

# Install

---

This guide describes how to install and remove PingGateway software. For information about how to install PingGateway for evaluation, refer to the [Quick install](#).

Read the [Release notes](#) before you install.

Product names changed when ForgeRock became part of Ping Identity. PingGateway was formerly known as ForgeRock Identity Gateway, for example. Learn more about the name changes from [New names for ForgeRock products](#)<sup>↗</sup>.

## Prepare to install

---

Before you install, make sure your installation meets the requirements in the [release notes](#).

### Create a PingGateway service account

To limit the impact of a security breach, install and run PingGateway from a dedicated service account. This is optional when evaluating PingGateway, but essential in production installations.

A hacker is constrained by the rights granted to the user account where PingGateway runs; therefore, never run PingGateway as root user.

1. In a terminal window, use a command similar to the following to create a service account:

<b>Linux</b>	<b>Windows</b>
--------------	----------------

```
$ sudo /usr/sbin/useradd \  
--create-home \  
--comment "Account for running PingGateway" \  
--shell /bin/bash PingGateway
```

2. Apply the principle of least privilege to the account, for example:

- Read/write permissions on the installation directory, `/path/to/identity-gateway-2024.9.0`.
- Execute permissions on the scripts in the installation `bin` directory, `/path/to/PingGateway-2024.9.0/bin`.

## Prepare the network

Configure the network to include hosts for PingGateway, AM, and the sample application. Learn more about host files from the Wikipedia entry, [Hosts \(file\)](#)<sup>↗</sup>.

1. Add the following entry to your host file:



```
127.0.0.1 localhost ig.example.com app.example.com
am.example.com
```

## Set up PingOne Advanced Identity Cloud

This documentation contains procedures for setting up items in PingOne Advanced Identity Cloud that you can use with PingGateway. For more information about setting up PingOne Advanced Identity Cloud, refer to the [PingOne Advanced Identity Cloud documentation](#).

### *Authenticate a PingGateway agent to PingOne Advanced Identity Cloud*

#### IMPORTANT

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

You can now authenticate PingGateway agents to PingOne Advanced 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 PingOne Advanced Identity Cloud.

For more information, refer to PingOne Advanced Identity Cloud's [Journeys](#).

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

1. Log in to the PingOne Advanced 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 a PingGateway agent to PingOne Advanced 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 **Q 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.

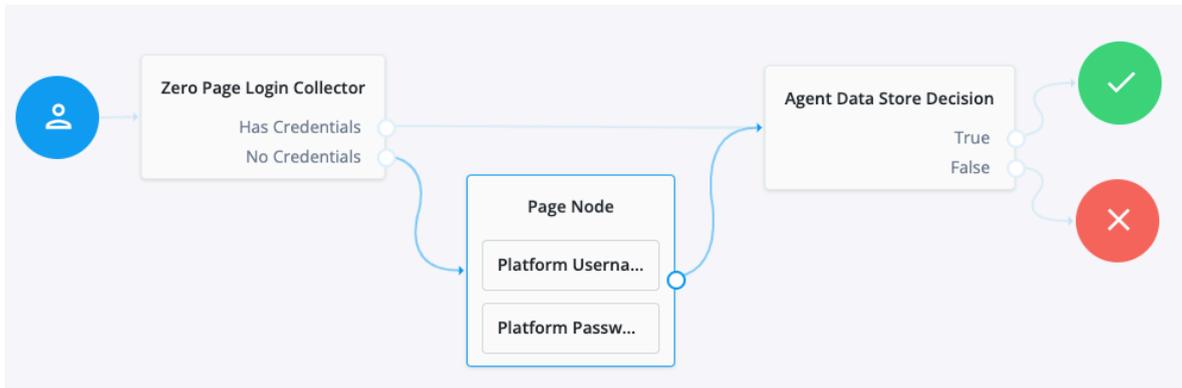
This node is required for compatibility with Java agent and Web agent.

- 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 PingGateway 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:



## Register a PingGateway agent in PingOne Advanced Identity Cloud

This procedure registers an agent profile for PingGateway.

1. Log in to the PingOne Advanced Identity Cloud admin UI as an administrator.
2. Click  **Gateways & Agents** > **+ New Gateway/Agent** > **Identity Gateway** > **Next** and use the hints in the following table to create the agent profile:

Field	Description	Example
<b>ID</b>	Set the unique agent profile name PingGateway uses to connect.	ig_agent
<b>Password</b>	Store the password PingGateway uses to connect in the agent profile.  Record the password to use when configuring PingGateway.	A strong password.  The examples in the documentation use password and its base64-encoding cGFzc3dvcmQ= .
<b>Use Secret Store for password</b>	Store the password in a secret and reference the secret by its label.  Follow the steps in Use the secret store for the password after you create the agent profile.	Click to enable

Field	Description	Example
<b>Secret Label Identifier</b>	<p>This field appears when you select <b>Use Secret Store for password</b>.</p> <p>This value represents the <code>identifier</code> part of the secret label for the agent. PingOne Advanced Identity Cloud uses the identifier to generate a secret label in the following format:  <code>am.application.agents.<i>identifier</i>.secret</code> . Learn more in <a href="#">Secret labels</a>.</p> <p>After setting this, add an ESV secret for the password and map the ESV to the secret label.</p>	<code>ig</code>

**IMPORTANT**

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

3. Click **Save Profile** > **Done** to display the new agent profile.
4. (Optional) Add the list of **Redirect URLs** used in PingGateway routes and click **Save** to update the profile.

*Use the secret store for the password*

When you select **Use Secret Store for password** and set a secret label for the agent profile, PingOne Advanced Identity Cloud creates the secret label but the secret isn't yet defined or mapped to the label:

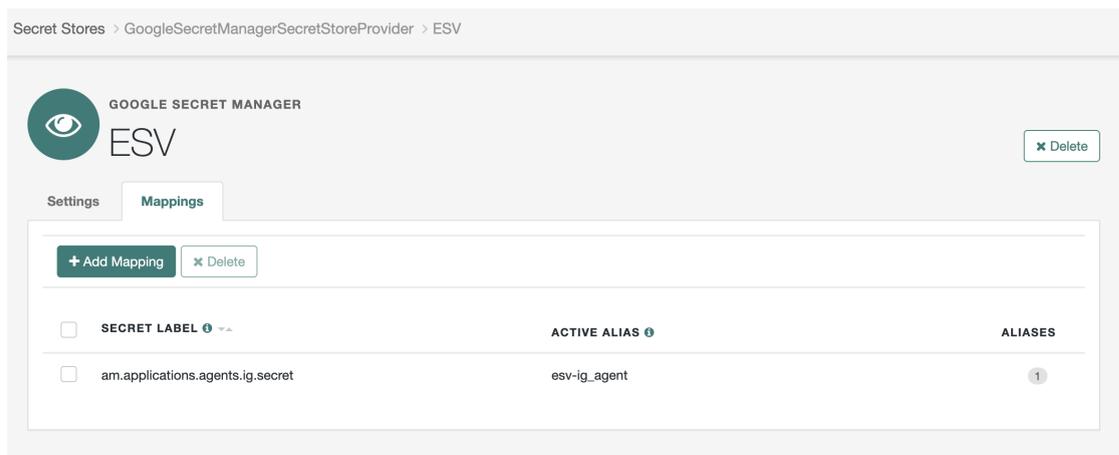
1. Define an ESV secret, such as `esv-ig_agent` , holding the password for PingGateway to connect.

The examples in the documentation use `password` .

Learn how in [creating ESV secrets](#). In production deployments, [restrict access to the password](#) from configuration placeholder and script contexts.

2. Map the ESV to the label created when you set the **Secret Label Identifier**:
  - a. Click  **Native Consoles** > **Access Management** > **Secret Stores** > **ESV** > **Mappings** > **+ Add mappings**.
  - b. In the **Add Mapping** modal, select the label, such as `am.application.agents.ig.secret` , in the **Secret Label** list.
  - c. In the **aliases** field, enter the ESV secret, such as `esv-ig_agent` , and click **Add**.

d. Click **Create** to add the mapping:



Learn more in [Map ESV secrets to secret labels](#).

Note the following points:

- If you update or delete the **Secret Label Identifier**, AM updates or deletes the corresponding mapping for the previous identifier unless another agent shares the mapping.
- When you rotate a secret, update the corresponding mapping.

### *Optional settings*

In the AM admin UI, consider the following additional optional settings for the agent profile under **Applications > Agents > Identity Gateway > agent ID**:

1. To direct login to a custom URL instead of the default AM login page, configure **Login URL Template for CDSSO**.
2. To apply a different introspection scope, click **Token Introspection** and select a scope from the list.
3. Click **Save** to update the profile.

### *Set up a demo user in PingOne Advanced Identity Cloud*

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

- a. Log in to the PingOne Advanced 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

## Recommendations

Use PingGateway with PingOne Advanced Identity Cloud as you would with any other service.

- During updates, individual PingOne Advanced Identity Cloud tenant servers go offline temporarily. PingGateway can receive HTTP 502 Bad Gateway responses for some requests during the update.

In your [ClientHandler](#) and [ReverseProxyHandler](#) configurations, configure PingGateway to retry operations when this occurs:

```
"retries": {  
  "enabled": true,  
  "condition": "${response.status.code == 502}"  
}
```

- Update PingGateway to use the latest version you can to benefit from fixes and improvements.

## Set up AM

This documentation contains procedures for setting up items in AM that you can use with PingGateway. For more information about setting up AM, refer to the [Access Management documentation](#).

### *Authenticate a PingGateway agent to AM*

#### IMPORTANT

##### **From AM 7.3**

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

Authentication chains and modules were deprecated in AM 7. When they are removed in a future release of AM, it will be necessary to configure an appropriate authentication tree when you 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 a PingGateway agent to AM. The tree has the following requirements:

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

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

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

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

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

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

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

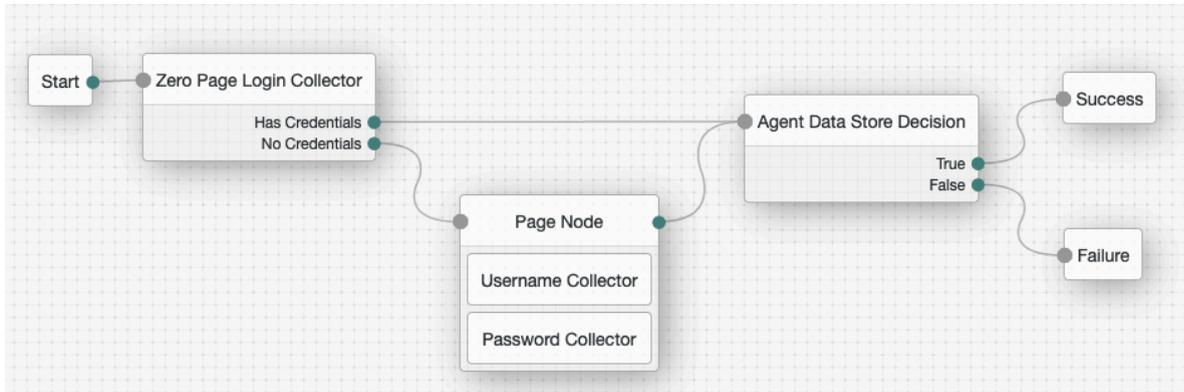
This node is required for compatibility with Java agent and Web agent.

- Page node to collect the agent credentials if they 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 PingGateway 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 a PingGateway agent in AM

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

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

<b>For SSO</b>	<b>For CDSSO</b>
1. <b>Agent ID</b> : ig_agent	
2. <b>Password</b> : password	

3. (Optional - From AM 7.5) Use AM's secret service to manage the agent profile password. If AM finds a matching secret in a secret store, it uses that secret instead of the agent password configured in Step 2.
  - a. In the agent profile page, set a label for the agent password in **Secret Label Identifier**.

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

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

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

- b. Select  **Secret Stores** and configure a secret store.
- c. Map the label to the secret. Learn more from AM's [mapping](#).

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

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

### *Set up a demo user in AM*

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

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

For information about how to manage identities in AM, refer to AM's [Identity stores](#).

### *Find the AM session cookie name*

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

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

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

## Download PingGateway

---

The .zip file unpacks into a `/path/to/identity-gateway-2024.9.0` directory with the following content:

- `bin` : Start and stop executables
- `classes` : Initially empty; used to install patches from ForgeRock support
- `docker/Dockerfile` : Dockerfile and README to build a PingGateway Docker image
- `legal-notices` : Licenses and copyrights
- `lib` : PingGateway and third-party libraries

1. Create a local installation directory for PingGateway. The examples in this section use `/path/to`.

**IMPORTANT**

The installation directory should be a new, empty directory. Installing PingGateway into an existing installation directory can cause errors.

2. Download `PingGateway-2024.9.0.zip` from the [BackStage download site](#), and copy the .zip file to the installation directory:

```
$ cp PingGateway-2024.9.0.zip /path/to/PingGateway-2024.9.0.zip
```

3. Unzip the file:

```
$ unzip PingGateway-2024.9.0.zip
```

The directory `/path/to/identity-gateway-2024.9.0` is created.

## Start and stop PingGateway

### Start PingGateway with default settings

Use the following step to start the instance of PingGateway, specifying the configuration directory where PingGateway looks for configuration files.

1. Start PingGateway:

<b>Linux</b>	<b>Windows</b>
<pre>\$ /path/to/PingGateway-2024.9.0/bin/start.sh</pre>	

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

By default, PingGateway configuration files are located under `$HOME/.openig` (on Windows `%appdata%\OpenIG`). For information about how to use a different location, refer to [Configuration location](#).

2. Check that PingGateway is running in one of the following ways:

- Ping PingGateway at `http://ig.example.com:8080/openig/ping` and make sure an HTTP 200 is returned.
- Display the product version and build information at `http://ig.example.com:8080/openig/api/info`.

## Start PingGateway with custom settings

By default, PingGateway runs on HTTP, on port 8080, from the instance directory `$HOME/.openig`.

To start PingGateway with custom settings, add the configuration file `admin.json` with the following properties, and restart PingGateway:

- `vertx`: Finely tune Vert.x instances.
- `connectors`: Customize server port, TLS, and Vert.x-specific configurations. Each `connectors` object represents the configuration of an individual port.
- `prefix`: Set the instance directory, and therefore, the base of the route for administration requests.

The following example starts PingGateway on non-default ports, and configures Vert.x-specific options for the connection on port 9091:

```
{
  "connectors": [{
    "port": 9090
  },
  {
    "port": 9091,
    "vertx": {
      "maxWebSocketFrameSize": 128000,
      "maxWebSocketMessageSize": 256000,
      "compressionLevel": 4
    }
  }
]
```

For more information, refer to [AdminHttpApplication\( admin.json \)](#).

## Allow startup when there is an existing PID file

By default, if there is an existing PID file during startup the startup fails. Use one of the following ways to allow startup when there is an existing PID file. PingGateway then removes the existing PID file and creates a new one during startup.

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

```
{
  "pidFileMode": "override"
}
```

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

**Linux**

**Windows**

```
$ IG_PID_FILE_MODE=override /path/to/PingGateway-
2024.9.0/bin/start.sh
```

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

**Linux**

**Windows**

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

```
export "IG_OPTS=-Dig.pid.file.mode=override"
```

## Stop PingGateway

Use the `stop.sh` script to stop an instance of PingGateway, specifying the instance directory as an argument. If the instance directory isn't specified, PingGateway uses the default instance directory:

**Linux**

**Windows**

```
$ /path/to/PingGateway-2024.9.0/bin/stop.sh $HOME/.openig
```

# Set up environment variables and system properties

Configure environment variables and system properties as follows:

- By adding environment variables on the command line when you start PingGateway.
- By adding environment variables in `$HOME/.openig/bin/env.sh`, where `$HOME/.openig` is the instance directory. After changing `env.sh`, restart PingGateway to load the new configuration.

## Start PingGateway with a customized router scan interval

By default, PingGateway scans every 10 seconds for changes to the route configuration files. Any changes to the files are automatically loaded into the configuration without restarting PingGateway. For more information about the router scan interval, refer to [Router](#).

The following example overwrites the default value of the Router scan interval to two seconds when you start up PingGateway:

<b>Linux</b>	<b>Windows</b>
<pre>\$ IG_ROUTER_SCAN_INTERVAL='2 seconds' /path/to/PingGateway-2024.9.0/bin/start.sh</pre>	

## Define environment variables for startup, runtime, and stop

PingGateway provides the following environment variables for Java runtime options:

### ***IG\_OPTS***

(Optional) Java runtime options for PingGateway and its startup process, such as JVM memory sizing options.

Include all options that are not shared with the `stop` script.

The following example specifies environment variables in the `env.sh` file to customize JVM options and keys:

<b>Linux</b>	<b>Windows</b>
<pre><i># Specify JVM options</i> JVM_OPTS="-Xms256m -Xmx2048m"</pre>	

```
# Specify the DH key size for stronger ephemeral DH keys,  
and to protect against weak keys  
JSSE_OPTS="-Djdk.tls.ephemeralDHKeySize=2048"  
  
# Wrap them up into the IG_OPTS environment variable  
export IG_OPTS="${IG_OPTS} ${JVM_OPTS} ${JSSE_OPTS}"
```

## JAVA\_OPTS

(Optional) Java runtime options for PingGateway include all options that are shared by the start and stop script.

## Encrypt and share JWT sessions

JwtSession objects store session information in JWT cookies on the user agent. The following sections describe how to set authenticated encryption for JwtSession, using symmetric keys.

Authenticated encryption encrypts data and then signs it with HMAC, in a single step. For more information, refer to [Authenticated Encryption](#)<sup>[2]</sup>. For information about JwtSession, refer to [JwtSession](#).

### Encrypt JWT sessions

This section describes how to set up a keystore with a symmetric key for authenticated encryption of a JWT session.

1. Set up a keystore to contain the encryption key, where the keystore and the key have the password password:
  - a. Locate a directory for secrets, and go to it:

```
$ cd /path/to/secrets
```

- b. Generate the key:

```
$ keytool \  
-genseckey \  
-alias symmetric-key \  
-keystore jwtsessionkeystore.pkcs12 \  
-storepass password \  
-storetype pkcs12 \  
-keyalg HmacSHA512 \  
-keysize 512
```

NOTE

**NOTE**

Because keytool converts all characters in its key aliases to lowercase, use only lowercase in alias definitions of a keystore.

2. Add the following route to PingGateway:

**Linux****Windows**

```
$HOME/.openig/config/routes/jwt-session-encrypt.json
```

```
{
  "name": "jwt-session-encrypt",
  "heap": [{
    "name": "KeyStoreSecretStore-1",
    "type": "KeyStoreSecretStore",
    "config": {
      "file": "/path/to/secrets/jwtsessionkeystore.pkcs12",
      "storeType": "PKCS12",
      "storePasswordSecretId": "keystore.secret.id",
      "secretsProvider": ["SystemAndEnvSecretStore-1"],
      "mappings": [{
        "secretId": "jwtsession.symmetric.secret.id",
        "aliases": ["symmetric-key"]
      }]
    }
  }],
  {
    "name": "SystemAndEnvSecretStore-1",
    "type": "SystemAndEnvSecretStore"
  }
],
  "session": {
    "type": "JwtSession",
    "config": {
      "authenticatedEncryptionSecretId":
"jwtsession.symmetric.secret.id",
      "encryptionMethod": "A256CBC-HS512",
      "secretsProvider": ["KeyStoreSecretStore-1"],
      "cookie": {
        "name": "IG",
        "domain": ".example.com"
      }
    }
  }
},
```

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

Notice the following features of the route:

- The route matches requests to `/jwt-session-encrypt`.
- The `KeyStoreSecretStore` uses the `SystemAndEnvSecretStore` in the heap to manage the store password.
- The `JwtSession` uses the `KeyStoreSecretStore` in the heap to manage the session encryption secret.

3. In the terminal where you will run the `PingGateway` instance, create an environment variable for the value of the keystore password:

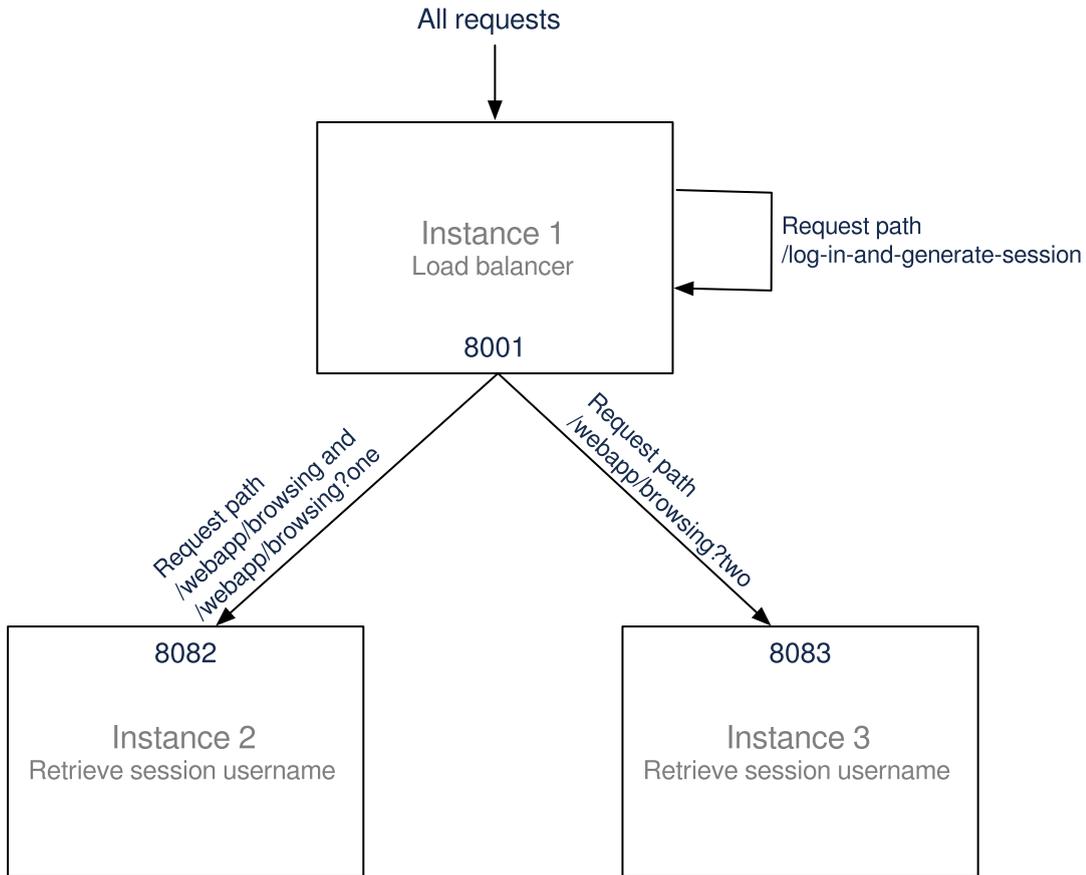
```
$ export KEYSTORE_SECRET_ID='cGFzc3dvcmQ='
```

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

## Share JWT sessions between multiple instances of PingGateway

When a session is shared between multiple instances of `PingGateway`, the instances are able to share the session information for load balancing and failover.

This section gives an example of how to set up a deployment with three instances of `PingGateway` that share a `JwtSession`.



1. Set up a keystore to contain the encryption key, where the keystore and the key have the password password :
  - a. Locate a directory for secrets, and go to it:

```
$ cd /path/to/secrets
```

- b. Generate the key:

```
$ keytool \
  -genseckey \
  -alias symmetric-key \
  -keystore jwtsessionkeystore.pkcs12 \
  -storepass password \
  -storetype pkcs12 \
  -keyalg HmacSHA512 \
  -keysize 512
```

**NOTE** — Because keytool converts all characters in its key aliases to lowercase, use only lowercase in alias definitions of a keystore.

2. Prepare the PingGateway installation:
  - a. Create an installation directory for PingGateway in /path/to .

- b. Download and unzip PingGateway-2024.9.0.zip in /path/to , as described in the [Install](#). The directory /path/to/identity-gateway-2024.9.0 is created.
3. Set up the first instance of PingGateway, which acts as the load balancer:
  - a. Create a configuration directory for the instance and go to it:

```
$ mkdir -p /path/to/config-instance1/config/routes
```

- b. Add the following route:

**Linux**

**Windows**

```
/path/to/config-instance1/config/routes/instance1-loadbalancer.json
```

```
{
  "name": "instance1-loadbalancer",
  "heap": [{
    "name": "KeyStoreSecretStore-1",
    "type": "KeyStoreSecretStore",
    "config": {
      "file":
"/path/to/secrets/jwtsessionkeystore.pkcs12",
      "storeType": "PKCS12",
      "storePasswordSecretId": "keystore.secret.id",
      "secretsProvider": ["SystemAndEnvSecretStore-1"],
      "mappings": [{
        "secretId": "jwtsession.symmetric.secret.id",
        "aliases": ["symmetric-key"]
      }]
    }
  }],
  {
    "name": "SystemAndEnvSecretStore-1",
    "type": "SystemAndEnvSecretStore"
  }
],
  "session": {
    "type": "JwtSession",
    "config": {
      "authenticatedEncryptionSecretId":
"/jwtsession.symmetric.secret.id",
      "encryptionMethod": "A256CBC-HS512",
```

```

    "secretsProvider": [ "KeyStoreSecretStore-1" ],
    "cookie": {
      "name": "IG",
      "domain": ".example.com"
    }
  },
  "handler": {
    "type": "DispatchHandler",
    "config": {
      "bindings": [ {
        "condition": "${find(request.uri.path,
'/webapp/browsing') and (contains(request.uri.query,
'one') or empty(request.uri.query))}",
        "baseURI": "http://ig.example.com:8002",
        "handler": "ReverseProxyHandler"
      }, {
        "condition": "${find(request.uri.path,
'/webapp/browsing') and contains(request.uri.query,
'two')}",
        "baseURI": "http://ig.example.com:8003",
        "handler": "ReverseProxyHandler"
      }, {
        "condition": "${find(request.uri.path, '/log-in-
and-generate-session')}",
        "handler": {
          "type": "Chain",
          "config": {
            "filters": [ {
              "type": "AssignmentFilter",
              "config": {
                "onRequest": [ {
                  "target": "${session.authUsername}",
                  "value": "Sam Carter"
                }
              ]
            }
          ]
        }
      }
    ]
  },
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "headers": {
        "Content-Type": [ "text/html;
charset=UTF-8" ]
      }
    }
  },

```

```
        "entity": "<html><body>Sam Carter logged  
IN. (JWT session generated)</body></html>"  
    }  
  }  
} ]  
}  
,  
"capture": "all"  
}
```

Notice the following features of the route:

- The route has no condition, so it matches all requests.
- When the request matches `/log-in-and-generate-session`, the `DispatchHandler` creates a JWT session, whose `authUsername` attribute contains the name `Sam Carter`.
- When the request matches `/webapp/browsing`, the `DispatchHandler` dispatches the request to instance 2 or instance 3, depending on the rest of the request path.

c. Add the following configuration:

<b>Linux</b>	<b>Windows</b>
<pre>/path/to/config-instance1/config/admin.json</pre>	

```
{  
  "connectors": [{  
    "port": 8001  
  }]  
}
```

d. In the terminal where you will run the `PingGateway` instance, create an environment variable for the value of the keystore password:

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

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

e. Start `PingGateway`:

<b>Linux</b>	<b>Windows</b>
--------------	----------------

```
$ /path/to/PingGateway-2024.9.0/bin/start.sh
/path/to/config-instance1/

...
... started in 1234ms on ports : [8001]
```

4. Set up and start the second instance of PingGateway:

a. Create a configuration directory for the instance:

```
$ mkdir -p /path/to/config-instance2/config/routes
```

b. Add the following route:

<b>Linux</b>	<b>Windows</b>
--------------	----------------

```
/path/to/config-instance2/config/routes/instance2-
retrieve-session-username.json
```

```
{
  "name": "instance2-retrieve-session-username",
  "heap": [{
    "name": "KeyStoreSecretStore-1",
    "type": "KeyStoreSecretStore",
    "config": {
      "file":
"/path/to/secrets/jwtsessionkeystore.pkcs12",
      "storeType": "PKCS12",
      "storePasswordSecretId": "keystore.secret.id",
      "secretsProvider": ["SystemAndEnvSecretStore-1"],
      "mappings": [{
        "secretId": "jwtsession.symmetric.secret.id",
        "aliases": ["symmetric-key"]
      }]
    }
  }],
  {
    "name": "SystemAndEnvSecretStore-1",
    "type": "SystemAndEnvSecretStore"
  }
}
```

```

],
"session": {
  "type": "JwtSession",
  "config": {
    "authenticatedEncryptionSecretId":
"jwtsession.symmetric.secret.id",
    "encryptionMethod": "A256CBC-HS512",
    "secretsProvider": ["KeyStoreSecretStore-1"],
    "cookie": {
      "name": "IG",
      "domain": ".example.com"
    }
  }
},
"handler": {
  "type": "StaticResponseHandler",
  "config": {
    "status": 200,
    "headers": {
      "Content-Type": [ "text/html; charset=UTF-8" ]
    },
    "entity": [
      "<html>",
      "  <body>",
      "    ${session.authUsername!= null?'Hello,
'.concat(session.authUsername).concat('
!'):'Session.authUsername is not defined'}! (instance2)",
      "  </body>",
      "</html>"
    ]
  }
},
"condition": "${find(request.uri.path,
'/webapp/browsing')}",
"capture": "all"
}

```

Notice the following features of the route compared to the route for instance 1:

- The route matches the condition `/webapp/browsing`. When a request matches `/webapp/browsing`, the `DispatchHandler` dispatches it to instance 2.
- The `StaticResponseHandler` displays information from the session context.

c. Add the following configuration:

Linux

Windows

```
/path/to/config-instance2/config/admin.json
```

```
{  
  "connectors": [{  
    "port": 8002  
  }]  
}
```

- d. In the terminal where you will run the PingGateway instance, create an environment variable for the value of the keystore password:

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

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

- e. Start PingGateway:

Linux

Windows

```
$ /path/to/PingGateway-2024.9.0/bin/start.sh  
/path/to/config-instance2/  
  
...  
... started in 1234ms on ports : [8002]
```

5. Set up and start the third instance of PingGateway:

- a. Create a configuration directory:

```
$ mkdir -p /path/to/config-instance3/config/routes
```

- b. Add the following route:

Linux

Windows

```
/path/to/config-instance3/config/routes/instance3-  
retrieve-session-username.json
```

```

{
  "name": "instance3-retrieve-session-username",
  "heap": [{
    "name": "KeyStoreSecretStore-1",
    "type": "KeyStoreSecretStore",
    "config": {
      "file":
"/path/to/secrets/jwtsessionkeystore.pkcs12",
      "storeType": "PKCS12",
      "storePasswordSecretId": "keystore.secret.id",
      "secretsProvider": ["SystemAndEnvSecretStore-1"],
      "mappings": [{
        "secretId": "jwtsession.symmetric.secret.id",
        "aliases": ["symmetric-key"]
      }]
    }
  ]},
  {
    "name": "SystemAndEnvSecretStore-1",
    "type": "SystemAndEnvSecretStore"
  }
],
"session": {
  "type": "JwtSession",
  "config": {
    "authenticatedEncryptionSecretId":
"jwtsession.symmetric.secret.id",
    "encryptionMethod": "A256CBC-HS512",
    "secretsProvider": ["KeyStoreSecretStore-1"],
    "cookie": {
      "name": "IG",
      "domain": ".example.com"
    }
  }
},
"handler": {
  "type": "StaticResponseHandler",
  "config": {
    "status": 200,
    "headers": {
      "Content-Type": [ "text/html; charset=UTF-8" ]
    },
    "entity": [
      "<html>",
      "  <body>"
    ]
  }
}

```

```

        "    ${session.authUsername!= null?'Hello,
'.concat(session.authUsername).concat('
!'): 'Session.authUsername is not defined'}! (instance3)",
        "    </body>",
        "</html>"
    ]
    },
    "condition": "${find(request.uri.path,
'/webapp/browsing')}",
    "capture": "all"
}

```

Notice that the route is the same as that for instance 2, apart from the text in the entity of the StaticResponseHandler.

c. Add the following configuration:

Linux	Windows
<div style="border: 1px solid #ccc; padding: 10px; width: fit-content; margin: 0 auto;">             /path/to/config-instance3/config/admin.json           </div>	

```

{
  "connectors": [{
    "port": 8003
  }]
}

```

d. In the terminal where you will run the PingGateway instance, create an environment variable for the value of the keystore password:

```

$ export KEYSTORE_SECRET_ID='cGFzc3dvcmQ='

```

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

e. Start PingGateway:

Linux	Windows
<div style="border: 1px solid #ccc; padding: 10px; width: fit-content; margin: 0 auto;"> <pre> \$ /path/to/PingGateway-2024.9.0/bin/start.sh /path/to/config-instance3/ </pre> </div>	

```
...
... started in 1234ms on ports : [8003]
```

## 6. Test the setup:

- a. Access instance 1, to generate a session:

```
$ curl -v http://ig.example.com:8001/log-in-and-generate-session

GET /log-in-and-generate-session HTTP/1.1
...

HTTP/1.1 200 OK
Content-Length: 84
Set-Cookie: IG=eyJ...HyI; Path=/; Domain=.example.com; HttpOnly
...
Sam Carter logged IN. (JWT session generated)
```

- b. Using the JWT cookie returned in the previous step, access instance 2:

```
$ curl -v http://ig.example.com:8001/webapp/browsing\?one
--header "cookie:IG=eyJ...HyI"

GET /webapp/browsing?one HTTP/1.1
...
cookie: IG=eyJ...HyI
...
HTTP/1.1 200 OK
...
Hello, Sam Carter !! (instance2)
```

Note that instance 2 can access the session info.

- c. Using the JWT cookie again, access instance 3:

```
$ curl -v http://ig.example.com:8001/webapp/browsing\?two
--header "cookie:IG=eyJ...HyI"

GET /webapp/browsing?two HTTP/1.1
...
cookie: IG=eyJ...HyI
...
HTTP/1.1 200 OK
```

```
...  
Hello, Sam Carter !! (instance3)
```

Note that instance 3 can access the session info.

## Prepare for load balancing and failover

---

For high scale or highly available deployments, consider using a pool of PingGateway servers with nearly identical configurations. Load balance requests across the pool to handle more load. Route around any servers that become unavailable.

### Manage state information

Before spreading requests across multiple servers, decide how to manage state information. PingGateway manages state information in the following ways:

#### *Stores state information in a context*

By using filters that can store information in the context. Most filters depend on information in the request, response, or context, some of which is first stored by PingGateway. For a summary of filters that can populate a context, refer to [Summary of contexts](#).

By using a handler such as the [ScriptableHandler](#) that can store state information in the context. Most handlers depend on information in the context, some of which is first stored by PingGateway.

#### *Retrieves state information to local memory*

By using filters and handlers that depend on the configuration of the local file system, such as the following filters:

- [FileAttributesFilter](#)
- [ScriptableFilter](#)
- [ScriptableHandler](#)
- [SqlAttributesFilter](#)

When a server becomes unavailable, state information held in local memory is lost. To prevent data loss when a server becomes unavailable, set up failover. Server failover should be transparent to client applications.

### Prepare stateless sessions

For example configurations, refer to [Encrypt and share JWT sessions](#).

## *JwtSession*

Manage stateless sessions through [JwtSession](#). Session content is stored on a JWT cookie on the user agent.

So that any server can read or update a JWT cookie from any other server in the same cookie domain, encrypt JWT sessions and share keys and secret across all PingGateway configurations.

[Encrypt JWT sessions](#). The maximum size of the JWT session cookie is 4 KBytes, as defined by the browser. If the cookie exceeds this size, PingGateway automatically splits it into multiple cookies.

## *Session stickiness*

Session stickiness helps to ensure that a client request goes to the server holding the original session data.

If data attached to a context must be stored on the server-side, configure session stickiness so that the load balancer sends all requests from the same client session to the same server.

For an example configuration, refer to [Share JWT sessions between multiple instances of PingGateway](#).

## SAML in deployments with multiple instances of PingGateway

PingGateway uses AM federation libraries to implement SAML. When PingGateway acts as a SAML service provider, some internal state information is maintained in the fedlet instead of the session cookie. In deployments that use multiple instances of PingGateway as a SAML service provider, set up sticky sessions so that requests go to the server that started the SAML interaction.

For information, refer to [Session state considerations](#) in *AM's SAML v2.0 guide*.

## Secure connections

---

PingGateway is often deployed to replay credentials or other security information. In a real world deployment, this information must be communicated over a secure connection using HTTPS, meaning HTTP over encrypted Transport Layer Security (TLS). Never send real credentials, bearer tokens, or other security information unprotected over HTTP.

Learn about how to use well-known CA-signed certificates from the documentation for the Java Virtual Machine (JVM).

After installing certificates for client-server trust, consider which cipher suites to use. PingGateway inherits the list of cipher suites from the underlying Java environment.

PingGateway uses the JSSE to secure connections. You can set security and system properties to configure the JSSE. For a list of properties to customize the JSSE in Oracle Java, refer to the *Customization* section of the [JSSE Reference guide](#).

## Configure PingGateway for TLS (client-side)

When PingGateway sends requests over HTTP to a proxied application, or requests services from a third-party application, PingGateway is acting as a client of the application, and the application is acting as a server. PingGateway is *client-side*.

When PingGateway sends requests securely over HTTPS, PingGateway must be able to trust the server. By default, PingGateway uses the Java environment truststore to trust server certificates. The Java environment truststore includes public key signing certificates from many well-known Certificate Authorities (CAs).

When servers present certificates signed by trusted CAs, then PingGateway can send requests over HTTPS to those servers, without any configuration to set up the HTTPS client connection. When server certificates are self-signed or signed by a CA whose certificate is not automatically trusted, the following objects can be required to configure the connection:

- [KeyStoreSecretStore](#), to manage a secret store for cryptographic keys and certificates, based on a standard Java keystore.
- [SecretsTrustManager](#), to manage trust material that verifies the credentials presented by a peer.
- (Optional) [SecretsKeyManager](#), to manage keys that authenticate a TLS connection to a peer.
- ClientHandler and ReverseProxyHandler reference to [ClientTlsOptions](#), for connecting to TLS-protected endpoints.

The following procedure describes how to set up PingGateway for HTTPS (client-side), when server certificates are self-signed or signed by untrusted CAs.

### ***Set up PingGateway for HTTPS (client-side) for untrusted servers***

1. Locate or set up the following directories:
  - Directory containing the sample application .jar: `sampleapp_install_dir`
  - Directory to store the sample application certificate and PingGateway keystore: `/path/to/secrets`
2. Extract the public certificate from the sample application:

```
$ cd /path/to/secrets
```

```
$ jar --verbose --extract \  
--file sampleapp_install_dir/PingGateway-sample-application-  
2024.9.0-jar-with-dependencies.jar tls/sampleapp-cert.pem  
  
inflated: tls/sampleapp-cert.pem
```

The file `/path/to/secrets/tls/sampleapp-cert.pem` is created.

3. Import the certificate into the PingGateway keystore, and answer `yes` to trust the certificate:

```
$ keytool -importcert \  
-alias ig-sampleapp \  
-file tls/sampleapp-cert.pem \  
-keystore reverseproxy-truststore.p12 \  
-storetype pkcs12 \  
-storepass password  
  
...  
Trust this certificate? [no]: yes  
  
Certificate was added to keystore
```

#### NOTE

Because `keytool` converts all characters in its key aliases to lowercase, use only lowercase in alias definitions of a keystore.

4. List the keys in the PingGateway keystore to make sure that a key with the alias `ig-sampleapp` is present:

```
$ keytool -list \  
-v \  
-keystore /path/to/secrets/reverseproxy-truststore.p12 \  
-storetype pkcs12 \  
-storepass password  
  
Keystore type: PKCS12  
Keystore provider: SUN  
Your keystore contains 1 entry  
Alias name: ig-sampleapp  
...  
...
```

5. 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"
}
```

6. Add the following route to PingGateway:

**Linux****Windows**

```
$HOME/.openig/config/routes/client-side-https.json
```

```
{
  "name": "client-side-https",
  "condition": "${find(request.uri.path, '/home/client-side-
https')}",
  "baseURI": "https://app.example.com:8444",
  "heap": [
    {
      "name": "Base64EncodedSecretStore-1",
      "type": "Base64EncodedSecretStore",
      "config": {
        "secrets": {
          "keystore.secret.id": "cGFzc3dvcmQ="
        }
      }
    },
    {
      "name": "KeyStoreSecretStore-1",
      "type": "KeyStoreSecretStore",
      "config": {
        "file": "/path/to/secrets/reverseproxy-
truststore.p12",
        "storeType": "PKCS12",

```

```

    "storePasswordSecretId": "keystore.secret.id",
    "secretsProvider": "Base64EncodedSecretStore-1",
    "mappings": [
      {
        "secretId": "trust.manager.secret.id",
        "aliases": [ "ig-sampleapp" ]
      }
    ]
  },
  {
    "name": "SecretsTrustManager-1",
    "type": "SecretsTrustManager",
    "config": {
      "verificationSecretId": "trust.manager.secret.id",
      "secretsProvider": "KeyStoreSecretStore-1"
    }
  },
  {
    "name": "ReverseProxyHandler-1",
    "type": "ReverseProxyHandler",
    "config": {
      "tls": {
        "type": "ClientTlsOptions",
        "config": {
          "trustManager": "SecretsTrustManager-1"
        }
      },
      "hostnameVerifier": "ALLOW_ALL"
    },
    "capture": "all"
  }
],
"handler": "ReverseProxyHandler-1"
}

```

Notice the following features of the route:

- The route matches requests to `/home/client-side-https`.
- The `baseURI` changes the request URI to point to the HTTPS port for the sample application.
- The `Base64EncodedSecretStore` provides the keystore password.
- The `SecretsTrustManager` points to the secret bound to the sample application certificate, coming from the `KeyStoreSecretStore`.

- The `KeyStoreSecretStore` contains the sample application certificate to validate the TLS connection. The password to access the keystore is provided by the `SystemAndEnvSecretStore`.
- The `ReverseProxyHandler` uses the `SecretsTrustManager` for the connection to TLS-protected endpoints. All hostnames are allowed.

## 7. Test the setup:

- Start the sample application

```
$ java -jar sampleapp_install_dir/PingGateway-sample-application-2024.9.0-jar-with-dependencies.jar
```

- Go to <http://ig.example.com:8080/home/client-side-https>.

The request is proxied transparently to the sample application, on the TLS port 8444 .

- Check the route log for a line like this:

```
GET https://app.example.com:8444/home/client-side-https
```

## Configure PingGateway for TLS (server-side)

When PingGateway is *server-side*, applications send requests to PingGateway or request services from PingGateway. PingGateway is acting as a server of the application, and the application is acting as a client.

To run PingGateway as a server over TLS:

- In the `admin.json` heap, add a `SecretsKeyManager` to authenticate PingGateway to the client. Key material is a private key and its certificate for PingGateway.
- In `admin.json`, update the `connectors` list to include a connector for the HTTPS port. The connector `tls` property must refer to a `ServerTlsOptions`.
- The `ServerTlsOptions` must configure `keyManager` to refer to the `SecretsKeyManager`.

The following example connector includes a `ServerTlsOptions` that refers to a `SecretsKeyManager` in the heap:

```
"connectors": [
  {
    "port": 8443,
    "tls": {
      "type": "ServerTlsOptions",
      "config": {
```

```
        "keyManager" : "SecretsKeyManager-1"
      }
    }
  }
]
```

Learn more from [About keys and certificates](#).

### *Serve one certificate for TLS connections to all server names*

This example uses PEM files and a PKCS#12 keystore for self-signed certificates, but you can adapt it to use official (non self-signed) keys and certificates.

Before you start, install PingGateway, as described in the [Install](#).

1. Locate a directory for the secrets, for example, `/path/to/secrets`.
2. Create self-signed keys in one of the following ways. If you have your own keys, use them and skip this step.

#### ▼ [Use your own keys](#)

If you have your own keys, use them and skip this step.

#### ▼ [Set up a self-signed certificate in a \(PKCS#12\) keystore](#)

1. Create the keystore, replacing `/path/to/secrets` with your path:

```
$ keytool \  
-genkey \  
-alias https-connector-key \  
-keyalg RSA \  
-keystore /path/to/secrets/keystore.pkcs12 \  
-storepass password \  
-keypass password \  
-dname "CN=ig.example.com,O=Example Corp,C=FR"
```

#### NOTE

Because keytool converts all characters in its key aliases to lowercase, use only lowercase in alias definitions of a keystore.

2. In the secrets directory, add a file called `keystore.pass`, containing the keystore password `password`:

```
$ cd /path/to/secrets/  
$ echo -n 'password' > keystore.pass
```

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

▼ [Set up self-signed certificate stored in PEM file](#)

a. Locate a directory for secrets, and go to it:

```
$ cd /path/to/secrets
```

b. Create the following secret key and certificate pair as PEM files:

```
$ 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 ig.example.com-key.pem \  
-out ig.example.com-certificate.pem
```

Two PEM files are created, one for the secret key, and another for the associated certificate.

c. Map the key and certificate to the same secret ID in PingGateway:

```
$ cat ig.example.com-key.pem ig.example.com-  
certificate.pem > key.manager.secret.id.pem
```

3. Set up TLS on PingGateway in one of the following ways:

▼ [Keys stored in a \(PKCS#12\) keystore](#)

Add the following file to PingGateway, replacing `/path/to/secrets` with your path:

**Linux**

**Windows**

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

```
{  
  "connectors": [  
    {  
      "port": 8080
```

```

    },
    {
      "port": 8443,
      "tls": "ServerTlsOptions-1"
    }
  ],
  "heap": [
    {
      "name": "ServerTlsOptions-1",
      "type": "ServerTlsOptions",
      "config": {
        "keyManager": {
          "type": "SecretsKeyManager",
          "config": {
            "signingSecretId": "key.manager.secret.id",
            "secretsProvider": "ServerIdentityStore"
          }
        }
      }
    },
    {
      "type": "FileSystemSecretStore",
      "name": "SecretsPasswords",
      "config": {
        "directory": "/path/to/secrets",
        "format": "PLAIN"
      }
    },
    {
      "name": "ServerIdentityStore",
      "type": "KeyStoreSecretStore",
      "config": {
        "file": "/path/to/secrets/IG-keystore",
        "storePasswordSecretId": "keystore.pass",
        "secretsProvider": "SecretsPasswords",
        "mappings": [
          {
            "secretId": "key.manager.secret.id",
            "aliases": ["https-connector-key"]
          }
        ]
      }
    }
  ]
}

```

Notice the following features of the file:

- PingGateway starts on port 8080, and on 8443 over TLS.
- PingGateway's private keys for TLS are managed by the SecretsKeyManager, whose ServerIdentityStore references a KeyStoreSecretStore.
- The KeyStoreSecretStore maps the keystore alias to the secret ID for retrieving the server keys (private key + certificate).
- The password of the KeyStoreSecretStore is provided by the FileSystemSecretStore.

▼ [Keys stored in PEM file](#)

Add the following file to PingGateway, replacing /path/to/secrets with your path:

<b>Linux</b>	<b>Windows</b>
<pre>\$HOME/.openig/config/admin.json</pre>	

```
{
  "connectors": [
    {
      "port": 8080
    },
    {
      "port": 8443,
      "tls": "ServerTlsOptions-1"
    }
  ],
  "heap": [
    {
      "name": "ServerTlsOptions-1",
      "type": "ServerTlsOptions",
      "config": {
        "keyManager": {
          "type": "SecretsKeyManager",
          "config": {
            "signingSecretId": "key.manager.secret.id",
            "secretsProvider": "ServerIdentityStore"
          }
        }
      }
    }
  ],
  {
```

```

"name": "ServerIdentityStore",
"type": "FileSystemSecretStore",
"config": {
  "format": "PLAIN",
  "directory": "/path/to/secrets",
  "suffix": ".pem",
  "mappings": [{
    "secretId": "key.manager.secret.id",
    "format": {
      "type": "PemPropertyFormat"
    }
  }]
}
]
}

```

Notice how this file differs to that for the keystore-based approach:

- The ServerIdentityStore is a FileSystemSecretStore.
- The FileSystemSecretStore reads the keys that are stored as file in the PEM standard format.

#### 4. Start PingGateway:

Linux	Windows
<pre>\$ /path/to/PingGateway-2024.9.0/bin/start.sh</pre>	

By default, PingGateway configuration files are located under `$HOME/.openig` (on Windows `%appdata%\OpenIG`). For information about how to use a different location, refer to [Configuration location](#).

#### 5. Test the connection in one of the following ways:

- Ping PingGateway and make sure an HTTP 200 is returned:

```
$ curl -v --cacert /path/to/secrets/ig.example.com-
certificate.pem \
https://ig.example.com:8443/openig/ping
```

- Display the product version and build information:

```
$ curl --cacert /path/to/secrets/ig.example.com-
certificate.pem \
```

```
https://ig.example.com:8443/openig/api/info
```

## *Use Server Name Indication (SNI) to serve different certificates for TLS connections to different server names*

This example uses PEM files for self-signed certificates, but you can adapt it to use official (non self-signed) keys and certificates.

Before you start, install PingGateway, as described in the [Install](#).

1. Locate a directory for secrets, for example, `/path/to/secrets`, and go to it.

```
$ cd /path/to/secrets
```

2. Create the following secret key and certificate pair as PEM files:

- a. For `ig.example.com`:

- i. Create a key and certificate:

```
$ 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 ig.example.com-key.pem \  
-out ig.example.com-certificate.pem
```

Two PEM files are created, one for the secret key, and another for the associated certificate.

- ii. Map the key and certificate to the same secret ID in PingGateway:

```
$ cat ig.example.com-key.pem ig.example.com-  
certificate.pem > key.manager.secret.id.pem
```

- b. For servers grouped by a wildcard:

- i. Create a key and certificate:

```
$ openssl req \  
-newkey rsa:2048 \  
-new \  
-nodes \  
-
```

```
-x509 \  
-days 3650 \  
-subj  
"/CN=*.example.com/OU=example/O=com/L=fr/ST=fr/C=fr" \  
-keyout wildcard.example.com-key.pem \  
-out wildcard.example.com-certificate.pem
```

ii. Map the key and certificate to the same secret ID in PingGateway:

```
$ cat wildcard.example.com-key.pem  
wildcard.example.com-certificate.pem >  
wildcard.secret.id.pem
```

c. For other, unmapped servers

i. Create a key and certificate:

```
$ openssl req \  
-newkey rsa:2048 \  
-new \  
-nodes \  
-x509 \  
-days 3650 \  
-subj  
"/CN=un.mapped.com/OU=example/O=com/L=fr/ST=fr/C=fr" \  
-keyout default.example.com-key.pem \  
-out default.example.com-certificate.pem
```

ii. Map the key and certificate to the same secret ID in PingGateway:

```
$ cat default.example.com-key.pem default.example.com-  
certificate.pem > default.secret.id.pem
```

3. Add the following file to PingGateway, replacing `/path/to/secrets` with your path, and then restart PingGateway:

**Linux**

**Windows**

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

```
{  
  "connectors": [  
    {  
      "port": 8080
```

```

    },
    {
      "port": 8443,
      "tls": "ServerTlsOptions-1"
    }
  ],
  "heap": [
    {
      "name": "ServerTlsOptions-1",
      "type": "ServerTlsOptions",
      "config": {
        "sni": {
          "serverNames": {
            "ig.example.com": "key.manager.secret.id",
            "*.example.com": "wildcard.secret.id"
          },
          "defaultSecretId" : "default.secret.id",
          "secretsProvider": "ServerIdentityStore"
        }
      }
    },
    {
      "name": "ServerIdentityStore",
      "type": "FileSystemSecretStore",
      "config": {
        "format": "PLAIN",
        "directory": "path/to/secrets",
        "suffix": ".pem",
        "mappings": [
          {
            "secretId": "key.manager.secret.id",
            "format": {
              "type": "PemPropertyFormat"
            }
          },
          {
            "secretId": "wildcard.secret.id",
            "format": {
              "type": "PemPropertyFormat"
            }
          },
          {
            "secretId": "default.secret.id",
            "format": {
              "type": "PemPropertyFormat"
            }
          }
        ]
      }
    }
  ]
}

```

```

    }
  }
]
}
]
}

```

Notice the following features of the file:

- The `ServerTlsOptions` object maps two servers to secret IDs, and includes a default secret ID
- The secret IDs correspond to the secret IDs in the `FileSystemSecretStore`, and the PEM files generated in an earlier step.

4. Run the following commands to request TLS connections to different servers, using different certificates:

- a. Connect to `ig.example.com`, and note that the certificate subject corresponds to the certificate created for `ig.example.com`:

```

$ openssl s_client -connect localhost:8443 -servername
ig.example.com

...
Server certificate
-----BEGIN CERTIFICATE-----
MII...dZC
-----END CERTIFICATE-----
subject=/CN=ig.example.com/OU=example/O=com/L=fr/ST=fr/C=f
r
issuer=/CN=ig.example.com/OU=example/O=com/L=fr/ST=fr/C=fr

```

- b. Connect to `other.example.com`, and note that the certificate subject corresponds to the certificate created with the wildcard, `*.example.com`:

```

$ openssl s_client -connect localhost:8443 -servername
other.example.com

...
Server certificate
-----BEGIN CERTIFICATE-----
MII...fY=
-----END CERTIFICATE-----
subject=/CN=*.example.com/OU=example/O=com/L=fr/ST=fr/C=fr
issuer=/CN=*.example.com/OU=example/O=com/L=fr/ST=fr/C=fr

```

- c. Connect to `unmapped.site.com`, and note that the certificate subject corresponds to the certificate created for the default secret ID:

```
$ openssl s_client -connect localhost:8443 -servername
unmapped.site.com

...
Server certificate
-----BEGIN CERTIFICATE-----
MII..rON
-----END CERTIFICATE-----
subject=/CN=un.mapped.com/OU=example/O=com/L=fr/ST=fr/C=fr
issuer=/CN=un.mapped.com/OU=example/O=com/L=fr/ST=fr/C=fr
```

## Configure PingGateway for mutual TLS (server-side)

When PingGateway is *server-side*, applications send requests to PingGateway or request services from PingGateway. PingGateway is acting as a server of the application, and the application is acting as a client.

To run PingGateway as a server for mutual TLS:

- Using information from [Configure PingGateway for TLS \(server-side\)](#), configure PingGateway for TLS connections.
- In `admin.json`, add a [SecretsTrustManager](#) to verify the credentials presented by the client:
  - The trust material is a public key certificate for a client or certificate authority.
  - The `SecretsTrustManager` must refer to a secret in a secrets store, where the secret is mapped to the certificate.
  - `ServerTlsOptions` must configure `trustManager` to refer to that `SecretsTrustManager` and `clientAuth` to require or request the client to authenticate.

The following example is used in [Mutual TLS](#).

Linux	Windows
<pre>\$HOME/.openig/config/admin.json</pre>	

```
{
  "mode": "DEVELOPMENT",
  "properties": {
```

```

    "ig_keystore_directory": "/path/to/ig/secrets",
    "oauth2_client_keystore_directory":
"/path/to/client/secrets"
  },
  "connectors": [
    {
      "port": 8080
    },
    {
      "port": 8443,
      "tls": {
        "type": "ServerTlsOptions",
        "config": {
          "alpn": {
            "enabled": true
          },
          "clientAuth": "REQUEST",
          "keyManager": "SecretsKeyManager-1",
          "trustManager": "SecretsTrustManager-1"
        }
      }
    }
  ],
  "heap": [
    {
      "name": "SecretsPasswords",
      "type": "FileSystemSecretStore",
      "config": {
        "directory": "&{ig_keystore_directory}",
        "format": "PLAIN"
      }
    },
    {
      "name": "SecretsKeyManager-1",
      "type": "SecretsKeyManager",
      "config": {
        "signingSecretId": "key.manager.secret.id",
        "secretsProvider": "ServerIdentityStore"
      }
    },
    {
      "name": "SecretsTrustManager-1",
      "type": "SecretsTrustManager",
      "config": {
        "verificationSecretId": "trust.manager.secret.id",

```

```

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

```

Was this helpful?  