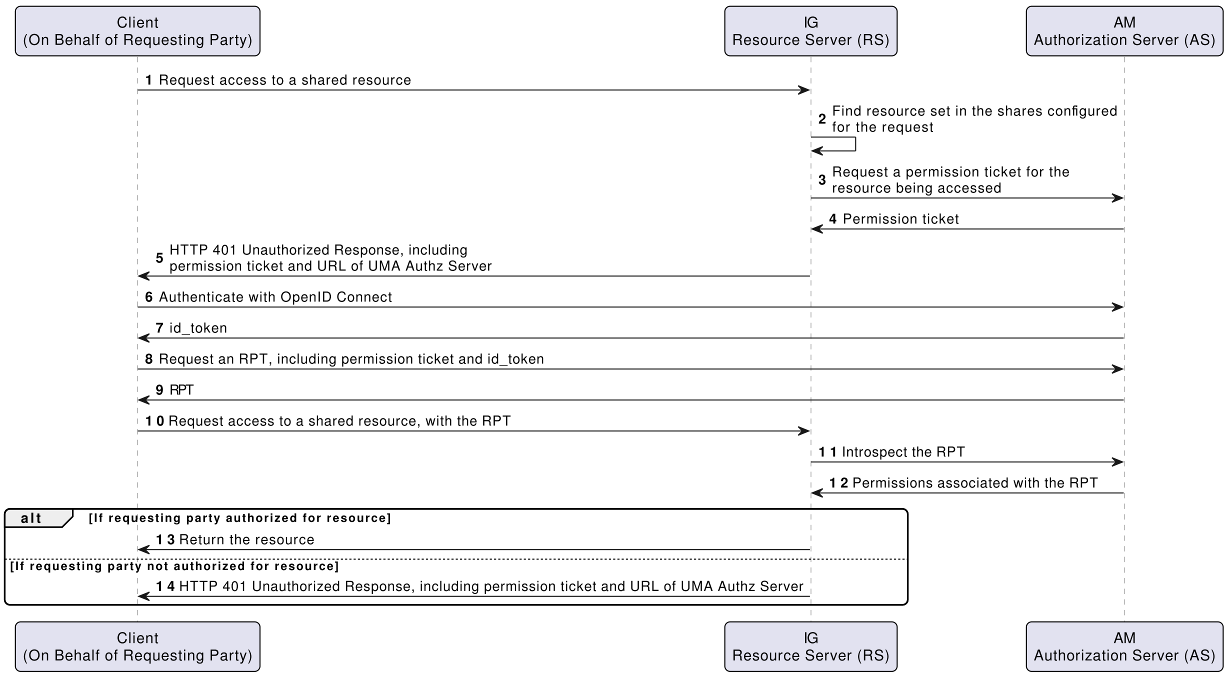


Requesting Party
Requesting Party Token
Client
(on behalf of Requesting Party)
Access
Redirect to Auth Server
Authorize
Protected Resource (requires RPT)
ForgeRock Identity Gateway Resource Server
Protect
Protection API Access Token
UMA Grant
Protection API (requires PAT)
ForgeRock Access Management 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)
IG Resource Server (RS)
AM 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

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)
IG Resource Server (RS)
AM 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

For information about CORS support, refer to Configure CORS support in AM's *Security guide*. This procedure describes how to modify the AM configuration to allow cross-site access.

Limitations of IG as an UMA resource server

When using IG as an UMA resource server, note the following points:

- IG depends on the resource owner for the PAT.

  When a PAT expires, no refresh token is available to IG. 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.

  IG has no mechanism for persisting the data across restarts. When IG 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 IG error conditions.
- By default, the REST API to manage share objects exposed by IG is protected only by CORS.
- When matching protected resource paths with share patterns, IG 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`, IG 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 are not 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=*`, do not 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. According to information in the The OAuth 2.0 Authorization Framework, minimize use of this grant type and utilize other grant types whenever possible.

Before you start, prepare AM, IG, and the sample application as described in Example installation for this guide.

If you use different settings for the sample application, refer to 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. (From AM 6.5) 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. (From AM 6.5) 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`
    g. Select **Identities**, and add an identity for a requesting party, with the following values:
        - **ID** : `bob`
        - **Password** : `UMAexamp1e`

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 use 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/CorsServi
ce \
  --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
/CorsService
```

2. Set up IG as an UMA resource server:

a. Add the following route to IG, to serve .css and other static resources for the sample application:

   1. Linux

   2. Windows

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

```
%appdata%\OpenIG\config\routes\static-resources.json
```

```
{
  "name" : "sampleapp-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css')}",
  "handler": "ReverseProxyHandler"
}
```

b. Add the following `admin.json` configuration to IG:

```
{
  "prefix": "openig",
  "connectors": [
    { "port" : 8080 }
  ],
  "heap": [
    {
      "name": "ClientHandler",
      "type": "ClientHandler"
    },
    {
      "name": "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 is overridden by the CorsFilter, which allows requests from the origin `http://app.example.com:8081`.

c. Add the following route to IG:

1. Linux

2. Windows

```
$HOME/.openig/config/routes/00-uma.json
```

```
%appdata%\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.

d. Restart IG to reload the configuration.

3. Test the setup:

a. If you are logged in to AM, log out and clear any cookies.

b. Go to http://app.example.com:8081/uma/.

c. 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 IG.
- 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&userna
me=alice&password=UMAexamp1e&client_id=OpenIG&cl
ient_secret=password' \
http://am.example.com:8088/openam/oauth2/access_
token
```

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 IG configuration matches that shown when you are running the test on http://app.example.com:8081/uma/.

d. 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 Using the sample application to temporary folder:

```
$ mkdir /tmp/uma
$ cd /tmp/uma
$ jar -xvf /path/to/IG-sample-application-2023.6.0.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/IG-sample-application-2023.6.0.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 Understanding IG APIs with API descriptors.

Extensibility

To achieve complex server interactions or intensive data transformations that you can't currently achieve with scripts or existing handlers, filters, or expressions, extend IG through scripting and customization. The following sections describe how to extend IG:

Extend IG through scripts

The following sections describe how to extend IG through scripts:

About scripts

IMPORTANT

IMPORTANT

When you are writing scripts or Java extensions, never use a `Promise` blocking method, such as `get()`, `getOrThrow()`, or `getOrThrowUninterruptibly()`, to obtain the response.

A promise represents the result of an asynchronous operation. Therefore, using a blocking method to wait for the result can cause deadlocks and/or race issues.

IG supports the Groovy dynamic scripting language through the use the scriptable objects. For information about scriptable object types, their configuration, and properties, refer to Scripts.

Scriptable objects are configured by the script's Internet media type, and either a source script included in the JSON configuration, or a file script that IG reads from a file. The configuration can optionally supply arguments to the script.

IG provides global variables to scripts at runtime, and provides access to Groovy's built-in functionality. Scripts can access the request and the context, store variables across executions, write messages to logs, make requests to a web service or to an LDAP directory service, and access responses returned in promise callback methods.

Before trying the scripts in this chapter, install and configure IG as described in the Getting started.

When developing and debugging your scripts, consider configuring a capture decorator to log requests, responses, and context data in JSON form. You can then turn off capturing when you move to production. For information, refer to CaptureDecorator.

Use a reference file script

The following example defines a ScriptableFilter written in Groovy, and stored in the following file:

1. Linux
2. Windows

```
$HOME/.openig/scripts/groovy/SimpleFormLogin.groovy
```

```
%appdata%\OpenIG\scripts\groovy\SimpleFormLogin.groovy
```

```
{
  "name": "SimpleFormLogin",
  "type": "ScriptableFilter",
  "config": {
    "type": "application/x-groovy",
    "file": "SimpleFormLogin.groovy"
  }
}
```

Relative paths in the file field depend on how IG is installed. If IG is installed in an application server, then paths for Groovy scripts are relative to `$HOME/.openig/scripts/groovy` (or `%appdata%\OpenIG\scripts\groovy`).

The base location `$HOME/.openig/scripts/groovy` (or `%appdata%\OpenIG\scripts\groovy`) is on the classpath when the scripts are executed. If some Groovy scripts are not in the default package, but instead have their own package names, they belong in the directory corresponding to their package name. For example, a script in package `com.example.groovy` belongs in `$HOME/.openig/scripts/groovy/com/example/groovy/` (or `%appdata%\OpenIG\scripts\groovy\com\example\groovy\`).

Scripts in Studio

You can use Studio to configure a ScriptableFilter or scriptableThrottlingPolicy, or use scripts to configure scopes in OAuth2ResourceServerFilter.

During configuration, you can enter the script directly into the object, or you can use a stored reference script. Note the following points about creating and using reference scripts:

- When you enter a script directly into an object, the script is added to the list of reference scripts.
- You can use a reference script in multiple objects in a route, but if you edit a reference script, all objects that use it are updated with the change.
- If you delete an object that uses a script, or remove the object from the chain, the script that it references remains in the list of scripts.
- If a reference script is used in an object, you can't rename or delete the script.

For an example of creating a ScriptableThrottlingPolicy in Studio, refer to Configure Scriptable Throttling. For information about using Studio, refer to Adding Configuration to a Route.

Script dispatch

To route requests when the conditions are complicated, use a `ScriptableHandler` instead of a `DispatchHandler` as described in DispatchHandler.

1. Add the following script to IG:
    1. Linux

2. Windows

```
$HOME/.openig/scripts/groovy/DispatchHandler.groovy
```

```
%appdata%\OpenIG\scripts\groovy\DispatchHandler.groovy
```

```
/*
 * This simplistic dispatcher matches the path part of the
HTTP request.
 * If the path is /mylogin, it checks Username and
Password headers,
 * accepting bjensen:H1falutin, and returning HTTP 403
Forbidden to others.
 * Otherwise it returns HTTP 401 Unauthorized.
 */

// Rather than returning a Promise of a Response from an
external source,
// this script returns the response itself.
response = new Response(Status.OK);

switch (request.uri.path) {

    case "/mylogin":

        if (request.headers.Username.values[0] ==
"bjensen" &&
                request.headers.Password.values[0] ==
"H1falutin") {

            response.status = Status.OK
            response.entity = "<html><p>Welcome back,
Babs!</p></html>"

        } else {

            response.status = Status.FORBIDDEN
            response.entity = "<html><p>Authorization
required</p></html>"

        }

        break
    default:

        response.status = Status.UNAUTHORIZED
        response.entity = "<html><p>Please <a
href='./mylogin'>log in</a>.</p></html>"

        break

}

// Return the locally created response, no need to wrap it
into a Promise
return response
```

2. Add the following route to IG, to set up headers required by the script when the user logs in:
   1. Linux
   2. Windows

```
$HOME/.openig/config/routes/98-dispatch.json
```

```
%appdata%\OpenIG\config\routes\98-dispatch.json
```

```
{
  "heap": [
    {
      "name": "DispatchHandler",
      "type": "DispatchHandler",
      "config": {
        "bindings": [{
          "condition": "${find(request.uri.path,
'/mylogin')}",
          "handler": {
            "type": "Chain",
            "config": {
              "filters": [
                {
                  "type": "HeaderFilter",
                  "config": {
                    "messageType": "REQUEST",
                    "add": {
                      "Username": [
                        "bjensen"
                      ],
                      "Password": [
                        "H1falutin"
                      ]
                    }
                  }
                }
              ],
              "handler": "Dispatcher"
            }
          }
        },
          {
            "handler": "Dispatcher",
            "condition": "${find(request.uri.path,
'/dispatch')}"
          }
        ]
      }
    },
    {
      "name": "Dispatcher",
      "type": "ScriptableHandler",
      "config": {
        "type": "application/x-groovy",
        "file": "DispatchHandler.groovy"
      }
    }
  ],
  "handler": "DispatchHandler",
  "condition": "${find(request.uri.path, '^/dispatch') or
find(request.uri.path, '^/mylogin')}"
}
```

3. Go to http://ig.example.com:8080/dispatch, and click `log in`.

   The HeaderFilter sets `Username` and `Password` headers in the request, and passes the request to the script. The script responds, `Welcome back, Babs!`

Script HTTP basic access authentication

HTTP basic access authentication is a simple challenge and response mechanism, where a server requests credentials from a client, and the client passes them to the server in an `Authorization` header. The credentials are base-64 encoded. To protect them, use SSL encryption for the connections between the server and client. For more information, refer to RFC 2617.

1. Add the following script to IG, to add an `Authorization` header based on a username and password combination:
   1. Linux
   2. Windows

   ```
   $HOME/.openig/scripts/groovy/BasicAuthFilter.groovy
   ```

   ```
   %appdata%\OpenIG\scripts\groovy\BasicAuthFilter.groovy
   ```

   ```
   /*
    * Perform basic authentication with the user name and
   password
    * that are supplied using a configuration like the
   following:
    *
    * {
    *     "name": "BasicAuth",
    *     "type": "ScriptableFilter",
    *     "config": {
    *         "type": "application/x-groovy",
    *         "file": "BasicAuthFilter.groovy",
    *         "args": {
    *             "username": "bjensen",
    *             "password": "H1falutin"
    *             }
    *         }
    * }
    */

   def userPass = username + ":" + password
   def base64UserPass = userPass.getBytes().encodeBase64()
   request.headers.add("Authorization", "Basic
   ${base64UserPass}" as String)

   // Credentials are only base64-encoded, not encrypted: Set
   scheme to HTTPS.

   /*
    * When connecting over HTTPS, by default the client tries
   ```

```
to trust the server.
 * If the server has no certificate
 * or has a self-signed certificate unknown to the client,
 * then the most likely result is an
SSLPeerUnverifiedException.
 *
 * To avoid an SSLPeerUnverifiedException,
 * set up HTTPS correctly on the server.
 * Either use a server certificate signed by a well-known
CA,
 * or set up the gateway to trust the server certificate.
 */
request.uri.scheme = "https"

// Calls the next Handler and returns a Promise of the
Response.
// The Response can be handled with asynchronous Promise
callbacks.
next.handle(context, request)
```

2. Add the following route to IG, to set up headers required by the script when the user logs in:

   1. Linux

   2. Windows

```
$HOME/.openig/config/routes/09-basic.json
```

```
%appdata%\OpenIG\config\routes\09-basic.json
```

```
{
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "ScriptableFilter",
          "config": {
            "type": "application/x-groovy",
            "file": "BasicAuthFilter.groovy",
            "args": {
              "username": "bjensen",
              "password": "H1falutin"
            }
          },
          "capture": "filtered_request"
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/plain; charset=UTF-8"
]
          },
          "entity": "Hello bjensen!"
        }
      }
    }
  },
  "condition": "${find(request.uri.path, '^/basic')}"
}
```

When the request path matches `/basic`, the route calls the Chain, which runs the ScriptableFilter. The capture setting captures the request as updated by the ScriptableFilter. Finally, IG returns a static page.

3. Go to http://ig.example.com:8080/basic.

The captured request in the console log shows that the scheme is now HTTPS, and that the `Authorization` header is set for HTTP Basic:

```
GET https://app.example.com:8081/basic HTTP/1.1
 ...
 Authorization: Basic Ymp...aW4=
```

Script authentication to LDAP-enabled servers

Many organizations use an LDAP directory service, such as ForgeRock Directory Services DS), to store user profiles and authentication credentials. This section describes how to authenticate to DS by using a script and a ScriptableFilter.

DS is secure by default, so connections between IG and DS must be configured for TLS. For convenience, this example uses a TrustAllManager to blindly accept any certificate presented by DS. In a production environment, use a TrustManager that is configured to accept *only* the appropriate certificates.

If the LDAP connection in your deployment is not secured with TLS, you can remove SSL options from the example script, and remove the TrustAllManager from the example route.

For more information about attributes and types for interacting with LDAP, see AttributeParser in DS's Javadoc. The ConnectionFactory heartbeat is enabled by default. For information about how to disable it, refer to LdapConnectionFactory in DS's Javadoc.

1. Install an LDAP directory server, such as ForgeRock Directory Services, and then generate or import some sample users who can authenticate over LDAP. For information about setting up DS and importing sample data, refer to Install DS for evaluation in Directory Services's *Installation guide*.
2. Add the following script to IG:
    1. Linux
    2. Windows

```
$HOME/.openig/scripts/groovy/LdapsAuthFilter.groovy
```

```
%appdata%\OpenIG\scripts\groovy\LdapsAuthFilter.groovy
```

```
import org.forgerock.opendj.ldap.*
import org.forgerock.opendj.security.SslOptions;

/* Perform LDAP authentication based on user credentials
from a form,
 * connecting to an LDAPS enabled server.
 *
 * If LDAP authentication succeeds, then return a promise
to handle the response.
 * If there is a failure, produce an error response and
return it.
 */

username = request.queryParams?.username[0]
password = request.queryParams?.password[0]

// Update port number to match the LDAPS port of your
directory service.
host = ldapHost ?: "localhost"
port = ldapPort ?: 1636

// Include options for SSL.
// In this example, the keyManager is not set (no mTLS
enabled), and both
// the trustManager and the LDAP secure protocol are
specified from the
// script arguments (see 'trustManager' and 'protocols'
arguments).
// In a development environment (when there is no TLS),
the SslOptions can be removed completely.
ldapOptions = ldap.defaultOptions(context)
SslOptions sslOptions = SslOptions.newSslOptions(null,
trustManager)

.enabledProtocols(protocols);
ldapOptions =
ldapOptions.set(CommonLdapOptions.SSL_OPTIONS,
sslOptions);

// Include SSL options in the LDAP connection
client = ldap.connect(host, port as Integer, ldapOptions)
try {

    // Assume the username is an exact match of either
    // the user ID, the email address, or the user's full
name.
    filter = "(|(uid=%s)(mail=%s)(cn=%s))"

    user = client.searchSingleEntry(
            "ou=people,dc=example,dc=com",
            ldap.scope.sub,
            ldap.filter(filter, username, username,
username))

    client.bind(user.name as String,
password?.toCharArray())

    // Authentication succeeded.

    // Set a header (or whatever else you want to do
here).
    request.headers.add("Ldap-User-Dn",
user.name.toString())

    // Most LDAP attributes are multi-valued.
    // When you read multi-valued attributes, use the
parse() method,
    // with an AttributeParser method
    // that specifies the type of object to return.
    attributes.cn = user.cn?.parse().asSetOfString()

    // When you write attribute values, set them directly.
    user.description = "New description set by my script"

    // Here is how you might read a single value of a
multi-valued attribute:
    attributes.description =
user.description?.parse().asString()

    // Call the next handler. This returns when the
request has been handled.
    return next.handle(context, request)

} catch (AuthenticationException e) {
    // LDAP authentication failed, so fail the response
with
    // HTTP status code 403 Forbidden.
    response = new Response(Status.FORBIDDEN)
    response.headers['Content-Type'] = "text/html;
charset=utf-8"
    response.entity = "<html><p>Authentication failed: " +
e.message + "</p></html>"

} catch (Exception e) {
    // Something other than authentication failed on the
server side,
    // so fail the response with HTTP 500 Internal Server
Error.
    response = new Response(Status.INTERNAL_SERVER_ERROR)
    response.headers['Content-Type'] = "text/html;
charset=utf-8"
    response.entity = "<html><p>Server error: " +
e.message + "</p></html>"

} finally {
    client.close()
}

// Return the locally created response, no need to wrap it
into a Promise
return response
```

Information about the script is given in the script comments. If necessary, adjust the script to match your DS installation.

3. Add the following route to IG:

   1. Linux

   2. Windows

```
$HOME/.openig/config/routes/10-ldap.json
```

```
%appdata%\OpenIG\config\routes\10-ldap.json
```

```
{
  "heap": [
    {
      "name": "DsTrustManager",
      "type": "TrustAllManager"
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "ScriptableFilter",
          "config": {
            "args": {
              "ldapHost": "localhost",
              "ldapPort": 1636,
              "protocols": "TLSv1.3",
              "trustManager": "${heap['DsTrustManager']}"
            },
            "type": "application/x-groovy",
            "file": "LdapsAuthFilter.groovy"
          }
        }
      ],
      "handler": {
        "type": "ScriptableHandler",
        "config": {
          "type": "application/x-groovy",
          "source": [
            "dn = request.headers['Ldap-User-
Dn'].values[0]",
            "entity = '<html><body><p>Ldap-User-Dn: ' + dn
+ '</p></body></html>'",
            "",
            "response = new Response(Status.OK)",
            "response.entity = entity",
            "return response"
          ]
        }
      }
    }
  },
  "condition": "${find(request.uri.path, '^/ldap')}"
}
```

Notice the following features of the route:

- The route matches requests to `/ldap`.
- The ScriptableFilter calls `LdapsAuthFilter.groovy` to authenticate the user over a secure LDAP connection, using the username and password provided in the request.
- The script uses TrustAllManager to blindly accept any certificate presented by DS.
- The script receives a connection to the DS server, using TLS options. Using the credentials in the request, the script tries to perform an LDAP bind operation. If the bind succeeds (the credentials are accepted by the LDAP server), the request continues to the ScriptableHandler. Otherwise, the request stops with an error.
- The ScriptableHandler returns the user DN.

4. Go to http://ig.example.com:8080/ldap?username=abarnes&password=chevron to specify credentials for the sample user `abarnes`.

   The script returns the user DN:

   ```
   Ldap-User-Dn: uid=abarnes,ou=People,dc=example,dc=com
   ```

Script SQL queries

This example builds on Password replay from a database to use scripts to look up credentials in a database, set the credentials in headers, and set the scheme in HTTPS to protect the request.

1. Set up and test the example in Password replay from a database.

2. Add the following script to IG, to look up user credentials in the database, by email address, and set the credentials in the request headers for the next handler:

   1. Linux
   2. Windows

```
$HOME/.openig/scripts/groovy/SqlAccessFilter.groovy
```

```
%appdata%\OpenIG\scripts\groovy\SqlAccessFilter.groovy
```

```
/*
 * Look up user credentials in a relational database
 * based on the user's email address provided in the
request form data,
 * and set the credentials in the request headers for the
next handler.
 */

def client = new SqlClient(dataSource)
def credentials =
client.getCredentials(request.queryParams?.mail[0])
request.headers.add("Username", credentials.Username)
request.headers.add("Password", credentials.Password)

// The credentials are not protected in the headers, so
use HTTPS.
request.uri.scheme = "https"

// Calls the next Handler and returns a Promise of the
Response.
// The Response can be handled with asynchronous Promise
callbacks.
next.handle(context, request)
```

3. Add the following script to IG to access the database, and get credentials:

   1. Linux
   2. Windows

```
$HOME/.openig/scripts/groovy/SqlClient.groovy
```

```
%appdata%\OpenIG\scripts\groovy\SqlClient.groovy
```

```
import groovy.sql.Sql

import javax.sql.DataSource

/**
 * Access a database with a well-known structure,
 * in particular to get credentials given an email
address.
 */
class SqlClient {

    // DataSource supplied as constructor parameter.
    def sql

    SqlClient(DataSource dataSource) {
        if (dataSource == null) {
            throw new IllegalArgumentException("DataSource
is null")
        }
        this.sql = new Sql(dataSource)
    }

    // The expected table is laid out like the following.

    // Table USERS
    // ----------------------------------------
    // | USERNAME  | PASSWORD |   EMAIL   |...|
    // ----------------------------------------
    // | <username>| <passwd> | <mail@...>|...|
    // ----------------------------------------

    String tableName = "USERS"
    String usernameColumn = "USERNAME"
    String passwordColumn = "PASSWORD"
    String mailColumn = "EMAIL"

    /**
     * Get the Username and Password given an email
address.
     *
     * @param mail Email address used to look up the
credentials
```

```
     * @return Username and Password from the database
     */
    def getCredentials(mail) {
        def credentials = [:]
        def query = "SELECT " + usernameColumn + ", " +
passwordColumn +
                " FROM " + tableName + " WHERE " +
mailColumn + "='$mail';"

        sql.eachRow(query) {
            credentials.put("Username",
it."$usernameColumn")
            credentials.put("Password",
it."$passwordColumn")
        }
        return credentials
    }
}
```

4. Add the following route to IG to set up headers required by the scripts when the user logs in:

   1. Linux

   2. Windows

```
$HOME/.openig/config/routes/11-db.json
```

```
%appdata%\OpenIG\config\routes\11-db.json
```

```
{
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "JdbcDataSource-1",
      "type": "JdbcDataSource",
      "config": {
        "driverClassName": "org.h2.Driver",
        "jdbcUrl": "jdbc:h2:tcp://localhost/~/test",
        "username": "sa",
        "passwordSecretId": "database.password",
        "secretsProvider": "SystemAndEnvSecretStore-1"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "ScriptableFilter",
          "config": {
            "args": {
              "dataSource": "${heap['JdbcDataSource-1']}"
            },
            "type": "application/x-groovy",
            "file": "SqlAccessFilter.groovy"
          }
        },
        {
          "type": "StaticRequestFilter",
          "config": {
            "method": "POST",
            "uri": "http://app.example.com:8081/login",
            "form": {
              "username": [
                "${request.headers['Username'][0]}"
              ],
              "password": [
                "${request.headers['Password'][0]}"
              ]
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  },
  "condition": "${find(request.uri.path, '^/db')}"
}
```

Notice the following features of the route:

- The route matches requests to `/db`.
- The JdbcDataSource in the heap sets up the connection to the database.
- The ScriptableFilter calls `SqlAccessFilter.groovy` to look up credentials over SQL.

`SqlAccessFilter.groovy`, in turn, calls `SqlClient.groovy` to access the database to get the credentials.

   - The StaticRequestFilter uses the credentials to build a login request.

     Although the script sets the scheme to HTTPS, for convenience in this example, the StaticRequestFilter resets the URI to HTTP.

5. To test the setup, go to a URL with a query string parameter that specifies an email address in the database, such as `http://ig.example.com:8080/db?mail=george@example.com`.

   The sample application profile page for the user is displayed.

Extend IG through the Java API

IMPORTANT

When you are writing scripts or Java extensions, never use a `Promise` blocking method, such as `get()`, `getOrThrow()`, or `getOrThrowUninterruptibly()`, to obtain the response.

A promise represents the result of an asynchronous operation. Therefore, using a blocking method to wait for the result can cause deadlocks and/or race issues.

IG includes a complete Java application programming interface to allow you to customize IG to perform complex server interactions or intensive data transformations that you cannot achieve with scripts or the existing handlers, filters, and expressions described in Expressions. The following sections describe how to extend IG through the Java API:

Key extension points

Interface Stability: Evolving, as defined in ForgeRock product stability labels.

The following interfaces are available:

Decorator

A `Decorator` adds new behavior to another object without changing the base type of the object.

When suggesting custom `Decorator` names, know that IG reserves all field names that use only alphanumeric characters. To avoid clashes, use dots or dashes in your field names, such as `my-decorator`.

ExpressionPlugin

An `ExpressionPlugin` adds a node to the `Expression` context tree, alongside `env` (for environment variables), and `system` (for system properties). For example, the expression `${system['user.home']}` yields the home directory of the user running the application server for IG.

In your `ExpressionPlugin`, the `getKey()` method returns the name of the node, and the `getObject()` method returns the unified expression language context object that contains the values needed to resolve the expression. The plugins for `env` and `system` return Map objects, for example.

When you add your own `ExpressionPlugin`, you must make it discoverable within your custom library. You do this by adding a services file named after the plugin interface, where the file contains the fully qualified class name of your plugin, under `META-INF/services/org.forgerock.openig.el.ExpressionPlugin` in the .jar file for your customizations. When you have more than one plugin, add one fully qualified class name per line. For information, refer to the reference documentation for the Java class ServiceLoader. If you build your project using Maven, then you can add this under the `src/main/resources` directory. Add custom libraries, as described in Embed customizations in IG.

Remember to provide documentation for IG administrators on how your plugin extends expressions.

Filter

A `Filter` serves to process a request before handing it off to the next element in the chain, in a similar way to an interceptor programming model.

The `Filter` interface exposes a `filter()` method, which takes a Context, a Request, and the Handler, which is the next filter or handler to dispatch to. The `filter()` method returns a Promise that provides access to the Response with methods for dealing with both success and failure conditions.

A filter can elect not to pass the request to the next filter or handler, and instead handle the request itself. It can achieve this by merely avoiding a call to `next.handle(context, request)`, creating its own response object and returning that in the promise. The filter is also at liberty to replace a response with another of its own. A filter can exist in more than one chain, therefore should make no assumptions or correlations using the chain it is supplied. The only valid use of a chain by a filter is to call its `handle()` method to dispatch the request to the rest of the chain.

Handler

A `Handler` generates a response for a request.

The `Handler` interface exposes a `handle()` method, which takes a Context, and a Request. It processes the request and returns a Promise that provides access to the link:../_attachments/apidocs/org/forgerock/http/protocol/Response .html[Response] with methods for dealing with both success and failure conditions. A handler can elect to dispatch the request to another handler or chain.

ClassAliasResolver

A `ClassAliasResolver` makes it possible to replace a fully qualified class name with a short name (an alias) in an object declaration's type.

The `ClassAliasResolver` interface exposes a `resolve(String)` method to do the following:

- Return the class mapped to a given alias
- Return `null` if the given alias is unknown to the resolver

  All resolvers available to IG are asked until the first non-null value is returned or until all resolvers have been contacted.

  The order of resolvers is nondeterministic. To prevent conflicts, don't use the same alias for different types.

Implement a customized sample filter

The `SampleFilter` class implements the `Filter` interface to set a header in the incoming request and in the outgoing response.

In the following example, the sample filter adds an arbitrary header:

```
package org.forgerock.openig.doc.examples;

import org.forgerock.http.Filter;
import org.forgerock.http.Handler;
import org.forgerock.http.protocol.Request;
import org.forgerock.http.protocol.Response;
import org.forgerock.openig.heap.GenericHeaplet;
import org.forgerock.openig.heap.HeapException;
import org.forgerock.openig.model.type.service.NoTypeInfo;
import org.forgerock.services.context.Context;
import org.forgerock.util.promise.NeverThrowsException;
import org.forgerock.util.promise.Promise;

/**
 * Filter to set a header in the incoming request and in the
outgoing response.
 */
public class SampleFilter implements Filter {

    /** Header name. */
    String name;

    /** Header value. */
    String value;
    /**
     * Set a header in the incoming request and in the outgoing
response.
     * A configuration example looks something like the
following.
     *
     * <pre>
     * {
     *      "name": "SampleFilter",
     *      "type": "SampleFilter",
     *      "config": {
     *          "name": "X-Greeting",
     *          "value": "Hello world"
     *      }
     * }
     * </pre>
     *
     * @param context            Execution context.
     * @param request            HTTP Request.
     * @param next               Next filter or handler in the
chain.
     * @return A {@code Promise} representing the response to be
returned to the client.
     */
    @Override
    public Promise<Response, NeverThrowsException> filter(final
Context context,
                                                          final
Request request,
                                                          final
Handler next) {

        // Set header in the request.
        request.getHeaders().put(name, value);

        // Pass to the next filter or handler in the chain.
        return next.handle(context, request)
                   // When it has been successfully executed,
execute the following callback
                   .thenOnResult(response -> {
                       // Set header in the response.
                       response.getHeaders().put(name, value);
                   });
    }
    /**
     * Create and initialize the filter, based on the
configuration.
     * The filter object is stored in the heap.
     */
    @NoTypeInfo
    public static class Heaplet extends GenericHeaplet {

        /**
         * Create the filter object in the heap,
         * setting the header name and value for the filter,
         * based on the configuration.
         *
         * @return                  The filter object.
         * @throws HeapException    Failed to create the object.
         */
        @Override
        public Object create() throws HeapException {

            SampleFilter filter = new SampleFilter();
            filter.name  =
config.get("name").as(evaluatedWithHeapProperties()).required().a
sString();
            filter.value =
config.get("value").as(evaluatedWithHeapProperties()).required().
asString();

            return filter;
        }
    }
}
```

The corresponding filter configuration is similar to this:

```
{
  "name": "SampleFilter",
  "type": "org.forgerock.openig.doc.examples.SampleFilter",
  "config": {
    "name": "X-Greeting",
    "value": "Hello world"
  }
}
```

Note how `type` is configured with the fully qualified class name for `SampleFilter`. To simplify the configuration, implement a class alias resolver, as described in Implement a Class Alias Resolver.

Implement a class alias resolver

To simplify the configuration of a customized object, implement a `ClassAliasResolver` to allow the use of short names instead of fully qualified class names.

In the following example, a `ClassAliasResolver` is created for the `SampleFilter` class:

```
package org.forgerock.openig.doc.examples;

import static java.util.stream.Collectors.toUnmodifiableSet;

import java.util.HashMap;
import java.util.Map;
import java.util.Optional;
import java.util.Set;

import org.forgerock.openig.alias.ClassAliasResolver;
import org.forgerock.openig.heap.Heaplet;
import org.forgerock.openig.heap.Heaplets;

/**
 * Allow use of short name aliases in configuration object types.
 *
 * This allows a configuration with {@code "type":
"SampleFilter"}
 * instead of {@code "type":
"org.forgerock.openig.doc.examples.SampleFilter"}.
 */
public class SampleClassAliasResolver implements
ClassAliasResolver {

    private static final Map<String, Class<?>> ALIASES =
            new HashMap<>();

    static {
        ALIASES.put("SampleFilter", SampleFilter.class);
    }

    /**
     * Get the class for a short name alias.
     *
     * @param alias Short name alias.
     * @return      The class, or null if the alias is not
defined.
     */
    @Override
    public Class<?> resolve(final String alias) {
        return ALIASES.get(alias);
    }

    @Override
    public Set<Class<? extends Heaplet>> supportedTypes() {
        return ALIASES.values()
                      .stream()
                      .map(Heaplets::findHeapletClass)
                      .filter(Optional::isPresent)
                      .map(Optional::get)
                      .collect(toUnmodifiableSet());
    }
}
```

With this `ClassAliasResolver`, the filter configuration in Implement a Customized Sample Filter can use the alias instead of the fully qualified class name, as follows:

```
{
  "name": "SampleFilter",
  "type": "SampleFilter",
  "config": {
    "name": "X-Greeting",
    "value": "Hello world"
  }
}
```

To create a customized `ClassAliasResolver`, add a services file with the following characteristics:

- Name the file after the class resolver interface.
- Store the file under `META-INF/services/org.forgerock.openig.alias.ClassAliasResolver`, in the customization .jar file.

  If you build your project using Maven, you can add the file under the `src/main/resources` directory.
- In your ClassAliasResolver file, add a line for the fully qualified class name of your resolver as follows:

```
org.forgerock.openig.doc.examples.SampleClassAliasResolver
```

If you have more than one resolver in your .jar file, add one line for each fully qualified class name.

Configure the heap object for the customization

Objects are added to the heap and supplied with configuration artifacts at initialization time. To be integrated with the configuration, a class must have an accompanying implementation of the Heaplet interface. The easiest and most common way of exposing the heaplet is to extend the GenericHeaplet class in a nested class of the class you want to create and initialize, overriding the heaplet's `create()` method.

Within the `create()` method, you can access the object's configuration through the `config` field.

Embed customizations in IG

1. Build your IG extension into a .jar file.
2. Create the directory `$HOME/.openig/extra`, where `$HOME/.openig` is the instance directory:

   ```
   $ mkdir $HOME/.openig/extra
   ```

3. Add the .jar file to the directory. The following example adds `sample-filter.jar` to `$HOME/.openig/extra`:

   ```
   $ cp ~/sample-filter/target/sample-filter.jar
   $HOME/.openig/extra
   ```

4. If the extension has dependencies that are not included in IG, also add them to the directory.
5. Start IG, as described in Start IG with default settings.

Record custom audit events

This section describes how to record a custom audit event to standard output. The example is based on the example in Validate access tokens through the introspection endpoint, adding an audit event for the custom topic `OAuth2AccessTopic`.

To record custom audit events to other outputs, adapt the route in the following procedure to use another audit event handler.

For information about how to configure supported audit event handlers, and exclude sensitive data from log files, refer to Auditing your deployment. For more information about audit event handlers, refer to Audit framework.

Record custom audit events to standard output

Before you start, prepare IG and the sample application as described in the Getting started.

1. Set up AM as described in Validate access tokens through the introspection endpoint.
2. Define the schema of an event topic called `OAuth2AccessTopic` by adding the following route to IG:
   1. Linux
   2. Windows

```
$HOME/.openig/audit-schemas/OAuth2AccessTopic.json
```

```
%appdata%\OpenIG\OpenIG\audit-
schemas/OAuth2AccessTopic.json
```

```
{
  "schema": {
    "$schema": "http://json-schema.org/draft-04/schema#",
    "id": "OAuth2Access",
    "type": "object",
    "properties": {
      "_id": {
        "type": "string"
      },
      "timestamp": {
        "type": "string"
      },
      "transactionId": {
        "type": "string"
      },
      "eventName": {
        "type": "string"
      },
      "accessToken": {
        "type": "object",
        "properties": {
          "scopes": {
            "type": "array",
            "items": {
              "type": "string"
            }
          },
          "expiresAt": "number",
          "sub": "string"
        },
        "required": [ "scopes" ]
      },
      "resource": {
        "type": "object",
        "properties": {
          "path": {
            "type": "string"
          },
          "method": {
            "type": "string"
          }
        }
      }
    }
  },
  "filterPolicies": {
    "field": {
      "includeIf": [
        "/_id",
        "/timestamp",
        "/eventName",
        "/transactionId",
        "/accessToken",
        "/resource"
      ]
    }
  },
  "required": [ "_id", "timestamp", "transactionId",
"eventName" ]
}
```

Notice that the schema includes the following fields:

- Mandatory fields `_id`, `timestamp`, `transactionId`, and `eventName`.

- accessToken , to include the access token scopes, expiry time, and the subject.
- resource , to include the path and method.
- filterPolicies , to specify additional event fields to include in the logs.

3. Define a script to generate audit events on the topic named OAuth2AccessTopic , by adding the following file to the IG configuration as:
   1. Linux
   2. Windows

```
$HOME/.openig/scripts/groovy/OAuth2Access.groovy
```

```
%appdata%\OpenIG\scripts\groovy/OAuth2Access.groovy
```

```
import static
org.forgerock.json.resource.Requests.newCreateRequest;
import static
org.forgerock.json.resource.ResourcePath.resourcePath;

// Helper functions
def String transactionId() {
    return contexts.transactionId.transactionId.value;
}

def JsonValue auditEvent(String eventName) {
    return json(object(field('eventName', eventName),
            field('transactionId', transactionId()),
            field('timestamp',
clock.instant().toEpochMilli())));
}

def auditEventRequest(String topicName, JsonValue
auditEvent) {
    return newCreateRequest(resourcePath("/" + topicName),
auditEvent);
}

def accessTokenInfo() {
    def accessTokenInfo = contexts.oauth2.accessToken;
    return object(field('scopes', accessTokenInfo.scopes
as List),
            field('expiresAt', accessTokenInfo.expiresAt),
            field('subname',
accessTokenInfo.info.subname));
}

def resourceEvent() {
    return object(field('path', request.uri.path),
            field('method', request.method));
}

// ------------------------------------------

// Build the event
JsonValue auditEvent = auditEvent('OAuth2AccessEvent')
        .add('accessToken', accessTokenInfo())
        .add('resource', resourceEvent());

// Send the event, and log a message if there is an error
auditService.handleCreate(context,
auditEventRequest("OAuth2AccessTopic", auditEvent))
        .thenOnException(e -> logger.warn("An error
occurred while sending the audit event", e));

// Continue onto the next filter
return next.handle(context, request)
```

The script generates audit events named `OAuth2AccessEvent`, on a topic named `OAuth2AccessTopic`. The events conform to the topic schema.

4. Set an environment variable for the IG agent password, and then restart IG:

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

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

5. Add the following route to IG:

   1. Linux
   2. Windows

```
$HOME/.openig/config/routes/30-custom.json
```

```
%appdata%\OpenIG\config\routes\30-custom.json
```

```
{
  "name": "30-custom",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/rs-introspect-
audit')}",
  "heap": [
    {
      "name": "AuditService-1",
      "type": "AuditService",
      "config": {
        "config": {},
        "eventHandlers": [
          {
            "class":
"org.forgerock.audit.handlers.json.stdout.JsonStdoutAuditE
ventHandler",
            "config": {
              "name": "jsonstdout",
              "elasticsearchCompatible": false,
              "topics": [
                "OAuth2AccessTopic"
              ]
            }
          }
        ]
      }
    },
    {
      "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"
                  }
                }
              }
            }
          }
        },
        {
          "type": "ScriptableFilter",
          "config": {
            "type": "application/x-groovy",
            "file": "OAuth2Access.groovy",
            "args": {
              "auditService": "${heap['AuditService-1']}",
              "clock": "${heap['Clock']}"
            }
          }
        }
      ],
      "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-introspect-audit`.
- The `accessTokenResolver` uses the token introspection endpoint to validate the access token.
- The HttpBasicAuthenticationClientFilter adds the credentials to the outgoing token introspection request.
- The ScriptableFilter uses the Groovy script `OAuth2Access.groovy` to generate audit events named `OAuth2AccessEvent`, with a topic named `OAuth2AccessTopic`.
- The audit service publishes the custom audit event to the JsonStdoutAuditEventHandler. A single line per audit event is published to standard output.

6. 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&sc
ope=mail%20employeenumber" \
http://am.example.com:8088/openam/oauth2/access_token |
jq -r ".access_token")
```

b. Access the route, with the access_token returned in the previous step:

```
$ curl -v http://ig.example.com:8080/rs-introspect-
audit --header "Authorization: Bearer ${mytoken}"
```

Information about the decoded access_token is returned.

c. Search the standard output for an audit message like the following example, that includes an audit event on the topic `OAuth2AccessTopic`:

```
{
  "_id": "fa2...-14",
  "timestamp": 155...541,
  "eventName": "OAuth2AccessEvent",
  "transactionId": "fa2...-13",
  "accessToken": {
    "scopes": ["employeenumber", "mail"],
    "expiresAt": 155...000,
    "subname": "demo"
  },
  "resource": {
    "path": "/rs-introspect-audit",
    "method": "GET"
  },
  "source": "audit",
  "topic": "OAuth2AccessTopic",
  "level": "INFO"
}
```

Configuration templates

This chapter contains template routes for common configurations. To use a template, set up IG as described in the Getting started, and modify the template for your deployment. Before you use a route in production, review the points in Security guide.

Proxy and capture

If you installed and configured IG with a router and default route as described in the Getting started, then you already proxy and capture the application requests coming in and the server responses going out.

This template route uses a `DispatchHandler` to change the scheme to HTTPS on login:

```
{
  "heap": [
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler",
      "comment": "Testing only: blindly trust the server cert
for HTTPS.",
      "config": {
        "tls": {
          "type": "ClientTlsOptions",
          "config": {
            "trustManager": {
              "type": "TrustAllManager"
            },
            "hostnameVerifier": "ALLOW_ALL"
          }
        }
      }
    }
  ],
  "handler": {
    "type": "DispatchHandler",
    "config": {
      "bindings": [
        {
          "condition": "${request.uri.path == '/login'}",
          "handler": "ReverseProxyHandler",
          "baseURI": "https://app.example.com:8444"
        },
        {
          "condition": "${request.uri.scheme == 'http'}",
          "handler": "ReverseProxyHandler",
          "baseURI": "http://app.example.com:8081"
        },
        {
          "handler": "ReverseProxyHandler",
          "baseURI": "https://app.example.com:8444"
        }
      ]
    }
  },
  "condition": "${find(request.uri.query, 'demo=capture')}"
}
```

To try this example with the sample application:

1. Add the following route to IG:

   1. Linux
   2. Windows

   ```
   $HOME/.openig/config/routes/20-capture.json
   ```

   ```
   %appdata%\OpenIG\config\routes\20-capture.json
   ```

2. Add the following route to serve static resources, such as .css, for the sample application:

   1. Linux
   2. Windows

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

   ```
   %appdata%\OpenIG\config\routes\static-resources.json
   ```

   ```
   {
     "name" : "sampleapp-resources",
     "baseURI" : "http://app.example.com:8081",
     "condition": "${find(request.uri.path,'^/css')}",
     "handler": "ReverseProxyHandler"
   }
   ```

3. Go to http://ig.example.com:8080/login?demo=capture.

   The login page of the sample application is displayed.

To use this as a default route with a real application:

1. Replace the test ReverseProxyHandler with one that is configured to trust the application’s public key server certificate. Otherwise, use a ReverseProxyHandler that references a truststore holding the certificate.

Configure the ReverseProxyHandler to strictly verifiy hostnames for outgoing SSL connections.

In production, do not use `TrustAllManager` for `trustManager`, or `ALLOW_ALL` for `hostnameVerifier`.

2. Change the baseURI settings to match the target application.
3. Remove the route-level condition on the handler that specifies a `demo` query string parameter.

Simple login form

This template route intercepts the login page request, replaces it with a login form, and logs the user into the target application with hard-coded username and password:

```
{
  "heap": [
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler",
      "comment": "Testing only: blindly trust the server cert
for HTTPS.",
      "config": {
        "tls": {
          "type": "ClientTlsOptions",
          "config": {
            "trustManager": {
              "type": "TrustAllManager"
            },
            "hostnameVerifier": "ALLOW_ALL"
          }
        }
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "PasswordReplayFilter",
          "config": {
            "loginPage": "${request.uri.path == '/login'}",
            "request": {
              "method": "POST",
              "uri": "https://app.example.com:8444/login",
              "form": {
                "username": [
                  "MY_USERNAME"
                ],
                "password": [
                  "MY_PASSWORD"
                ]
              }
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  },
  "condition": "${find(request.uri.query, 'demo=simple')}"
}
```

To try this example with the sample application:

1. Add the following route to IG:
    1. Linux
    2. Windows

    ```
    $HOME/.openig/config/routes/21-simple.json
    ```

    ```
    %appdata%\OpenIG\config\routes\21-simple.json
    ```

2. Replace `MY_USERNAME` with `demo`, and `MY_PASSWORD` with `Ch4ng31t`.
3. Add the following route to serve static resources, such as .css, for the sample application:
    1. Linux
    2. Windows

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

    ```
    %appdata%\OpenIG\config\routes\static-resources.json
    ```

```
{
  "name" : "sampleapp-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css')}",
  "handler": "ReverseProxyHandler"
}
```

4. Go to http://ig.example.com:8080/login?demo=simple.

   The sample application profile page for the demo user displays information about the request:

   ```
   Username        demo

   REQUEST INFORMATION
   Method  POST
   URI     /login
   Cookies
   …
   ```

To use this as a default route with a real application:

1. Replace the test ReverseProxyHandler with one that is configured to trust the application's public key server certificate. Otherwise, use a ReverseProxyHandler that references a truststore holding the certificate.

   Configure the ReverseProxyHandler to strictly verifiy hostnames for outgoing SSL connections.

   In production, do not use `TrustAllManager` for `trustManager`, or `ALLOW_ALL` for `hostnameVerifier`.
2. Change the `uri`, `form`, and `baseURI` to match the target application.
3. Remove the route-level condition on the handler that specifies a `demo` query string parameter.

Login form with cookie from login page

Like the previous route, this template route intercepts the login page request, replaces it with the login form, and logs the user into the target application with hard-coded username and password. This route also adds a CookieFilter to manage cookies.

The route uses a default `CookieFilter` to manage cookies. In this default configuration, cookies from the protected application are intercepted and stored in the IG session. They are not sent to the browser. For information, see CookieFilter.

```
{
  "heap": [
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler",
      "comment": "Testing only: blindly trust the server cert
for HTTPS.",
      "config": {
        "tls": {
          "type": "ClientTlsOptions",
          "config": {
            "trustManager": {
              "type": "TrustAllManager"
            },
            "hostnameVerifier": "ALLOW_ALL"
          }
        }
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "PasswordReplayFilter",
          "config": {
            "loginPage": "${request.uri.path == '/login'}",
            "request": {
              "method": "POST",
              "uri": "https://app.example.com:8444/login",
              "form": {
                "username": [
                  "MY_USERNAME"
                ],
                "password": [
                  "MY_PASSWORD"
                ]
              }
            }
          }
        },
        {
          "type": "CookieFilter"
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  },
  "condition": "${find(request.uri.query, 'demo=cookie')}"
}
```

To try this example with the sample application:

1. Add the following route to IG:

    1. Linux
    2. Windows

    ```
    $HOME/.openig/config/routes/22-cookie.json
    ```

    ```
    %appdata%\OpenIG\config\routes\22-cookie.json
    ```

2. Replace `MY_USERNAME` with `kramer`, and `MY_PASSWORD` with `N3wman12`.
3. Add the following route to serve static resources, such as .css, for the sample application:

    1. Linux
    2. Windows

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

    ```
    %appdata%\OpenIG\config\routes\static-resources.json
    ```

    ```
    {
      "name" : "sampleapp-resources",
      "baseURI" : "http://app.example.com:8081",
      "condition": "${find(request.uri.path,'^/css')}",
      "handler": "ReverseProxyHandler"
    }
    ```

4. Go to http://ig.example.com:8080/login?demo=cookie.

    The sample application page is displayed.

```
Method    POST
URI       /login
Cookies
Headers  content-type: application/x-www-form-urlencoded
         content-length: 31
         host: app.example.com:8444
         connection: Keep-Alive
         user-agent: Apache-HttpAsyncClient/… (Java/…)
```

5. Refresh your connection to http://ig.example.com:8080/login?demo=cookie.

   Compared to the example in Login form with cookie from login page, this example displays additional information about the session cookie:

   ```
   Cookies  session-cookie=123…
   ```

To use this as a default route with a real application:

1. Replace the test ReverseProxyHandler with one that is configured to trust the application's public key server certificate. Otherwise, use a ReverseProxyHandler that references a truststore holding the certificate.

   Configure the ReverseProxyHandler to strictly verifiy hostnames for outgoing SSL connections.

   In production, do not use `TrustAllManager` for `trustManager`, or `ALLOW_ALL` for `hostnameVerifier`.
2. Change the `uri` and `form` to match the target application.
3. Remove the route-level condition on the handler that specifies a `demo` query string parameter.

Login form with password replay and cookie filters

When a user without a valid session tries to access a protected application, this template route works with an application to return a login page.

The route uses a PasswordReplayFilter to find the login page by using a pattern that matches a mock AM Classic UI page.

Cookies sent by the user agent are retained in the CookieFilter, and not forwarded to the protected application. Similarly, set-cookies sent by the protected application are retained in the CookieFilter and not forwarded back to the user agent.

The route uses a default `CookieFilter` to manage cookies. In this default configuration, cookies from the protected application are intercepted and stored in the IG session. They are not sent to the browser. For information, see CookieFilter.

```
{
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "PasswordReplayFilter",
          "config": {
            "loginPageContentMarker": "OpenAM\\s\\(Login\\)",
            "request": {
              "comments": [
                "An example based on OpenAM classic UI: ",
                "uri is for the OpenAM login page; ",
                "IDToken1 is the username field; ",
                "IDToken2 is the password field; ",
                "host takes the OpenAM FQDN:port.",
                "The sample app simulates OpenAM."
              ],
              "method": "POST",
              "uri":
"http://app.example.com:8081/openam/UI/Login",
              "form": {
                "IDToken0": [
                  ""
                ],
                "IDToken1": [
                  "demo"
                ],
                "IDToken2": [
                  "Ch4ng31t"
                ],
                "IDButton": [
                  "Log+In"
                ],
                "encoded": [
                  "false"
                ]
              },
              "headers": {
                "host": [
                  "app.example.com:8081"
                ]
              }
            }
          }
        },
        {
          "type": "CookieFilter"
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  },
  "condition": "${find(request.uri.query, 'demo=classic')}"
}
```

To try this example with the sample application:

1. Save the file as `$HOME/.openig/config/routes/23-classic.json`.
2. Use the following **`curl`** command to check that it works:

```
$ curl -D- http://ig.example.com:8080/login?demo=classic

HTTP/1.1 200 OK
Set-Cookie: IG_SESSIONID=24446BA29E866F840197C8E0EAD57A89;
Path=/; HttpOnly
...
```

To use this as a default route with a real application:

1. Change the `uri` and `form` to match the target application.
2. Remove the route-level condition on the handler that specifies a `demo` query string parameter.

Login which requires a hidden value from the login page

This template route extracts a hidden value from the login page, and includes it the static login form that it then POSTs to the target application.

```
{
  "properties": {
    "appBaseUri":  "https://app.example.com:8444"
  },
  "heap": [
      {
        "name": "ReverseProxyHandler",
        "type": "ReverseProxyHandler",
        "comment": "Testing only: blindly trust the server cert
for HTTPS.",
        "config": {
          "tls": {
            "type": "ClientTlsOptions",
            "config": {
              "trustManager": {
                "type": "TrustAllManager"
              },
              "hostnameVerifier": "ALLOW_ALL"
            }
          }
        }
      }
    ],
    "handler": {
      "type": "Chain",
      "config": {
        "filters": [
          {
            "type": "PasswordReplayFilter",
            "config": {
              "loginPage": "${request.uri.path == '/login'}",
              "loginPageExtractions": [
                {
                  "name": "hidden",
                  "pattern": "loginToken\\s+value=\"(.*)\""
                }
              ],
              "request": {
                "method": "POST",
                "uri": "${appBaseUri}/login",
                "form": {
                  "username": [
                    "MY_USERNAME"
                  ],
                  "password": [
                    "MY_PASSWORD"
                  ],
                  "hiddenValue": [
                    "${attributes.extracted.hidden}"
                  ]
                }
              }
            }
          }
        ],
        "handler": "ReverseProxyHandler"
      }
    },
    "condition": "${find(request.uri.query, 'demo=hidden')}",
    "baseURI": "${appBaseUri}"
  }
```

The parameters in the PasswordReplayFilter form, `MY_USERNAME` and `MY_PASSWORD`, can have string values or can use expressions.

To try this example with the sample application:

1. Add the following route to IG:
   1. Linux
   2. Windows

   ```
   $HOME/.openig/config/routes/24-hidden.json
   ```

   ```
   %appdata%\OpenIG\config\routes\24-hidden.json
   ```

2. Replace `MY_USERNAME` with `scarter`, and `MY_PASSWORD` with `S9rain12`.
3. Add the following route to serve static resources, such as .css, for the sample application:
   1. Linux
   2. Windows

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

   ```
   %appdata%\OpenIG\config\routes\static-resources.json
   ```

   ```
   {
     "name" : "sampleapp-resources",
     "baseURI" : "http://app.example.com:8081",
     "condition": "${find(request.uri.path,'^/css')}",
   ```

```
    "handler": "ReverseProxyHandler"
  }
```

4. Go to http://ig.example.com:8080/login?demo=hidden.

To use this as a default route with a real application:

1. Replace the test ReverseProxyHandler with one that is configured to trust the application’s public key server certificate. Otherwise, use a ReverseProxyHandler that references a truststore holding the certificate.

   Configure the ReverseProxyHandler to strictly verifiy hostnames for outgoing SSL connections.

   In production, do not use `TrustAllManager` for `trustManager`, or `ALLOW_ALL` for `hostnameVerifier`.
2. Change the `loginPage`, `loginPageExtractions`, `uri`, and `form` to match the target application.
3. Remove the route-level condition on the handler that specifies a `demo` query string parameter.

HTTP and HTTPS application

This template route proxies traffic to an application with both HTTP and HTTPS ports. The application uses HTTPS for authentication and HTTP for the general application features. Assuming that all login requests are made over HTTPS, you must add the login filters and handlers to the chain.

```
{
  "heap": [
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler",
      "comment": "Testing only: blindly trust the server cert
for HTTPS.",
      "config": {
        "tls": {
          "type": "ClientTlsOptions",
          "config": {
            "trustManager": {
              "type": "TrustAllManager"
            },
            "hostnameVerifier": "ALLOW_ALL"
          }
        }
      }
    }
  ],
  "handler": {
    "type": "DispatchHandler",
    "config": {
      "bindings": [
        {
          "condition": "${request.uri.scheme == 'http'}",
          "handler": "ReverseProxyHandler",
          "baseURI": "http://app.example.com:8081"
        },
        {
          "condition": "${request.uri.path == '/login'}",
          "handler": {
            "type": "Chain",
            "config": {
              "comment": "Add one or more filters to handle
login.",
              "filters": [],
              "handler": "ReverseProxyHandler"
            }
          },
          "baseURI": "https://app.example.com:8444"
        },
        {
          "handler": "ReverseProxyHandler",
          "baseURI": "https://app.example.com:8444"
        }
      ]
    }
  },
  "condition": "${find(request.uri.query, 'demo=https')}"
}
```

To try this example with the sample application:

1. Add the following route to IG:
    1. Linux
    2. Windows

    ```
    $HOME/.openig/config/routes/25-https.json
    ```

```
%appdata%\OpenIG\config\routes\25-https.json
```

2. Add the following route to serve static resources, such as .css, for the sample application:

   1. Linux
   2. Windows

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

```
%appdata%\OpenIG\config\routes\static-resources.json
```

```
{
  "name" : "sampleapp-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css')}",
  "handler": "ReverseProxyHandler"
}
```

3. Go to http://ig.example.com:8080/login?demo=https.

   The login page of the sample application is displayed.

To use this as a default route with a real application:

1. Replace the test ReverseProxyHandler with one that is configured to trust the application's public key server certificate. Otherwise, use a ReverseProxyHandler that references a truststore holding the certificate.

   Configure the ReverseProxyHandler to strictly verifiy hostnames for outgoing SSL connections.

   In production, do not use `TrustAllManager` for `trustManager`, or `ALLOW_ALL` for `hostnameVerifier`.
2. Change the `loginPage`, `loginPageExtractions`, `uri`, and `form` to match the target application.
3. Remove the route-level condition on the handler that specifies a `demo` query string parameter.

1. Replace the test ReverseProxyHandler with one that is configured to trust the application's public key server certificate. Otherwise, use a ReverseProxyHandler that references a truststore holding the certificate.

   Configure the ReverseProxyHandler to strictly verifiy hostnames for outgoing SSL connections.

In production, do not use `TrustAllManager` for `trustManager`, or `ALLOW_ALL` for `hostnameVerifier`.

2. Change the `loginPage`, `loginPageExtractions`, `uri`, and `form` to match the target application.
3. Remove the route-level condition on the handler that specifies a `demo` query string parameter.

AM integration with headers

This template route logs the user into the target application by using headers such as those passed in from an AM policy agent. If the passed in header contains only a user name or subject and requires a lookup to an external data source, you must add an attribute filter to the chain to retrieve the credentials.

```
{
  "heap": [
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler",
      "comment": "Testing only: blindly trust the server cert
for HTTPS.",
      "config": {
        "tls": {
          "type": "ClientTlsOptions",
          "config": {
            "trustManager": {
              "type": "TrustAllManager"
            },
            "hostnameVerifier": "ALLOW_ALL"
          }
        }
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "PasswordReplayFilter",
          "config": {
            "loginPage": "${request.uri.path == '/login'}",
            "request": {
              "method": "POST",
              "uri": "https://app.example.com:8444/login",
              "form": {
                "username": [
                  "${request.headers['username'][0]}"
                ],
                "password": [
                  "${request.headers['password'][0]}"
                ]
              }
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  },
  "condition": "${find(request.uri.query, 'demo=headers')}"
}
```

To try this example with the sample application:

1. Add the route to IG:
   1. Linux
   2. Windows

```
$HOME/.openig/config/routes/26-headers.json
```

```
%appdata%\OpenIG\config\routes\26-headers.json
```

2. Use the **`curl`** command to simulate the headers being passed in from an AM policy agent, as in the following example:

```
$ curl \
--header "username: kvaughan" \
--header "password: B5ibery12" \
http://ig.example.com:8080/login?demo=headers

...
<title id="welcome">Howdy, kvaughan</title>
...
```

To use this as a default route with a real application:

1. Replace the test ReverseProxyHandler with one that is configured to trust the application's public key server certificate. Otherwise, use a ReverseProxyHandler that references a truststore holding the certificate.

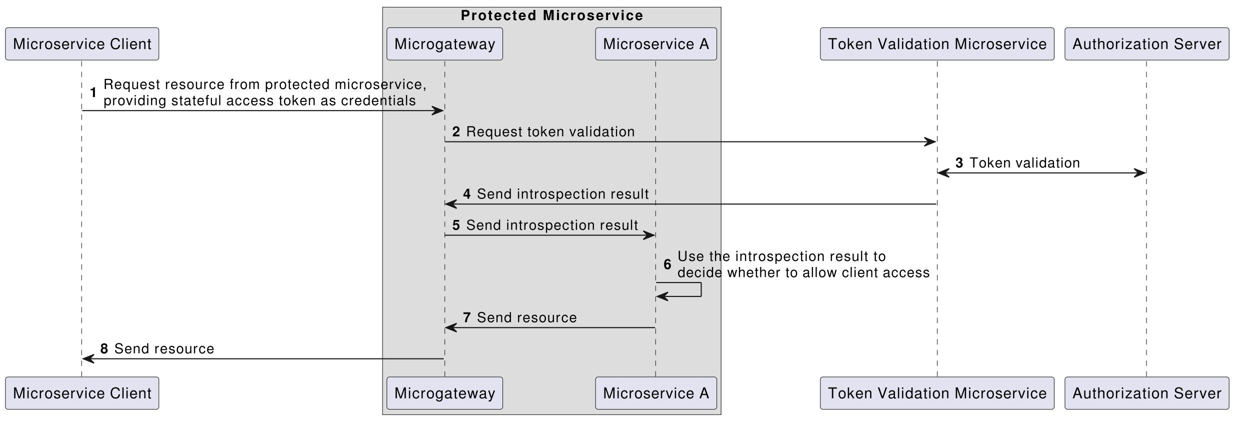
   Configure the ReverseProxyHandler to strictly verifiy hostnames for outgoing SSL connections.

   In production, do not use `TrustAllManager` for `trustManager`, or `ALLOW_ALL` for `hostnameVerifier`.
2. Change the `loginPage`, `uri`, and `form` to match the target application.
3. Remove the route-level condition on the handler that specifies a `demo` query string parameter.

IG 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 IG 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 Using 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 IG:

   1. Linux

   2. Windows

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

```
%appdata%\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 IG 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, IG, 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 IG 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.

Copyright © 2010-2023 ForgeRock, all rights reserved.