Configure

This guide shows you how to configure PingGateway software features.

Configuration files and routes

PingGateway processes requests and responses by using the following JSON files: admin.json, config.json, Route, and Router.

Configuration location

The following table summarizes the default location of the PingGateway configuration and logs.

Purpose	Default location on Linux	Default location on Windows
Log messages from PingGateway and third-party dependencies	`$HOME/.openig/logs`	`%appdata%\OpenIG\logs`
Administration (admin.json) Gateway (config.json)	`$HOME/.openig/config`	`%appdata%\OpenIG\config`
Routes (Route)	`$HOME/.openig/config/routes`	`%appdata%\OpenIG\config\routes`
SAML 2.0	`$HOME/.openig/SAML`	`%appdata%\OpenIG\SAML`
Groovy scripts for scripted filters and handlers, and other objects	`$HOME/.openig/scripts/groovy`	`%appdata%\OpenIG\scripts\groovy`

Purpose	Default location on Linux	Default location on Windows
Temporary directory To change the directory, configure `temporaryDirectory` in admin.json	`$HOME/.openig/tmp`	`%appdata%\OpenIG\tmp`
JSON schema for custom audit To change the directory, configure `topicsSchemasDirectory` in AuditService.	`$HOME/.openig/audit-schemas`	`%appdata%\OpenIG\audit-schemas`

Configuration security

Allow the following access to `$HOME/.openig/logs`, `$HOME/.openig/tmp`, and all configuration directories:

- Highest privilege the PingGateway system account.
- Least privilege for specific accounts, on a case-by-case basis
- No privilege for all other accounts, by default

Change the base location

By default, the base location for PingGateway configuration files is in the following directory:

Linux Windows

```
$HOME/.openig
```

Change the location by using an argument with the startup command. The following example reads the configuration from the `config` directory under `/path/to/config-dir`:

Linux Windows

```
$ /path/to/identity-gateway-2024.11.0/bin/start.sh
```

```
/path/to/config-dir
```

Route names, IDs, and filenames

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

The router scans the `$HOME/.openig/config/routes` folder for files with the .json extension. It uses the route `name` property to order the routes in the configuration. If the route doesn't 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 can't be `default.json`, and the route's `name` property and route ID can't be `default`.

Inline and heap objects

Inline objects

An inline object is declared in a route or configuration, outside of the heap.

The following example shows an inline declaration for a handler to route requests:

```
{
  "handler": {
    "name": "My Router",
    "type": "Router"
  }
}
```

The `name` property for inline objects is optional but useful for logging.

Other objects in the configuration can never refer to named or unnamed inline objects.

Heap objects

A heap object is declared inside the heap.

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

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

The `name` property for heap objects is required. Other objects in the configuration or its child configirations can refer to the heap obect by its `name` property.

Comment the configuration

JSON format doesn't specify a notation for comments. If PingGateway does not recognize a JSON field name, it ignores the field. As a result, it's 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
    }
  }
```

Restart after configuration change

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

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

- Change the configuration of any route, when the `scanInterval` of Router is `disabled` (refer to 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`.

Prevent reload of routes

To prevent routes from being reloaded after startup, stop PingGateway, edit the router `scanInterval`, and restart PingGateway. 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"
  }
}
```

Reserved routes

For information about reserved routes, refer to Reserved routes.

Routes and Common REST

NOTE

When PingGateway is in production mode, you can't manage, list, or read routes through Common REST. For information about switching to development mode, refer to Operating modes.

Through Common REST, you can read, add, delete, and edit routes on PingGateway without manually accessing the file system. You can also list the routes in the order that they're 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 Common REST.

Manage routes through Common REST
Before you start, prepare PingGateway as described in the Quick install.

1. Add the following route to PingGateway:

   Linux | Windows

   ```
   $HOME/.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/_rout
   er/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/_rout
   er/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/_rout
er/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/_rout
er/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/_rou
ter/routes?_queryFilter=true"
```

The list of loaded routes is displayed.

Decorators

Decorators are heap objects to extend what other objects can do. PingGateway defines `baseURI`, `capture`, `timer`, and `tracing` decorators you can use without explicitly configuring them. You can find more information about available decorators in Decorators.

Use decorations that are compatible with the object type. For example, `timer` records the time to process filters and handlers, but doesn't 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. PingGateway 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.

Decorate 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"
    }
  }
}
```

Decorate 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"
    }
  }
}
```

Decorate 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"
    }
  }
}
```

Decorate named objects 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.

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.

Decorate PingGateway's interactions with AM

To log interactions between PingGateway 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 PingGateway route of Validate access tokens with introspection. Test the setup as described for the example, and note that the route's log file contains an HTTP call to the introspection endpoint.

Decorate an object multiple times

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, PingGateway 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`.

Operating modes

Production mode (immutable mode)

To prevent unwanted changes to the configuration, PingGateway is by default in production mode after installation. Production mode has the following characteristics:

- The `/routes` endpoint isn't exposed or accessible.
- Studio is effectively disabled. You can't 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`.

Development mode (mutable mode)

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

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

Use development mode to evaluate or demo PingGateway, or to develop configurations on a single instance. This mode isn't suitable for production.

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

Switch from production mode to development mode

Switch from production mode to development mode in one of the following ways, applied in order of precedence:

1. Add the following configuration to `admin.json`, and restart PingGateway:

```
{
  "mode": "DEVELOPMENT",
  "connectors": [
    { "port" : 8080 }
  ]
}
```

2. Define an environment variable for the configuration token `ig.run.mode`, and then start PingGateway in the same terminal.

   If `mode` is not defined in `admin.json`, the following example starts an instance of PingGateway in development mode:

   Linux | Windows

   ```
   $ IG_RUN_MODE=development /path/to/identity-gateway-
   2024.11.0/bin/bin/start.sh
   ```

3. Define a system property for the configuration token `ig.run.mode` when you start PingGateway.

   If `mode` is not defined in `admin.json`, or an `IG_RUN_MODE` environment variable is not set, the following file starts an instance of PingGateway with the system property `ig.run.mode` to force development mode:

   Linux | Windows

   ```
   $HOME/.openig/env.sh
   ```

   ```
   export JAVA_OPTS='-Dig.run.mode=development'
   ```

Switch from development mode to production mode

Switch from development mode to production mode to prevent unwanted changes to the configuration.

1. In `$HOME/.openig/config/admin.json` (on Windows, `%appdata%\OpenIG\config`), change the value of `mode` from `DEVELOPMENT` to `PRODUCTION`:

```
{
  "mode": "PRODUCTION"
}
```

The file changes the operating mode from development mode to production mode. For more information about the `admin.json` file, refer to AdminHttpApplication (`admin.json`).

The value set in `admin.json` overrides any value set by the `ig.run.mode` configuration token when it is used in an environment variable or system property. For information about `ig.run.mode`, refer to Configuration Tokens.

2. (Optional) Prevent routes from being reloaded after startup:
   - To prevent all routes in the configuration from being reloaded, add a `config.json` as described in the Quick install, and configure the `scanInterval` property of the main Router.
   - To prevent individual routes from being reloaded, configure the `scanInterval` of the routers in those routes.

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

   Learn more in Router.

3. Restart PingGateway.

   When PingGateway starts up, the route endpoints aren't displayed in the logs, and aren't available. You can't access Studio on http://ig.example.com:8080/openig/studio.

Configuration templates

This page shows template routes for common configurations. Before you start, set up PingGateway as described in Quick install.

Modify the template routes for your deployment. Before you use a route in production, review the points in Security.

Proxy and capture

When you followed the instructions in Quick install, you enabled PingGateway to proxy and capture application requests and server responses.

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')}"
}
```

Try this example with the sample application:

1. Add the following route to PingGateway:

   Linux | Windows

   ```
   $HOME/.openig/config/routes/20-capture.json
   ```

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

   Linux | Windows

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

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

3. Go to http://ig.example.com:8080/login?demo=capture.

   The sample application display the login page.

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, don't 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 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": "${request.uri.path == '/login'}",
            "request": {
              "method": "POST",
              "uri": "https://app.example.com:8444/login",
              "form": {
                "username": [
                  "MY_USERNAME"
                ],
                "password": [
                  "MY_PASSWORD"
                ]
              }
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  },
  "condition": "${find(request.uri.query, 'demo=simple')}"
}
```

Try this example with the sample application:

1. Add the following route to PingGateway:

   Linux | Windows

   ```
   $HOME/.openig/config/routes/21-simple.json
   ```

2. Replace `MY_USERNAME` with `demo`, and `MY_PASSWORD` with `Ch4ng31t`.
3. Add the following route to PingGateway to serve the sample application .css and other static resources:

   Linux | Windows

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

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

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.

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, don’t 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

This template route intercepts the login page request, replaces it with the login form, and signs the user on to the target application with hard-coded username and password credentials.

The route uses a default CookieFilter to manage cookies. By default, the filter intercepts cookies from the protected application and stores them in the PingGateway session. It doesn’t send the cookies to the browser.

```
{
  "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')}"
}
```

Try this example with the sample application:

1. Add the following route to PingGateway:

   Linux | Windows

   ```
   $HOME/.openig/config/routes/22-cookie.json
   ```

2. Replace `MY_USERNAME` with `kramer`, and `MY_PASSWORD` with `N3wman12`.

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

   Linux | Windows

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

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

4. Go to http://ig.example.com:8080/login?demo=cookie.

   The sample application page displays.

5. Refresh your connection to http://ig.example.com:8080/login?demo=cookie.

   Compared to 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, don't 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 with a pattern to match a mock AM classic UI page.

The route uses a default CookieFilter to manage cookies. The CookieFilter retains cookies from the browser and doesn't forward them to the protected application. Similarly, the CookieFilter retains cookies in set-cookie headers from the protected application and doesn't forward them to the browser.

```
{
  "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')}"
}
```

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 posts a static login form 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`, take string values or expressions.

Try this example with the sample application:

1. Add the following route to PingGateway:

   Linux | Windows

   ```
   $HOME/.openig/config/routes/24-hidden.json
   ```

2. Replace `MY_USERNAME` with `scarter`, and `MY_PASSWORD` with `S9rain12`.
3. Add the following route to PingGateway to serve the sample application .css and other static resources:

   Linux | Windows

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

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

```
    "handler": "ReverseProxyHandler"
  }
```

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, don't 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. When all login requests use 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')}"
}
```

Try this example with the sample application:

1. Add the following route to PingGateway:

   Linux | Windows

   ```
   $HOME/.openig/config/routes/25-https.json
   ```

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

   Linux | Windows

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

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

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, don't 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 using headers like those from AM policy agents. If a header contains only a username or subject to look up in 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')}"
}
```

Try this example with the sample application:

1. Add the route to PingGateway:

   Linux | Windows

   ```
   $HOME/.openig/config/routes/26-headers.json
   ```

2. Use the **`curl`** command to simulate the headers from a policy agent:

   ```
   $ curl \
   --header "username: kvaughan" \
   --header "password: B5ibery12" \
   http://ig.example.com:8080/login?demo=headers
   ...
   <title id="title">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, don't 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.

Extend

To achieve complex server interactions or intensive data transformations that you can't currently achieve with scripts or existing handlers, filters, or expressions, extend

PingGateway through scripting and customization. The following sections describe how to extend PingGateway:

Add .jar files for extensions

PingGateway includes a complete Java application programming interface for extending your deployment with customizations. For more information, refer to Extend PingGateway through the Java API

Create a directory to hold .jar files for PingGateway extensions:

Linux Windows

```
$HOME/.openig/extra
```

When PingGateway starts up, the JVM loads .jar files in the `extra` directory.

Extend PingGateway through scripts

The following sections describe how to extend PingGateway through scripts:

About scripts

IMPORTANT

IMPORTANT

When writing scripts or Java extensions that use the Promise API, avoid the blocking methods `get()`, `getOrThrow()`, and `getOrThrowUninterruptibly()`. 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.

Instead, consider using `then()` methods, such as `thenOnResult()`, `thenAsync()`, or `thenCatch()`, which allow execution blocks to be executed when the response is available.

Blocking code example

```
def response = next.handle(ctx, req).get() // Blocking method 'get' used
response.headers['new']="new header value"
return response
```

Non-blocking code example

```
return next.handle(ctx, req)
             //Process result when it is available
            .thenOnResult { response ->
              response.headers['new']="new header value"
            }
```

PingGateway 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 PingGateway reads from a file. The configuration can optionally supply arguments to the script.

PingGateway 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.
- Access responses returned in promise callback methods.

Before trying the scripts in this chapter, install and configure PingGateway as described in the Quick install.

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. Learn more in CaptureDecorator.

Use a reference file script

The following example defines a ScriptableFilter written in Groovy, and stored in the following file:

Linux | Windows

```
$HOME/.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 PingGateway is installed. If PingGateway 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 `$HOME/.openig/scripts/groovy` (or `%appdata%\OpenIG\scripts\groovy`) is on the classpath when the scripts are executed. If some Groovy scripts aren't in the default package, but instead have their own package names, they belong in the directory corresponding to their package name. For example, a script in package `com.example.groovy` belongs in `$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 PingGateway:

Linux Windows

```
$HOME/.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 PingGateway, to set up headers required by the script when the user logs in:

Linux Windows

```
$HOME/.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 PingGateway, to add an `Authorization` header based on a username and password combination:

   Linux Windows

```
$HOME/.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 PingGateway, to set up headers required by the script when the user logs in:

Linux | Windows

```
$HOME/.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, PingGateway 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 SQL queries

IMPORTANT

The example in this section uses SqlClient, which exposes a JdbcDataSource. Because the JDBC API provides only blocking APIs, the example code offloads JdbcDataSource calls to another thread.

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 PingGateway, to look up user credentials in the database, by email address, and set the credentials in the request headers for the next handler:

   Linux Windows

   ```
   $HOME/.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.
    */
   import org.forgerock.openig.http.protocol.ResponseUtils
   import org.forgerock.util.promise.Promises

   // Return a promise.
   Promises
           // Submit the task to another thread (asynchronous
   execution).
           .executeAsync(service) {
               // Get the credentials with the synchronous JDBC
   APIs.
               new
   SqlClient(dataSource).getCredentials(request.queryParams?.mail
   [0])
           }
           // When the task completes...
           .thenAsync { credentials ->
               // ...with a result, it successfully got the
   credentials.
               // Add the credentials as headers in the request
   object.
               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'

            // Let the chain continue to process the request.
            next.handle(context, request)
        } { exception ->
            // ...with a checked exception
            // because the dataSource has thrown a JDBC
exception,
            // fail the promise with an illegal state
exception.

ResponseUtils.newIllegalStateExceptionPromise(exception)
        } { runtimeException ->
            // ...with a runtime exception
            // because the dataSource had an unrecoverable
error,
            // fail the promise with an illegal state
exception.

ResponseUtils.newIllegalStateExceptionPromise(runtimeException
)
        }
```

3. Add the following script to PingGateway to access the database, and get credentials:

Linux | Windows

```
$HOME/.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 PingGateway to set up headers required by the scripts when the user logs in:

Linux | Windows

```
$HOME/.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": "${heap['JdbcDataSource-1']}",
              "service": "${heap['ScheduledExecutorService']}"
            },
            "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 PingGateway through the Java API

IMPORTANT

IMPORTANT

When writing scripts or Java extensions that use the Promise API, avoid the blocking methods `get()`, `getOrThrow()`, and `getOrThrowUninterruptibly()`. 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.

Instead, consider using `then()` methods, such as `thenOnResult()`, `thenAsync()`, or `thenCatch()`, which allow execution blocks to be executed when the response is available.

Blocking code example

```
def response = next.handle(ctx, req).get() // Blocking method 'get' used
response.headers['new']="new header value"
return response
```

Non-blocking code example

```
return next.handle(ctx, req)
             //Process result when it is available
            .thenOnResult { response ->
              response.headers['new']="new header value"
            }
```

PingGateway includes a complete Java application programming interface to allow you to customize PingGateway 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 PingGateway 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 PingGateway 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 PingGateway.

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 PingGateway.

Remember to provide documentation for PingGateway 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 PingGateway 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(reason = "documentation sample")
    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().as
String();
            filter.value =
config.get("value").as(evaluatedWithHeapProperties()).required().a
sString();

            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 PingGateway

1. Build your PingGateway 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 aren't included in PingGateway, add them to the directory.
5. Start PingGateway, as described in Start and stop PingGateway.

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 with introspection, 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 Audit the deployment. For more information about audit event handlers, refer to Audit framework.

Record custom audit events to standard output
Before you start, prepare PingGateway and the sample application as described in the Quick install.

1. Set up AM as described in Validate access tokens with introspection.
2. Define the schema of an event topic called `OAuth2AccessTopic` by adding the following route to PingGateway:

   Linux | Windows

   ```
   $HOME/.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 PingGateway configuration as:

Linux | Windows

```
$HOME/.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 PingGateway agent password, and then restart PingGateway:

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

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

5. Add the following route to PingGateway:

Linux | Windows

```
$HOME/.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.JsonStdoutAuditEvent
Handler",
            "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&scope
   =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"
}
```

Was this helpful?