# Oracle® Fusion Middleware

API Gateway OAuth User Guide
11g Release 2 (11.1.2.2.0)

August 2013

Oracle API Gateway OAuth User Guide, 11g Release 2 (11.1.2.2.0)



28 August 2013

# Contents

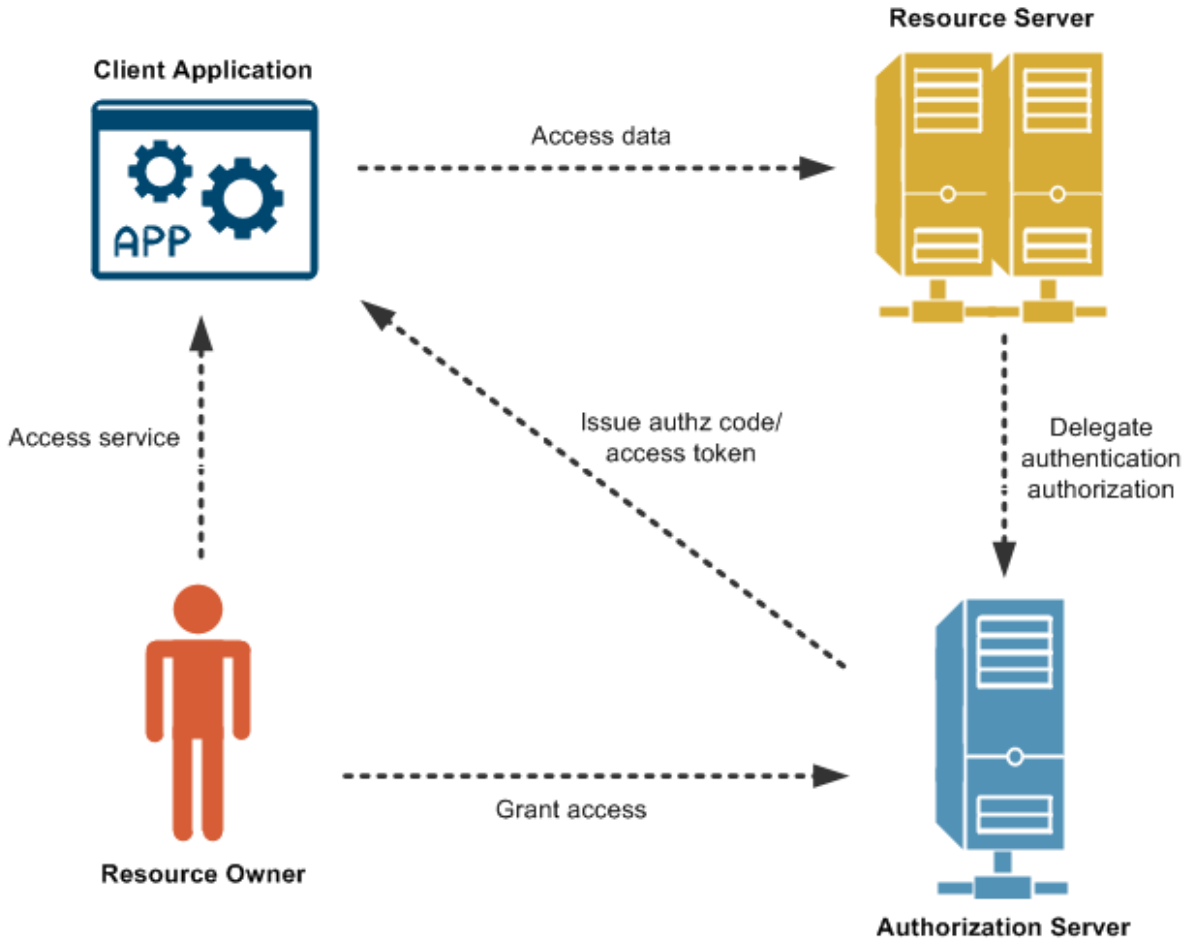
# Introduction to API Gateway OAuth 2.0

## Overview

OAuth is an open standard for authorization that enables client applications to access server resources on behalf of a specific Resource Owner. OAuth also enables Resource Owners (end users) to authorize limited third-party access to their server resources without sharing their credentials. For example, a Gmail user could allow LinkedIn or Flickr to have access to their list of contacts without sharing their Gmail username and password.

The Oracle API Gateway can be used as an Authorization Server and as a Resource Server. An Authorization Server issues tokens to client applications on behalf of a Resource Owner for use in authenticating subsequent API calls to the Resource Server. The Resource Server hosts the protected resources, and can accept or respond to protected resource requests using access tokens.

The following diagram shows the main OAuth components:



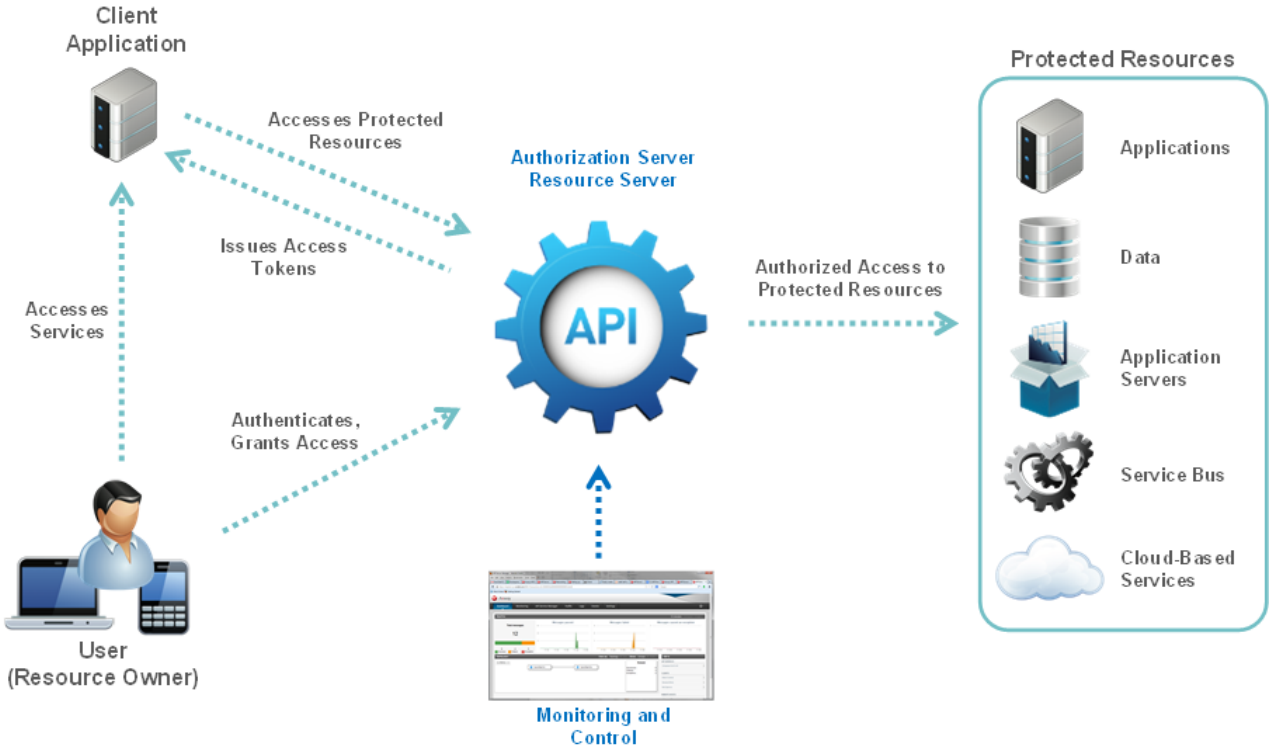

This guide assumes that you are already familiar with the terms and concepts of *The OAuth 2.0 Authorization Framework* specification:
http://tools.ietf.org/html/rfc6749

## OAuth 2.0 Concepts

The API Gateway uses the following definitions of basic OAuth 2.0 terms:

- **Resource Owner**:
  An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end user.
- **Resource Server**:
  The server hosting the protected resources, and which is capable of accepting and responding to protected resource requests using access tokens. In this case, the API Gateway acts as a gateway implementing the Resource Server that sits in front of the protected resources.
- **Client Application**:
  A client application making protected requests on behalf of the resource owner and with its authorization.
- **Authorization Server**:
  The server issuing access tokens to the client application after successfully authenticating the Resource Owner and obtaining authorization. In this case, the API Gateway acts both as the Authorization Server and as the Resource Server.
- **Scope**:
  Used to control access to the Resource Owner's data when requested by a client application. You can validate the OAuth scopes in the incoming message against the scopes registered in the API Gateway. An example scope is `https://localhost:8090/auth/userinfo.email`.

The following diagram shows the roles of the API Gateway as an OAuth 2.0 Resource Server and Authorization Server:




## API Gateway OAuth Features

The API Gateway ships with the following features to support OAuth 2.0:

- Web-based client application registration
- Generation of authorization codes, access tokens, and refresh tokens
- Support for the following OAuth flows:
  - Authorization Code
  - Implicit Grant
  - Resource Owner Password Credentials
  - Client Credentials
  - JWT
  - Refresh Token
  - Revoke Token
  - Token Information Service
- Sample client applications for all supported flows

These API Gateway features are explained in the topics that follow.

## OAuth 2.0 Authentication Flows

The API Gateway supports the following authentication flows:

- **OAuth 2.0 Authorization Code Grant (Web Server)**:
  The Web server authentication flow is used by applications that are hosted on a secure server. A critical aspect of the Web server flow is that the server must be able to protect the issued client application's secret.
- **OAuth 2.0 Implicit Grant (User-Agent)**:
  The user-agent authentication flow is used by client applications residing in the user's device. This could be implemented in a browser using a scripting language such as JavaScript or Flash. These client applications cannot keep the client application secret confidential.
- **OAuth 2.0 Resource Owner Password Credentials**:
  This username-password authentication flow can be used when the client application already has the Resource Owner's credentials.
- **OAuth 2.0 Client Credentials**:
  This username-password flow is used when the client application needs to directly access its own resources on the Resource Server. Only the client application's credentials are used in this flow. The Resource Owner's credentials are not required.
- **OAuth 2.0 JWT**:
  This flow is similar to OAuth 2.0 Client Credentials. A JSON Web Token (JWT) is a JSON-based security token encoding that enables identity and security information to be shared across security domains.
- **OAuth 2.0 Refresh Token**:
  After the client application has been authorized for access, it can use a refresh token to get a new access token. This is only done after the consumer already has received an access token using the Authorization Code Grant or Resource Owner Password Credentials flow.
- **OAuth 2.0 Revoke Token**:
  A revoke token request causes the removal of the client application permissions associated with the particular token to access the end-user's protected resources.
- **OAuth 2.0 Token Information Service**:
  The OAuth Token Info service responds to requests for information on a specified OAuth 2.0 access token.

## Further Information

For more details on the API Gateway OAuth 2.0 support, see the following topics:

- *Setting up API Gateway OAuth 2.0*
- *Managing OAuth 2.0 Applications*
- *API Gateway OAuth 2.0 Authentication Flows*

For more details on OAuth 2.0, see *The OAuth 2.0 Authorization Framework*:
http://tools.ietf.org/html/rfc6749

# Setting up API Gateway OAuth 2.0

## Overview

This chapter describes how to configure the OAuth 2.0 support provided with the API Gateway. It describes how to enable the OAuth 2.0 endpoints used to manage client applications, and how to import the pre-registered examples provided with the API Gateway. It how explains how to migrate existing OAuth 2.0 applications.

## Enabling OAuth 2.0 Management

The API Gateway provides the following endpoints used to manage OAuth 2.0 client applications:

| **Description** | **URL** |
|---|---|
| Authorization Endpoint (REST API) | `https://GATEWAY:8089/api/oauth/authorize` |
| Token Endpoint (REST API) | `https://GATEWAY:8089/api/oauth/token` |
| Token Info Endpoint (REST API) | `https://GATEWAY:8089/api/oauth/tokeninfo` |
| Revoke Endpoint (REST API) | `https://GATEWAY:8089/api/oauth/revoke` |
| Oracle Client Application Registry (HTML Interface) | `https://GATEWAY:8089` |
| Oracle Client Application Registry (REST API) | `https://GATEWAY:8089/api/kps/ClientApplicationRegistry` |

In this table, *GATEWAY* refers to the machine on which the API Gateway is installed.

### Important

You must first enable the OAuth listener port in the API Gateway before these endpoints are available.

**Enabling OAuth endpoints**
To enable the OAuth management endpoints on your API Gateway, perform the following steps:

1. In the Policy Studio tree, select **Listeners** -> **API Gateway** -> **OAuth 2.0 Services** -> **Ports**.
2. Right-click the **OAuth 2.0 Interface** in the panel on the right, and select **Edit**.
3. Select **Enable Interface** in the dialog.
4. Click the **Deploy** button in the toolbar.
5. Enter a description and click **Finish**.

### Note

On Linux-based systems, such as Oracle Enterprise Linux, you must open the firewall to allow external access to port `8089`. If you need to change the port number, set the value of the `env.PORT.OAUTH2.SERVICES` environment variable. For details on setting external environment vari-

ables for API Gateway instances, see the *API Gateway Deployment and Promotion Guide*.

## Importing Client Applications

The API Gateway ships with a number of pre-registered sample client applications. This section explains how to import these applications into the Client Application Registry.

### Note

The sample client applications are for demonstration purposes only and should be removed before moving the Authorization Server into production.

For example, the default example client applications include the following:

| Client ID | Client Secret |
|---|---|
| `SampleConfidentialApp` | `6808d4b6-ef09-4b0d-8f28-3b05da9c48ec` |
| `SamplePublicApp` | `3b001542-e348-443b-9ca2-2f38bd3f3e84` |

**Importing the sample client applications**
To import the pre-registered example client applications, perform the following steps:

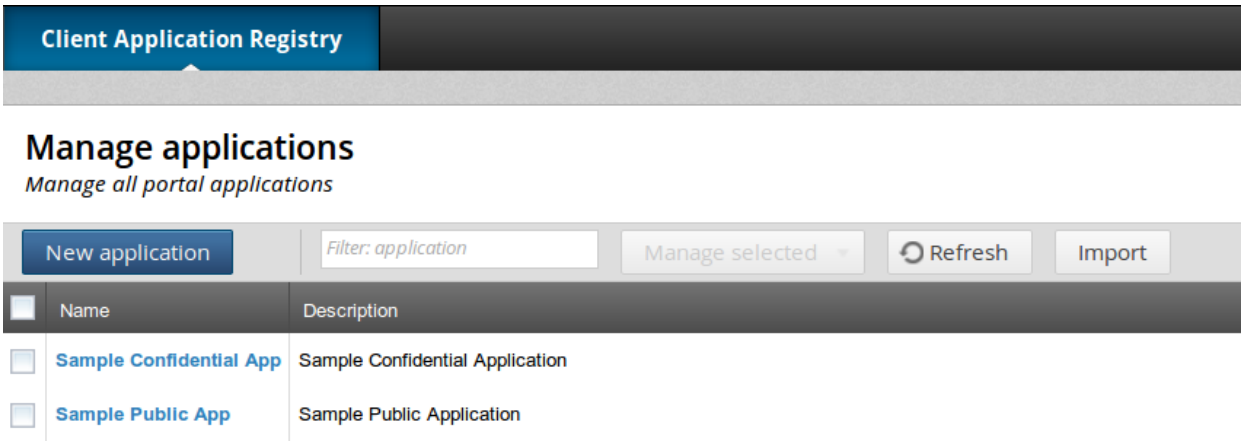
1. Access the **Client Application Registry** Web interface at the following URL:

   ```
   https://localhost:8089
   ```

2. Enter the default username/password of `admin/changeme`. Alternatively, if you have installed the API Management Solution Pack, enter `apiadmin@localhost/changeme`.
3. Click the **Import** button at the top right of the screen.
4. Select the following sample file in the dialog:

   ```
   $VDISTDIR/samples/scripts/oauth/sampleapps.dat
   ```

   `VDISTDIR` specifies the directory in which the API Gateway is installed.
5. You can also enter a **Decryption Secret** in the dialog. However, the `sampleapps.dat` file is in plaintext format, and does not require a password.
6. Click **OK** to import the two sample applications. The following screen shows these applications imported into the **Client Application Registry**:




Alternatively, you can use the following script to import the sample client application data without using the Client Application Registry Web interface:

```
$VDISTDIR/samples/scripts/oauth/importSampleData.py
```

You can edit this script to configure your user credentials and file location.

## Migrating Client Applications

If you are migrating from API Gateway version 11.1.2.0.x, you can use the following script to migrate your existing OAuth client applications:

```
$VDISTDIR/samples/scripts/oauth/migrateFrom71.py
```

This script enables you to first export your existing client application data, which you can then import as described in the section called “Importing Client Applications”. This script has a `--password` parameter if you wish to encrypt the exported data for transport.

**Migrating your existing client applications**
To migrate your existing client applications, perform the following steps:

1. After installing API Gateway 11.1.2.2.0, copy the `$VDISTIR/samples/oauth/migrateFrom71.py` file to the same location in your existing API Gateway 11.1.2.0.x installation:

   ```
   $VDISTIR/samples/oauth/migrateFrom71.py
   ```

2. In your existing API Gateway 11.1.2.0.x installation, ensure that `$VDISTIR/samples/scripts/common.py` has the correct `defServerName` and `defGroupName` variables set for your existing topology.
3. Run the `migrateFrom71.py` script against your running version 11.1.2.0.x Admin Node Manager and API Gateway. The script outputs the following file:

   ```
   $VDISTIR/samples/oauth/appregistry/encodedapps.dat
   ```

   ### Note

   If you wish to encrypt the data, run the script with the `--password` parameter.

4. Check the `encodedapps.dat` file to ensure that the export has been successful.
5. Import the `encodedapps.dat` output by the script into a running API Gateway 11.1.2.2.0 using the Client Applica-

tion Registry web interface. For more details, see the section called “Importing Client Applications”. When importing encrypted data, you must enter a password in the **Decryption Secret** field.

## Upgrading API Gateway Configuration

If you are migrating from a previous API Gateway version, you must upgrade your API Gateway configuration. To generate an upgraded API Gateway version 11.1.2.2.0 configuration, perform the following steps:

1. Run the following script from your version 11.1.2.2.0 installation directory:

```
<11.1.2.2.0_install>/platform/bin/upgradeConfig --groups -d <previous-version-install>
  -o path/to/upgrade/output/
```

2. In Policy Studio, select **File** -> **Open File**.
3. Specify the following file:

```
path/to/upgrade/output/groups/group-2/conf/<guid>/configs.xml
```

4. In the open configuration in the Policy Studio tree, under **Key Property Stores**, delete **ApiKeyStore** and **ClientApplicationRegistry**.
5. Select **File** -> **Save** -> **Deployment Package** to export a `.fed` file.
6. Start the version 11.1.2.2.0 Admin Node Manager and API Gateway instance.
7. In Policy Studio, close the connection to the file, and connect to the now running 7.2 Admin Node Manager. Before connecting to the API Gateway instance, click **Deploy**.
8. Click **Browse for .fed**, and select the `.fed` file exported previously in step 4.
9. Import the client applications using the the web-based portal on `https://localhost:8089` by clicking **Import**, and browsing to the file created in the previous section:

```
<11.1.2.2.0_install>/samples/oauth/appregistry/encodedapps.dat>
```

For more details on upgrading API Gateway configuration, see the *API Gateway Installation and Configuration Guide*.

# Managing OAuth 2.0 Applications

## Overview

Client applications that send OAuth requests to the API Gateway's Authorization Server must be registered with the Authorization Server. This chapter describes the registry used to store these client applications, and how to manage them using a REST API-based HTML interface. This topic also includes details on the relational database schema, and SSL commands used for the example client applications.

### Note

This topic assumes that you have already performed the steps described in *Setting up API Gateway OAuth 2.0*. These include enabling the OAuth endpoints, importing sample applications, and migrating existing client applications.

## Managing Registered Client Applications

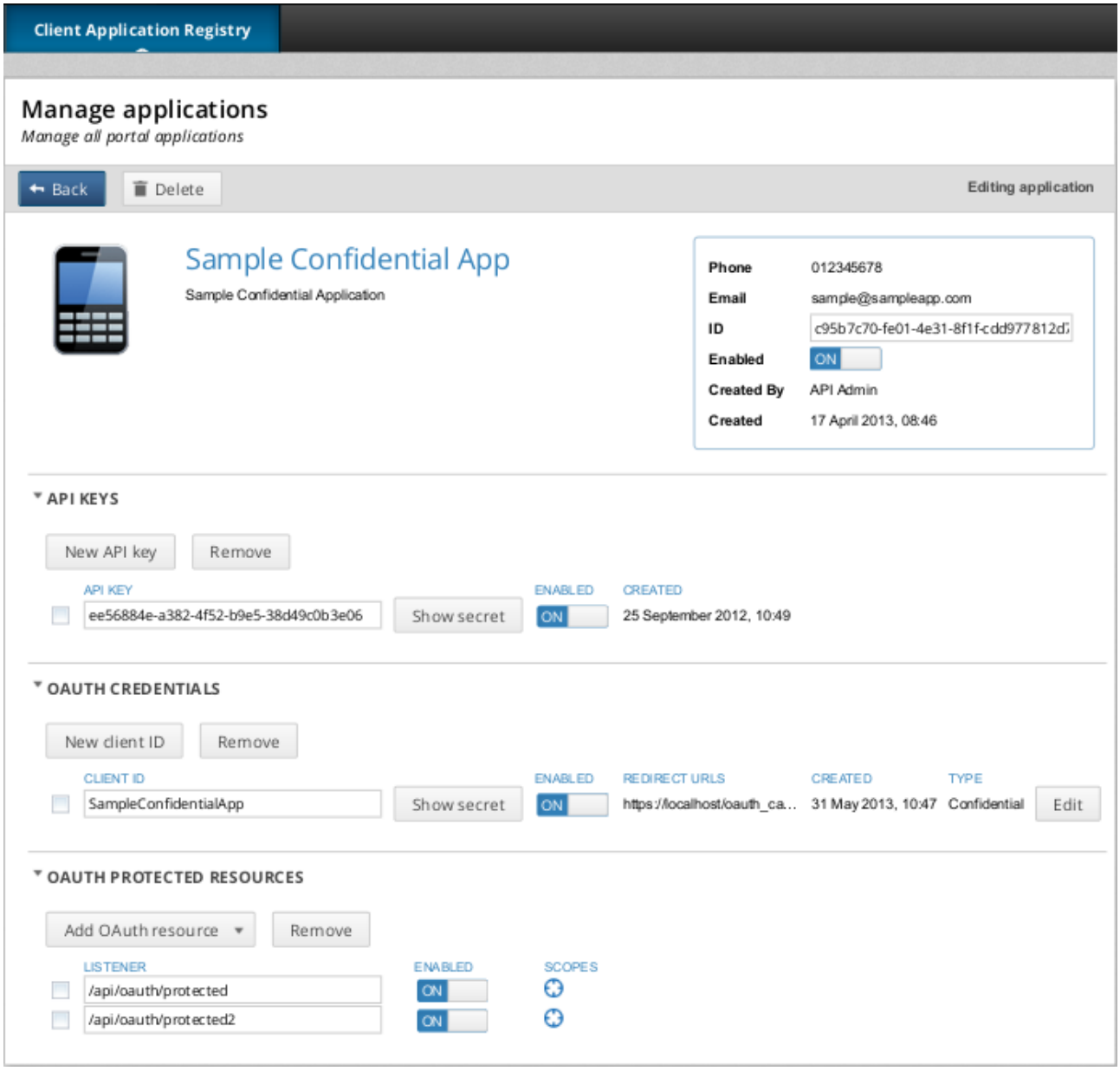
Every client application that sends OAuth requests to the API Gateway's OAuth Authorization Server must be registered with the Client Application Registry. The API Gateway provides the Client Application Registry Web-based HTML interface for managing registered client applications. If you have the API Management Solution Pack installed, the Client Application Registry is available in the API Portal web-based interface. The API Gateway also provides the Client Application Registry REST API to enable you to manage registered clients on the command line.

**Accessing the Client Application Registry Web Interface**
You can access the Client Application Registry Web interface at the following URL:

```
https://localhost:8089
```

The default username/password is `admin/changeme`. Alternatively, if you have installed the API Management Solution Pack, enter `apiadmin@localhost/changeme`.

You can select a client registration entry to update its details. For example, you can can configure API keys, OAuth credentials, and protected resources:




By default, the Client Application Registry is backed by an embedded Apache Cassandra database.

## Running the Sample Client Applications

The API Gateway includes sample Jython client applications for all supported OAuth flows in the following directory your API Gateway installation:

```
INSTALL_DIR/samples/scripts/oauth
```

To run a sample script, open a UNIX shell or DOS command prompt in the following directory:

```
INSTALL_DIR/samples/scripts
```

**Windows**
For example, run the following command:

```
> run.bat oauth\implicit_grant.py
```

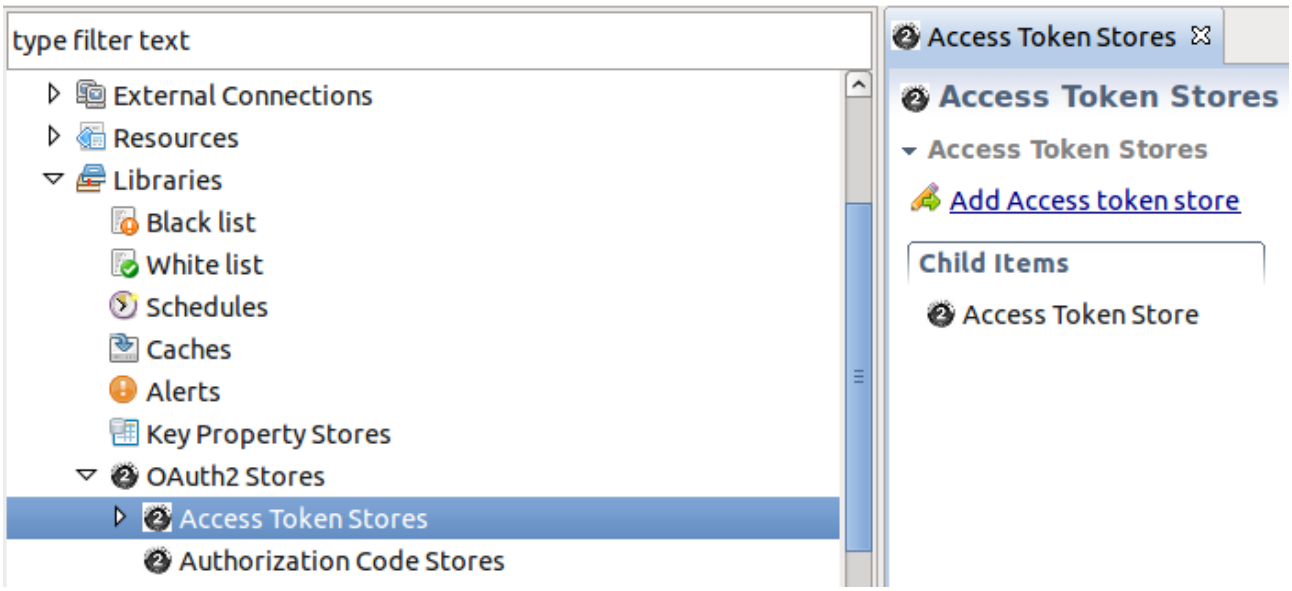
**Linux/Solaris**
For example, run the following command:

```
> sh run.sh oauth/implicit_grant.py
```

## Managing Access Tokens and Authorization Codes

The API Gateway can store generated authorization codes and access tokens in its caches, in an embedded database, or in a relational database. The Authorization Server issues tokens to clients on behalf of a Resource Owner to use when authenticating subsequent API calls to the Resource Server. These issued tokens must be persisted so that subsequent client requests to the Authorization Server can be validated.

The following screen shows the OAuth stores in the Policy Studio:

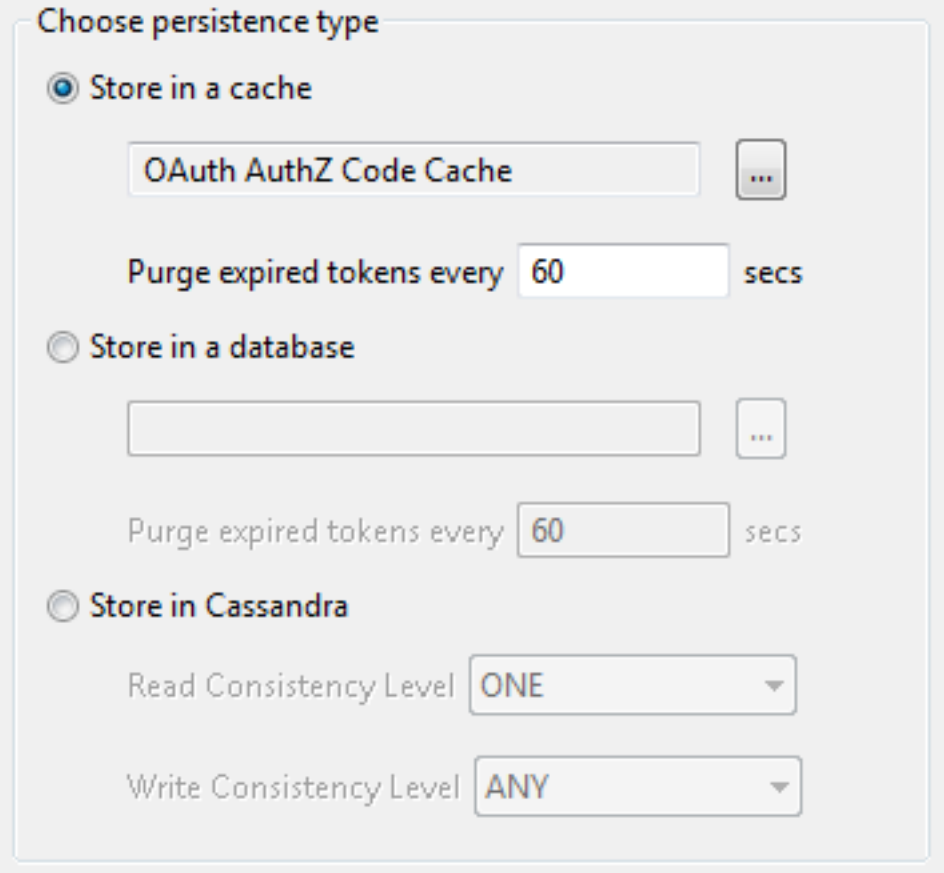



The Authorization Server can cache authorization codes and access tokens depending on the OAuth flow. The steps for adding an authorization code cache are similar to adding an access token cache.

The Authorization Server offers the following persistent storage options for access tokens and authorization codes:

- API Gateway cache (default)
- Relational Database Management System (RDBMS)
- Embedded Apache Cassandra database

The following screen shows these options in the Policy Studio:



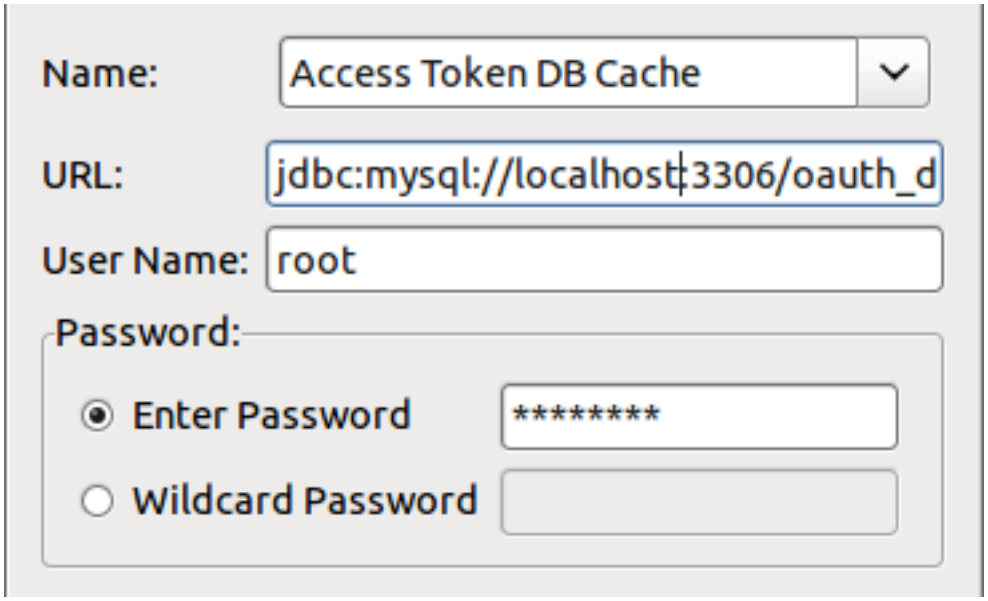

The **Purge expired tokens every** 60 secs setting enables you to configure the time in seconds that a background process polls the cache or database looking for expired access/refresh tokens or authorization codes.

**Storing in a cache**
Perform the following steps:

1. Right-click **Access Token Stores** in the Policy Studio tree, and select **Add Access Token Store**.
2. In dialog that enables you to choose the persistence type, select **Store in a cache**, and select the browse button to display the cache configuration dialog.
3. Add a new cache (for example, `OAuth Access Token Cache`). For more details, see the *API Gateway User Guide*.

**Storing in a relational database**
Perform the following steps:

1. Right-click **Access Token Stores** in the Policy Studio tree, and select **Add Access Token Store**.
2. In the dialog that enables you to choose the persistence type, select **Store in a database**, and select the browse button to display a database configuration dialog.
3. Complete the database configuration details. The following example uses a MySQL instance named `oauth_db`. For more details, see the *API Gateway User Guide*.




### Note

On first use of the database for caching access tokens, the following tables are created automatically: `oauth_access_token` and `oauth_refresh_token`. A table named `oauth_authz_code` is created for caching authorization codes.

For more details, see the section called "Relational Database-Backed Client Application Registry".

**Storing in Cassandra**
Perform the following steps:

1. Right-click **Access Token Stores** in the Policy Studio tree, and select **Add Access Token Store**.
2. This displays the dialog that enables you to choose the persistence type. Select **Store in Cassandra**.
3. You can configure **Read** and **Write** consistency levels for the Cassandra database. These control how up-to-date and synchronized a row of data is on all of its replicas. The default **Read** setting of `ONE` means that the database returns a response from the closest replica. The default **Write** setting of `ANY` means that a write must be written to at least one replica node. For more details, see http://www.datastax.com/docs/0.8/dml/data_consistency.

## Querying OAuth 2.0 Message Attributes

Most of the OAuth 2.0 policy filters in the API Gateway generate message attributes that can be queried further using API Gateway selector syntax. The message attributes generated by the OAuth filters are as follows:

- `accesstoken`
- `accesstoken.authn`
- `authzcode`
- `authentication.subject.id`
- `oauth.client.details`

**accesstoken methods**
The following methods are available to call on the `accesstoken` message attribute:

```
${accesstoken.getValue()}
${accesstoken.getExpiration()}
${accesstoken.getExpiresIn()}
${accesstoken.isExpired()}
${accesstoken.getTokenType()}
${accesstoken.getRefreshToken()}
${accesstoken.getOAuth2RefreshToken().getValue()}
${accesstoken.getOAuth2RefreshToken().getExpiration()}
${accesstoken.getOAuth2RefreshToken().getExpiresIn()}
${accesstoken.getOAuth2RefreshToken().hasExpired()}
${accesstoken.hasRefresh()}
${accesstoken.getScope()}
${accesstoken.getAdditionalInformation()}
```

The following example shows output from querying each of the `accesstoken` methods:

```
so0HlJYASrnXqn2fL2VWgiunaLfSBhWv6W7JMbmOa131HoQzZB1rNJ
Fri Oct 05 17:16:54 IST 2012
3599
false
Bearer
xif9oNHi83N4ETQLQxmSGoqfu9dKcRcFmBkxTkbc6yHDfK
xif9oNHi83N4ETQLQxmSGoqfu9dKcRcFmBkxTkbc6yHDfK
Sat Oct 06 04:16:54 IST 2012
43199
false
true
https://localhost:8090/auth/userinfo.email
{department=engineering}
```

**accesstoken.authn methods**
The following methods are available to call on the `accesstoken.authn` message attribute:

```
${accesstoken.authn.getUserAuthentication()}
${accesstoken.authn.getAuthorizationRequest().getScope()}
${accesstoken.authn.getAuthorizationRequest().getClientId()}
${accesstoken.authn.getAuthorizationRequest().getState()}
${accesstoken.authn.getAuthorizationRequest().getRedirectUri()}
${accesstoken.authn.getAuthorizationRequest().getParameters()}
```

The following example shows output from querying each of the `accesstoken.authn` methods:

```
admin
[https://localhost:8090/auth/userinfo.email]
SampleConfidentialApp
343dqak32ksla
https://localhost/oauth_callback
{client_secret=6808d4b6-ef09-4b0d-8f28-3b05da9c48ec,
  scope=https://localhost:8090/auth/userinfo.email, grant_type=authorization_code,
  redirect_uri=https://localhost/oauth_callback, state=null,
  code=FOT4nudbglQouujRl8oH3EOMzaOlQP, client_id=SampleConfidentialApp}
```

**authzcode methods**
The following methods are available to call on the `authzcode` message attribute:

```
${authzcode.getCode()}
${authzcode.getState()}
${authzcode.getApplicationName()}
${authzcode.getExpiration()}
${authzcode.getExpiresIn()}
```

```
${authzcode.getRedirectURI()}
${authzcode.getScopes()}
${authzcode.getUserIdentity()}
${authzcode.getAdditionalInformation()}
```

The following example shows output from querying each of the `authzcode` methods:

```
F8aHby7zctNRknmWlp3voe61H20Md1
sds12dsd3343ddsd
SampleConfidentialApp
Fri Oct 05 15:47:39 IST 2012
599 (expiry in secs)
https://localhost/oauth_callback
[https://localhost:8090/auth/userinfo.email]
admin
{costunit=hr}
```

**oauth.client.details methods**
The following methods are available to call on the `oauth.client.details` message attribute:

```
${authzcode.getCode()}
${authzcode.getState()}
${authzcode.getApplicationName()}
${authzcode.getExpiration()}
${authzcode.getExpiresIn()}
${authzcode.getRedirectURI()}
${authzcode.getScopes()}
${authzcode.getUserIdentity()}
```

The following example shows output from querying each of the `oauth.client.details` methods:

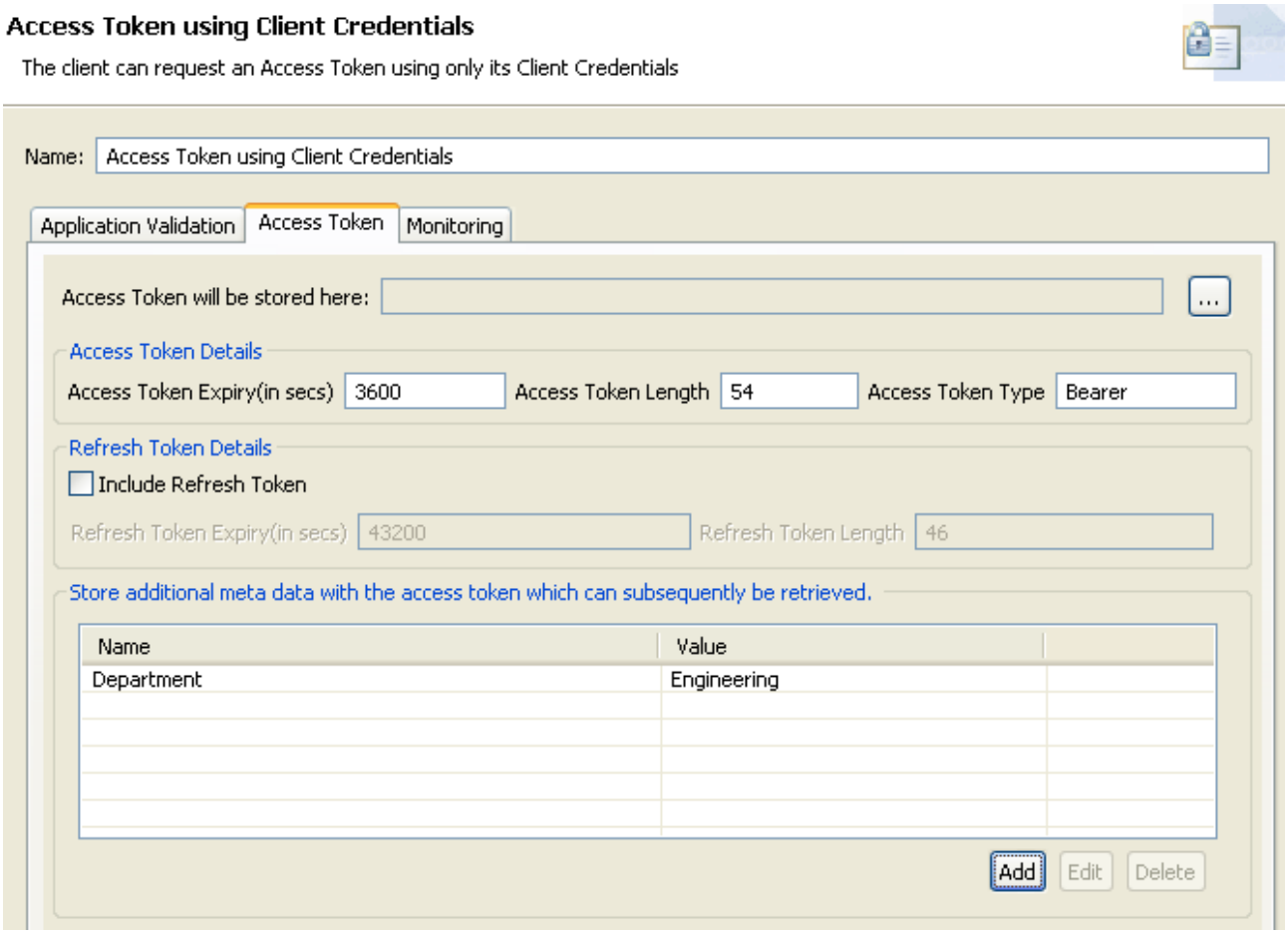
```
F8aHby7zctNRknmWlp3voe61H20Md1
sds12dsd3343ddsd
SampleConfidentialApp
Fri Oct 05 15:47:39 IST 2012
599 (expiry in secs)
https://localhost/oauth_callback
[https://localhost:8090/auth/userinfo.email]
admin
```

**Example of querying message attribute**
If you add additional access token parameters to the OAuth 2.0 **Access Token Info** filter, you can return a lot of additional information about the token. For example:

```
{
   "audience" : "SampleConfidentialApp",
   "user_id" : "admin",
   "scope" : "https://localhost:8090/auth/userinfo.email",
   "expires_in" : 3567,
   "Access Token Expiry Date" : "Wed Aug 15 11:19:19 IST 2012",
   "Authentication parameters" : "{username=admin,
    client_secret=6808d4b6-ef09-4b0d-8f28-3b05da9c48ec,
    scope=https://localhost:8090/auth/userinfo.email, grant_type=password,
    redirect_uri=null, state=null, client_id=SampleConfidentialApp,
    password=changeme}",
   "Access Token Type:" : "Bearer"
```

You also have the added flexibility to add extra name/value pair settings to access tokens upon generation.The OAuth

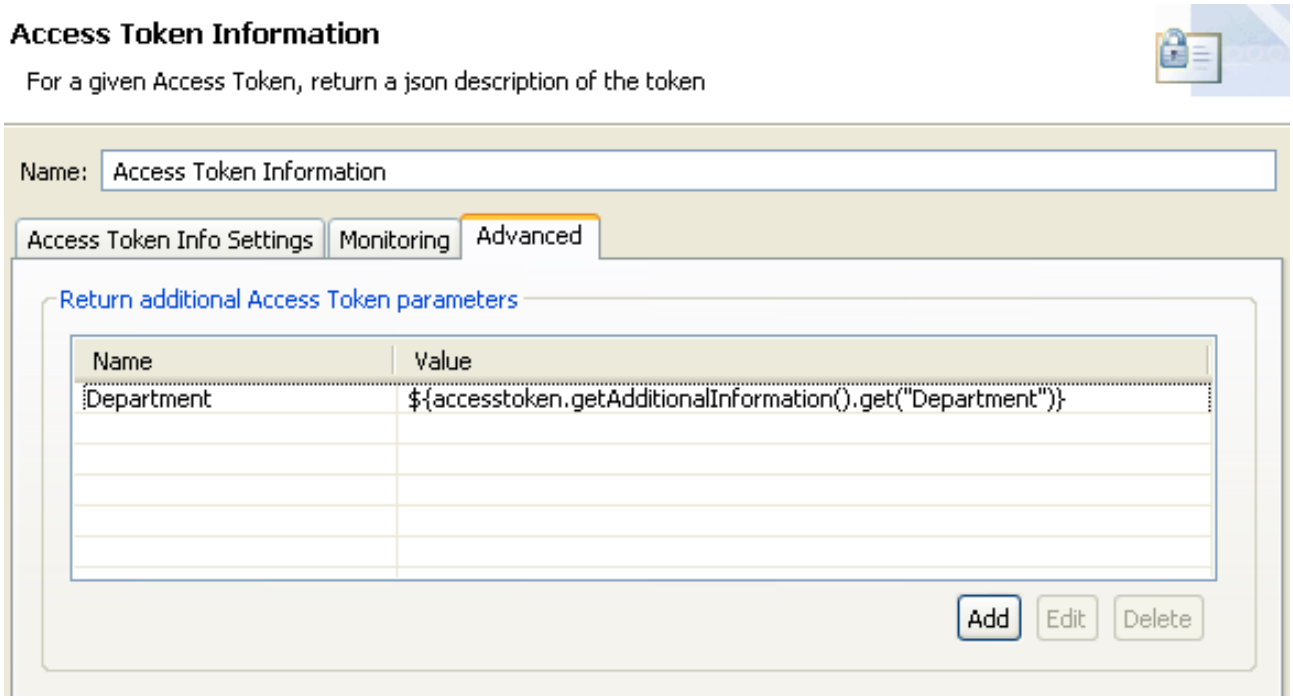
2.0 access token generation filters provide an option to store additional parameters for an access token. For example, if you add the name/value pair `Department/Engineering` to the **Client Credentials** filter:




You can then update the **Access Token Info** filter to add a name/value pair using a selector to get the following value:

```
Department/${accesstoken.getAdditionalInformation().get("Department")}
```

For example:




Then the JSON response is as follows:

```
{
  "audience" : "SampleConfidentialApp",
  "user_id" : "SampleConfidentialApp",
  "scope" : "https://localhost:8090/auth/userinfo.email",
  "expires_in" : 3583,
  "Access Token Type:" : "Bearer",
  "Authentication parameters" :
  "{client_secret=6808d4b6-ef09-4b0d-8f28-3b05da9c48ec,
    scope=https://localhost:8090/auth/userinfo.email, grant_type=client_credentials,
    redirect_uri=null, state=null, client_id=SampleConfidentialApp}",
  "Department" : "Engineering",
  "Access Token Expiry Date" : "Wed Aug 15 12:10:57 IST 2012"
```

You can also use API Gateway selector syntax when storing additional information with the token. For more details on selectors, see the *API Gateway User Guide*.

**OAuth scope attributes**
In addition, the following message attributes are used by the OAuth filters to manage OAuth scopes. The scopes are stored as a set of strings (for example, `https://localhost:8090/auth/user.photos` `https://localhost:8090/auth/userinfo.email`):

- `scopes.in.token`
  Stores the OAuth scopes that have been sent in to the Authorization Server when requesting the access token.
- `scopes.for.token`
  Stores the OAuth scopes that have been granted for the access token request.
- `scopes.required`
  Used by the **Validate Access Token** filter only. If there is a failure accessing an OAuth resource due to incorrect scopes in the access token, an `insufficent_scope` exception is sent back in the `WWW-Authenticate` header. When **Get scopes by calling a policy** is set, the configured policy can set the `scopes.required` message attribute. This enables the OAuth Resource Server to properly interact with client applications and provide useful error re-

sponse messages. For example:

```
WWW-Authenticate  Bearer realm="DefaultRealm",error="insufficient_scope",
   error_description="scope(s) associated with access token are not valid
   to access this resource", scope="Scopes must match All of these scopes:
   https://localhost:8090/auth/user.photos https://localhost:8090/auth/userinfo.email"
```

## Relational Database-Backed Client Application Registry

By default, the Oracle Client Application Registry Key Property Store (KPS) is backed by an Apache Cassandra database. The Oracle Client Application Registry KPS can also be backed by a relational database such as Oracle, MySQL, DB2, or Microsoft MySQL Server. For more details, see the *Key Property Store User Guide*, available from Oracle Support.

**OAuth relational database schemas**
For example, the OAuth relational database schemas displayed by example `mysql` commands are as follows:

**oauth_access_token schema**
The following shows the result from the `show columns from auth_access_token;` command:

```
+---------------+--------------+------+-----+---------+-------+
| Field         | Type         | Null | Key | Default | Extra |
+---------------+--------------+------+-----+---------+-------+
| id            | varchar(2    | NO   | PRI | NULL    |       |
| auth_request  | blob         | NO   |     | NULL    |       |
| client_id     | varchar(255  | NO   |     | NULL    |       |
| expiry_ti     | datetime     | NO   |     | NULL    |       |
| token         | blob         | NO   |     | NULL    |       |
| refresh_token | varchar(2    | YES  |     | NULL    |       |
| user_auth     | varchar(255  | NO   |     | NULL    |       |
| user_name     | varchar(255  | NO   |     | NULL    |       |
+---------------+--------------+------+-----+---------+-------+
```

**oauth_refresh_token schema**
The following shows the result from the `show columns from oauth_refresh_token;` command:

```
+--------------+--------------+------+-----+---------+-------+
| Field        | Type         | Null | Key | Default | Extra |
+--------------+--------------+------+-----+---------+-------+
| token_id     | varchar(255) | NO   | PRI | NULL    |       |
| auth_request | blob         | NO   |     | NULL    |       |
| expiry_time  | datetime     | NO   |     | NUL     |       |
| token        | blob         | NO   |     | NULL    |       |
| user_name    | varchar(255) | NO   |     | NULL    |       |
+--------------+--------------+------+-----+---------+-------+
```

**oauth_refresh_token schema**
The following shows the result from the `show columns from oauth_refresh_token;` command:

```
+---------------+--------------+------+-----+---------+-------+
| Field         | Type         | Null | Key | Default | Extra |
+---------------+--------------+------+-----+---------+-------+
| id            | varchar(255) | NO   | PRI | NULL    |       |
| authorization | blob         | NO   |     | NULL    |       |
| expiry_time   | datetime     | NO   |     | NULL    |       |
+---------------+--------------+------+-----+---------+-------+
```

## Generating a Certificate and Private Key for a Client Application

The following example `openssl` command shows generating a client application certificate and private key:

```
$ openssl req -x509 -nodes -days 365 -newkey rsa:1024 -keyout mykey.pem
-out mycert.pem
Generating a 1024 bit RSA private key
.....................................................................
...........++++++
.....++++++
writing new private key to 'mykey.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank.
For some fields there will be a default value.
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MA
Locality Name (eg, city) []:Newton
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Oracle
Organizational Unit Name (eg, section) []:API Gateway
Common Name (eg, YOUR name) []:SampleConfidentialApp
Email Address []:support@widgits.com
```

# API Gateway OAuth 2.0 Authentication Flows

## Overview

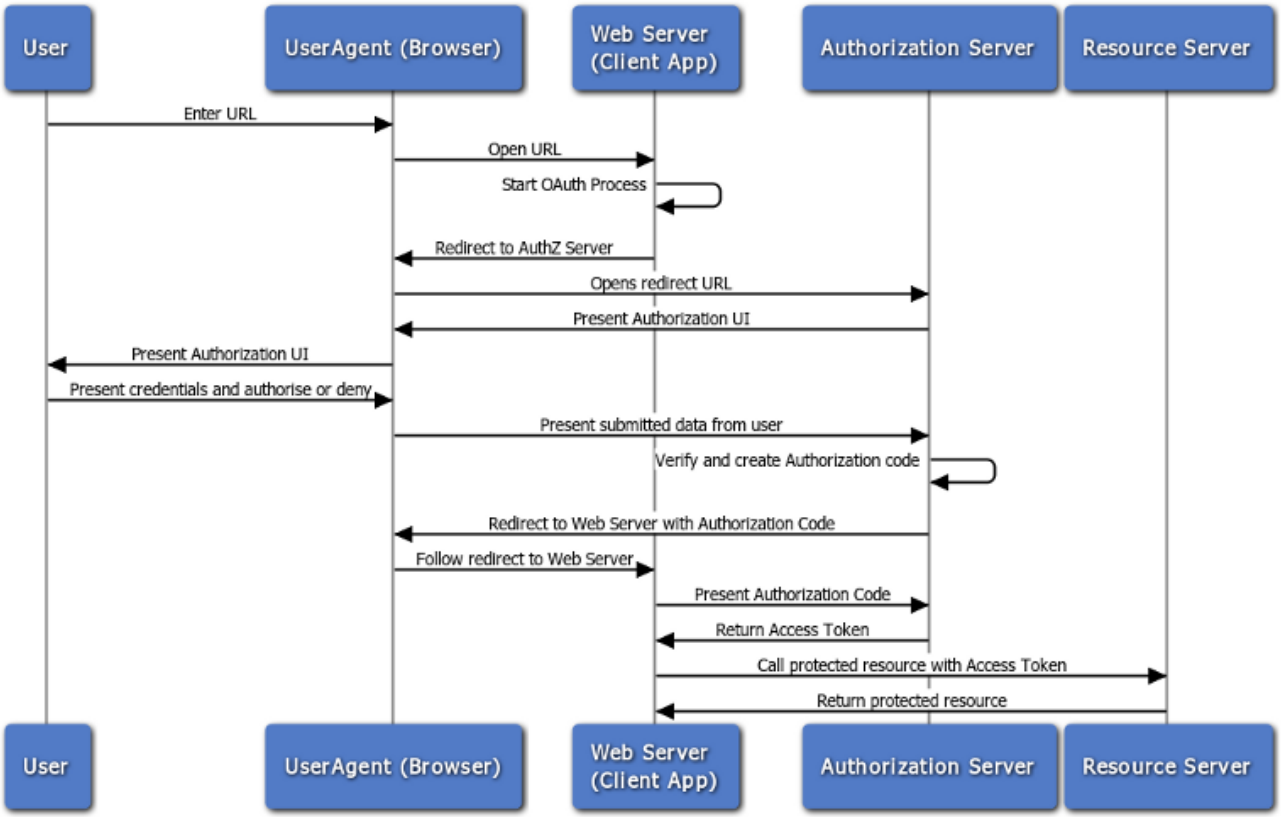
The API Gateway can use the OAuth 2.0 protocol for authentication and authorization. The API Gateway can act as an OAuth 2.0 Authorization Server and supports several OAuth 2.0 flows that cover common Web server, JavaScript, device, installed application, and server-to-server scenarios. This topic describes each of the supported OAuth 2.0 flows in detail, and shows how to run example client applications.

## Authorization Code (or Web Server) Flow

The Authorization Code or Web server flow is suitable for clients that can interact with the end-user's user-agent (typically a Web browser), and that can receive incoming requests from the authorization server (can act as an HTTP server). The Authorization Code flow is also known as the *Three-Legged OAuth* flow.

The Authorization Code flow is as follows:

1. The Web server redirects the user to the API Gateway acting as an Authorization Server to authenticate and authorize the server to access data on their behalf.
2. After the user approves access, the Web server receives a callback with an authorization code.
3. After obtaining the authorization code, the Web server passes back the authorization code to obtain an access token response.
4. After validating the authorization code, the API Gateway passes back a token response to the Web server.
5. After the token is granted, the Web server accesses their data.




**Obtaining an access token**
The detailed steps for obtaining an access token are as follows:

1. Redirect the user to the authorization endpoint with the following parameters:

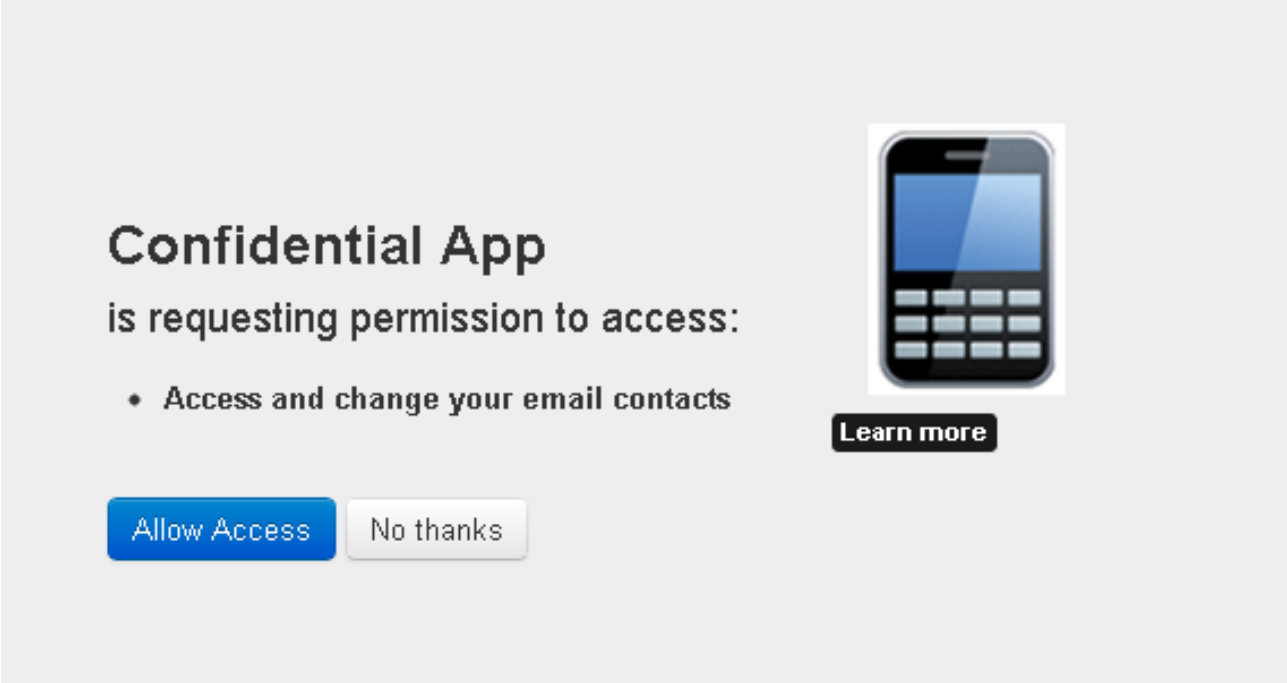
| Parameter | Description |
|---|---|
| `response_type` | Required. Must be set to `code`. |
| `client_id` | Required. The Client ID generated when the application was registered in the Oracle Client Application Registry. |
| `redirect_uri` | Optional. Where the authorization code will be sent. This value must match one of the values provided in the Oracle Client Application Registry. |
| `scope` | Optional. A space delimited list of scopes, which indicate the access to the Resource Owner's data being requested by the application. |
| `state` | Optional. Any state the consumer wants reflected back to it after approval during the callback. |

The following is an example URL:

```
https://apigateway/oauth/authorize?client_id=SampleConfidentialApp&
response_type=code&&redirect_uri=http%3A%2F%2Flocalhost%3A8090%2Fauth%2Fredirect.html&
scope=https%3A%2F%2Flocalhost%3A8090%2Fauth%2Fuserinfo.email
```

## Note

During this step the Resource Owner user must approve access for the application Web server to access their protected resources, as shown in the following example screen.




2. The response to the above request is sent to the `redirect_uri`. If the user approves the access request, the response contains an authorization code and the `state` parameter (if included in the request). If the user does not approve the request, the response contains an error message. All responses are returned to the Web server on the query string. For example:

```
https://localhost/oauth_callback&code=9srN6sqmjrvG5bWvNB42PCGju0TFVV
```

3. After the Web server receives the authorization code, it may exchange the authorization code for an access token and a refresh token. This request is an HTTPS `POST`, and includes the following parameters:

| Parameter | Description |
|---|---|
| `grant_type` | Required. Must be set to `authorization_code`. |
| `code` | Required. The authorization code received in the redirect above. |
| `redirect_uri` | Required. The redirect URL registered for the application during application registration. |

| Parameter | Description |
|---|---|
| client_id* | Optional. The client_id obtained during application registration. |
| client_secret* | Optional. The client_secret obtained during application registration. |
| format | Optional. Expected return format. The default is json. Possible values are:<br>• urlencoded<br>• json<br>• xml |

* If the client_id and client_secret are not provided as parameters in the HTTP POST, they must be provided in the HTTP Basic Authentication header (Authorization base64Encoded(client_id:client_secret)).

The following example HTTPS POST shows some parameters:

```
POST /api/oauth/token HTTP/1.1
Content-Type: application/x-www-form-urlencoded

client_id=SampleConfidentialApp&client_secret=6808d4b6-ef09-4b0d-8f28-3b05da9c48ec
 &code=9srN6sqmjrvG5bWvNB42PCGju0TFVV&redirect_uri=http%3A%2F%2Flocalhost%3A809
 0%2Fauth%2Fredirect.html&grant_type=authorization_code&format=query
```

4. After the request is verified, the API Gateway sends a response to the client. The following parameters are in the response body:

| Parameter | Description |
|---|---|
| access_token | The token that can be sent to the Resource Server to access the protected resources of the Resource Owner (user). |
| refresh_token | A token that may be used to obtain a new access token. |
| expires | The remaining lifetime on the access token. |
| type | Indicates the type of token returned. At this time, this field always has a value of Bearer. |

The following is an example response:

```
HTTP/1.1 200 OK
Cache-Control: no-store
Content-Type: application/json
Pragma: no-cache{
    "access_token": “O91G451HZ0V83opz6udiSEjchPynd2Ss9......",
    "token_type": "Bearer",
    "expires_in": "3600",
```

```
}
```

5. After the Web server has obtained an access token, it can gain access to protected resources on the Resource Server by placing it in an `Authorization: Bearer` HTTP header:

```
GET /oauth/protected HTTP/1.1
Authorization: Bearer O91G451HZ0V83opz6udiSEjchPynd2Ss9
Host: apigateway.com
```

For example, the `curl` command to call a protected resource with an access token is as follows:

```
curl -H "Authorization: Bearer O91G451HZ0V83opz6udiSEjchPynd2Ss9"
  https://apigateway.com/oauth/protected
```

**Running the sample client**
The following Jython sample client creates and sends an authorization request for the authorization grant flow to the Authorization Server:

```
INSTALL_DIR/samples/scripts/oauth/authorization_code.py
```

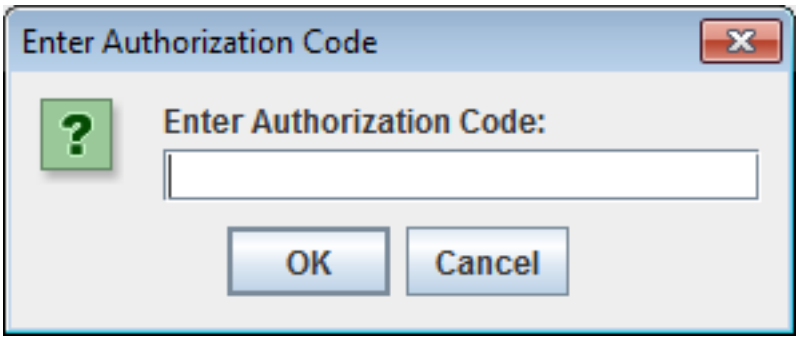
To run the sample, perform the following steps:

1. Open a shell prompt at `INSTALL_DIR/samples/scripts`, and execute the following command:

   ```
   > run oauth/authorization_code.py
   ```

   The script outputs the following:

   ```
   > Go to the URL here:
   http://127.0.0.1:8080/api/oauth/authorize?client_id=SampleConfidentialApp
     &response_type=code&scope=https%3A%2F%2Flocalhost%3A8090%2Fauth%2Fuserinfo.email
     &redirect_uri=https%3A%2F%2Flocalhost%2Foauth_callback
   Enter Authorization code in dialog
   ```

2. Copy the URL output to the command prompt into a browser, and perform the following steps as prompted:
   a. Provide login credentials to the authorization server. The default values are:
      - Username: `admin`
      - Password: `changeme`
   b. When prompted, grant access to the client application to access the protected resource.
3. After the Resource Owner has authorized and approved access to the application, the Authorization Server redirects a fragment containing the authorization code to the redirection URI. For example:

```
https://localhost/oauth_callback&code=AaI5Or3RYB2uOgiyqVsLs1ATIY0ll0
```

In this example, the authorization code is:

```
AaI5Or3RYB2uOgiyqVsLs1ATIY0ll0
```

Enter this value into the **Enter Authorization Code** dialog. The script will exchange the authorization code for an access token, and then access the protected resource using the access token. For example:

```
Enter Authorization code in dialog
AuthZ code: AaI5Or3RYB2uOgiyqVsLs1ATIY0ll0
Exchange authZ code for access token
Sending up access token request using grant_type set to authorization_code
Response from access token request: 200
Parsing the json response
**********ACCESS TOKEN RESPONSE**********
Access token received from authorization server icPgKP2uVUD2thvAZ5ENhsQb66ffnZEC
 XHyRQEz5zP8aGzcobLV3AR
Access token type received from authorization server Bearer
Access token expiry time: 3599
Refresh token:  NpNbzIVVvj8MhMmcWx2zsawxxJ3YADfc0XIxlZvw0tIhh8
**********
Now we can try access the protected resource using the access token
Executing get request on the protected url
Response from protected resource request is: 200
<html>Congrats! You've hit an OAuth protected resource</html>
```

**Further information**
For details on API Gateway filters that support this flow, see the following topics:

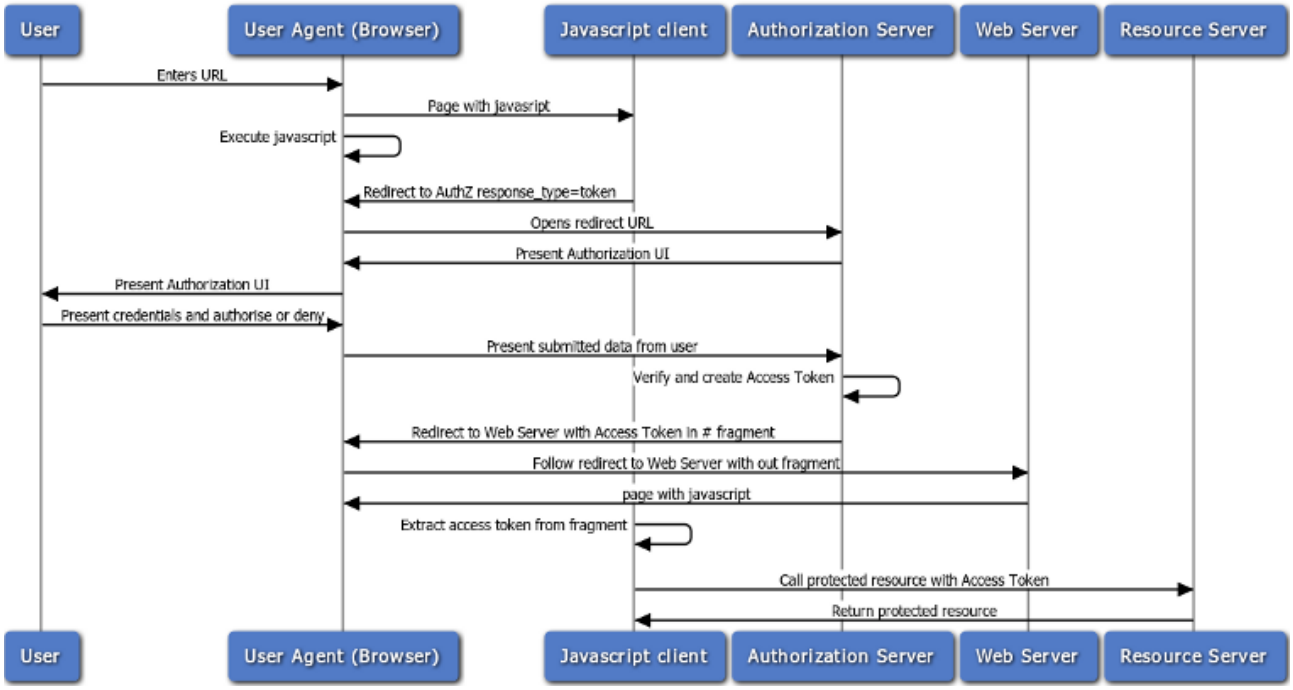
- *Access Token Using Authorization Code*
- *Authorization Code Flow*
- *Authorize Transaction*

## Implicit Grant (or User Agent) Flow

The Implicit Grant (User-Agent) authentication flow is used by client applications (consumers) residing in the user's device. This could be implemented in a browser using a scripting language such as JavaScript, or from a mobile device or a desktop application. These consumers cannot keep the client secret confidential (application password or private key).

The User Agent flow is as follows:

1. The Web server redirects the user to the API Gateway acting as an Authorization Server to authenticate and authorize the server to access data on their behalf.
2. After the user approves access, the Web server receives a callback with an access token in the fragment of the redirect URL.
3. After the token is granted, the application can access the protected data with the access token.




**Obtaining an access token**
The detailed steps for obtaining an access token are as follows:

1. Redirect the user to the authorization endpoint with the following parameters:

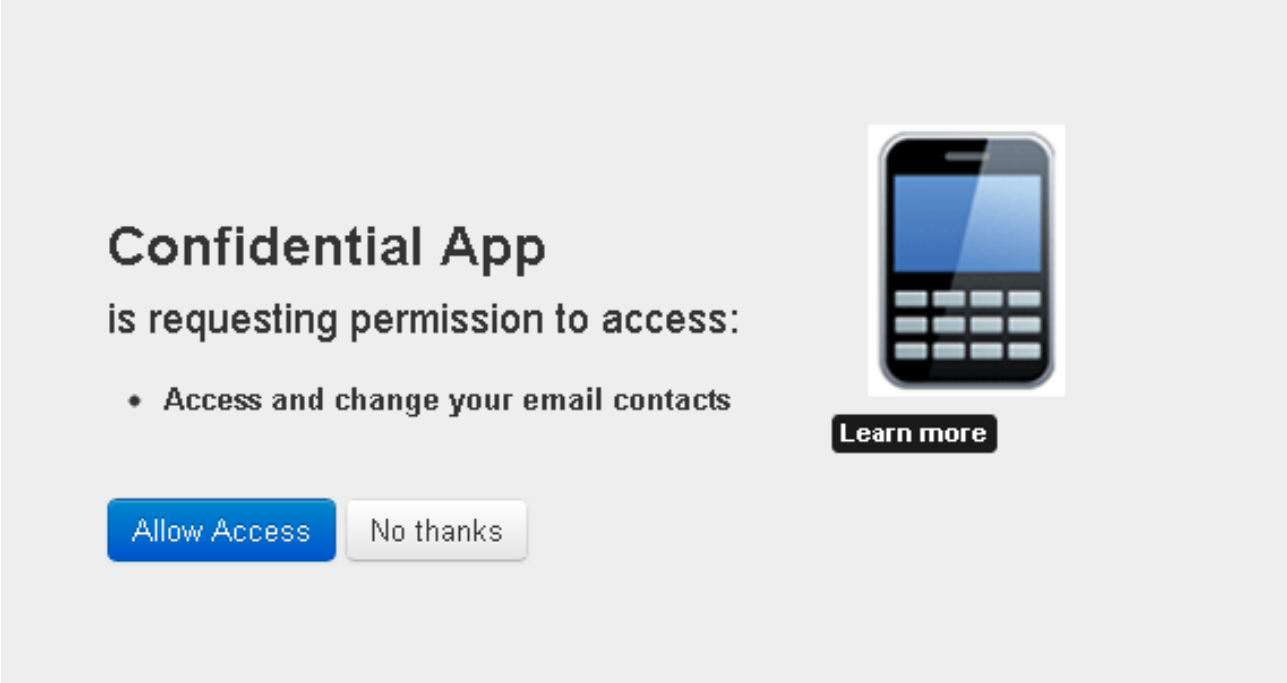
| Parameter | Description |
|---|---|
| `response_type` | Required. Must be set to token. |
| `client_id` | Required. The Client ID generated when the application was registered in the Oracle Client Application Registry. |
| `redirect_uri` | Optional. Where the access token will be sent. This value must match one of the values provided in the Oracle Client Application Registry. |
| `scope` | Optional. A space delimited list of scopes, which indicates the access to the Resource Owner's data requested by the application. |
| `state` | Optional. Any state the consumer wants reflected back to it after approval during the callback. |

The following is an example URL:

```
https://apigateway/oauth/authorize?client_id=SampleConfidentialApp&response_type=
  token&&redirect_uri=http%3A%2F%2Flocalhost%3A8090%2Fauth%2Fredirect.html&scope=
  https%3A%2F%2Flocalhost%3A8090%2Fauth%2Fuserinfo.email
```

## Note

During this step the Resource Owner user must approve access for the application (Web server) to access their protected resources, as shown in the following example screen.




2. The response to the above request is sent to the `redirect_uri`. If the user approves the access request, the response contains an access token and the state parameter (if included in the request). For example:

```
https://localhost/oauth_callback#access_token=19437jhj2781FQd44AzqT3Zg
 &token_type=Bearer&expires_in=3600
```

If the user does not approve the request, the response contains an error message.

3. After the request is verified, the API Gateway sends a response to the client. The following parameters are contained in the fragment of the redirect:

| Parameter | Description |
|---|---|
| `access_token` | The token that can be sent to the Resource Server to access the protected resources of the Resource Owner (user). |
| `expires` | The remaining lifetime on the access token. |
| `type` | Indicates the type of token returned. At this time, this field will always have a value of `Bearer`. |
| `state` | Optional. If the client application sent a value for state in the original authorization request, the state parameter is populated with this value. |

4. After the application has obtained an access token, it can gain access to protected resources on the Resource Server by placing it in an `Authorization: Bearer` HTTP header:

```
GET /oauth/protected HTTP/1.1
Authorization: Bearer O91G451HZ0V83opz6udiSEjchPynd2Ss9
Host: apigateway.com
```

For example, the `curl` command to call a protected resource with an access token is as follows:

```
curl -H "Authorization: Bearer O91G451HZ0V83opz6udiSEjchPynd2Ss9"
https://apigateway.com/oauth/protected
```

**Running the sample client**

The following Jython sample client creates and sends an authorization request for the implicit grant flow to the Authorization Server:

```
INSTALL_DIR/samples/scripts/oauth/implicit_grant.py
```

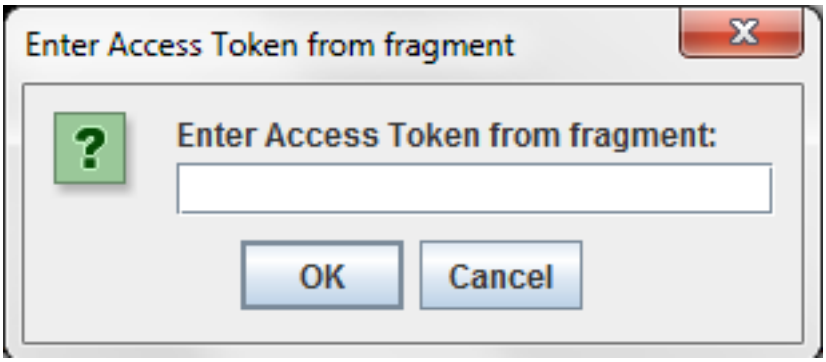
To run the sample, perform the following steps:

1. Open a shell prompt at `INSTALL_DIR/samples/scripts`, and execute the following command:

   ```
   > run oauth/implicit_grant.py
   ```

   The script outputs the following:

   ```
   > Go to the URL here:
    http://127.0.0.1:8080/api/oauth/authorize?client_id=SampleConfidentialApp&
     response_type=token&scope=https%3A%2F%2Flocalhost%3A8090%2Fauth%2Fuserinfo.email&
     redirect_uri=https%3A%2F%2Flocalhost%2Foauth_callback&state=1956901292
    Enter Access Token code in dialog
   ```

2. After the Resource Owner has authorized and approved access to the application, the Authorization Server redirects to the redirection URI a fragment containing the access token. For example:

   ```
   https://localhost/oauth_callback#access_token=
    4owzGyokzLLQB5FH4tOMk7Eqf1wqYfENEDXZ1mGvN7u7a2Xexy2OU9&expires_in=
    3599&state=1956901292&token_type=Bearer
   ```

   In this example, the access token is:

   ```
   4owzGyokzLLQB5FH4tOMk7Eqf1wqYfENEDXZ1mGvN7u7a2Xexy2OU9
   ```

Enter this value into the **Enter Access Token from fragment** dialog, and the script attempts to access the protected resource using the access token. For example:

```
**********************ACCESS TOKEN RESPONSE*****************************
Access token received from authorization server 4owzGyokzLLQB5FH4tOMk7Eqf1wqYfEN
EDXZ1mGvN7u7a2Xexy2OU9
************************************************************************
Now we can try access the protected resource using the access token
Executing get request on the protected url
Response from protected resource request is: 200
<html>Congrats! You've hit an OAuth protected resource</html>
```

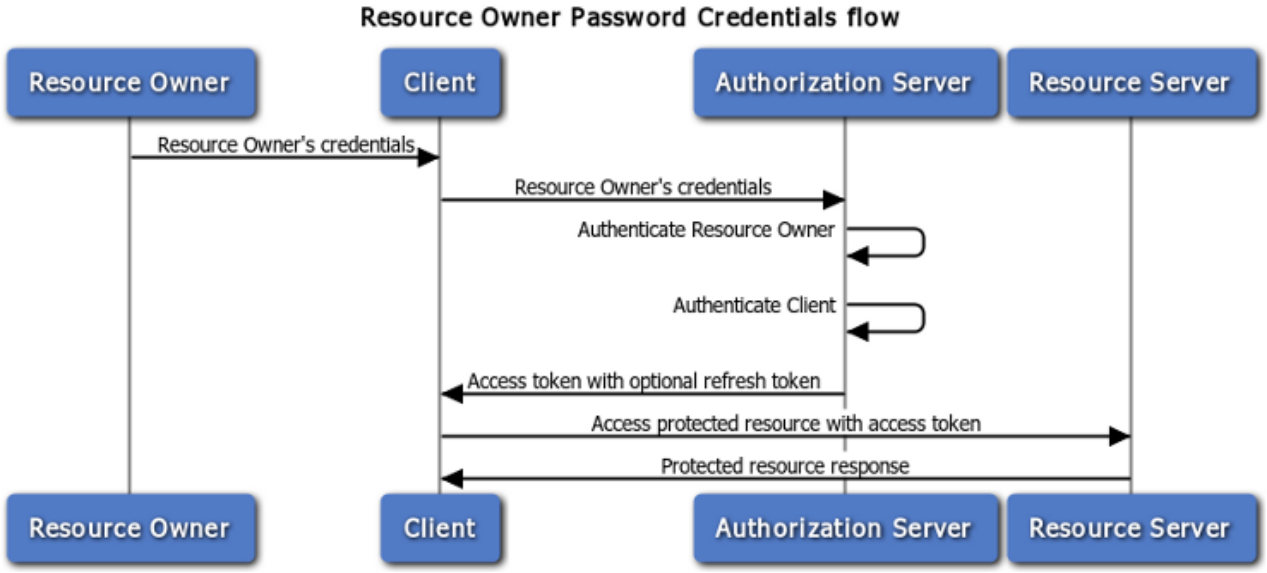
**Further information**
For details on the API Gateway filter that supports this flow, see the *Authorization Code Flow* filter.

## Resource Owner Password Credentials Flow

The Resource Owner password credentials flow is also known as the username-password authentication flow. This flow can be used as a replacement for an existing login when the consumer already has the user's credentials.

The Resource Owner password credentials grant type is suitable in cases where the Resource Owner has a trust relationship with the client (for example, the device operating system or a highly privileged application). The Authorization Server should take special care when enabling this grant type, and only allow it when other flows are not viable.

This grant type is suitable for clients capable of obtaining the Resource Owner's credentials (username and password, typically using an interactive form). It is also used to migrate existing clients using direct authentication schemes such as HTTP Basic or Digest authentication to OAuth by converting the stored credentials to an access token.




**Requesting an access token**
The client token request should be sent in an HTTP `POST` to the token endpoint with the following parameters:

| Parameter | Description |
|---|---|
| `grant_type` | Required. Must be set to `password` |

| Parameter | Description |
|---|---|
| `username` | Required. The Resource Owner's user name. |
| `password` | Required. The Resource Owner's password. |
| `scope` | Optional. The scope of the authorization. |
| `format` | Optional. Expected return format. The default is `json`. Possible values are:<br>• `urlencoded`<br>• `json`<br>• `xml` |

The following is an example HTTP `POST` request:

```
POST /api/oauth/token HTTP/1.1
Content-Length: 424
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Host: 192.168.0.48:8080
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JWgrant_type=password&username=
 johndoe&password=A3ddj3w
```

**Handling the response**
The API Gateway will validate the resource owner's credentials and authenticate the client against the Oracle Client Application Registry. An access token, and optional refresh token, is sent back to the client on success. For example, a valid response is as follows:

```
HTTP/1.1 200 OK
 Cache-Control: no-store
 Content-Type: application/json
 Pragma: no-cache
 {
     "access_token": "O91G451HZ0V83opz6udiSEjchPynd2Ss9......",
     "token_type": "Bearer",
     "expires_in": "3600",
     "refresh_token": "8722gffy2229220002iuueee7GP..........."
 }
```

**Running the sample client**
The following Jython sample client sends a request to the Authorization Server using the Resource Owner password credentials flow:

```
INSTALL_DIR/samples/scripts/oauth/resourceowner_password_credentials.py
```

To run the sample, open a shell prompt at `INSTALL_DIR/samples/scripts`, and execute the following command:

```
> run oauth/resourceowner_password_credentials.py
```

The script outputs the following:

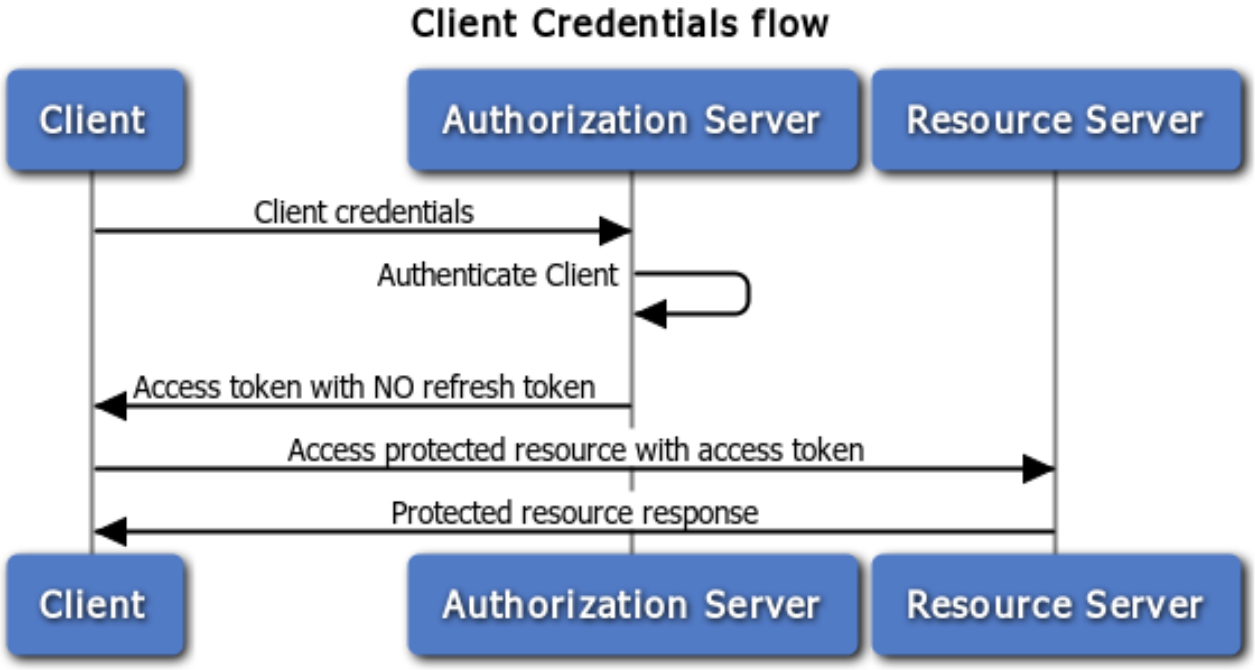
```
Sending up access token request using grant_type set to password
```

```
Response from access token request: 200
Parsing the json response
**********************ACCESS TOKEN RESPONSE*********************************
Access token received from authorization server lrGHhFhFwSmycXStIza1jjvXlSaac9
 JNIgviF7oPiV8OnxlSIsrxVA
Access token type received from authorization server Bearer
Access token expiry time: 3600
*****************************************************************************
Now we can try access the protected resource using the access token
Executing get request on the protected url
Response from protected resource request is: 200
<html>Congrats! You've hit an OAuth protected resource</html>
```

**Further information**
For details on the API Gateway filter that supports this flow, see *Resource Owner Credentials*.

## Client Credentials Grant Flow

The client credentials grant type must only be used by confidential clients. The client can request an access token using only its client credentials (or other supported means of authentication) when the client is requesting access to the protected resources under its control. The client can also request access to those of another Resource Owner that has been previously arranged with the Authorization Server (the method of which is beyond the scope of the specification).




**Requesting an access token**
The client token request should be sent in an HTTP `POST` to the token endpoint with the following parameters:

| Parameter | Description |
|---|---|
| `grant_type` | Required. Must be set to `client_credentials`. |
| `scope` | Optional. The scope of the authorization. |

| Parameter | Description |
|---|---|
| `format` | Optional. Expected return format. The default is `json`. Possible values are:<br>• `urlencoded`<br>• `json`<br>• `xml` |

The following is an example POST request:

```
POST /api/oauth/token HTTP/1.1
Content-Length: 424
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Host: 192.168.0.48:8080
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
grant_type=client_credentials
```

**Handling the response**
The API Gateway authenticates the client against the Oracle Client Application Registry. An access token is sent back to the client on success. A refresh token is not included in this flow. An example valid response is as follows:

```
HTTP/1.1 200 OK
Cache-Control: no-store
Content-Type: application/json
Pragma: no-cache
{    "access_token": “O91G451HZ0V83opz6udiSEjchPynd2Ss9......",
     "token_type": "Bearer",
     "expires_in": "3600"
}
```

**Running the sample client**
The following Jython sample client sends a request to the Authorization Server using the client credentials flow:

```
INSTALL_DIR/samples/scripts/oauth/client_credentials.py
```

To run the sample, open a shell prompt at `INSTALL_DIR/samples/scripts`, and execute the following command:

```
> run oauth/client_credentials.py
```

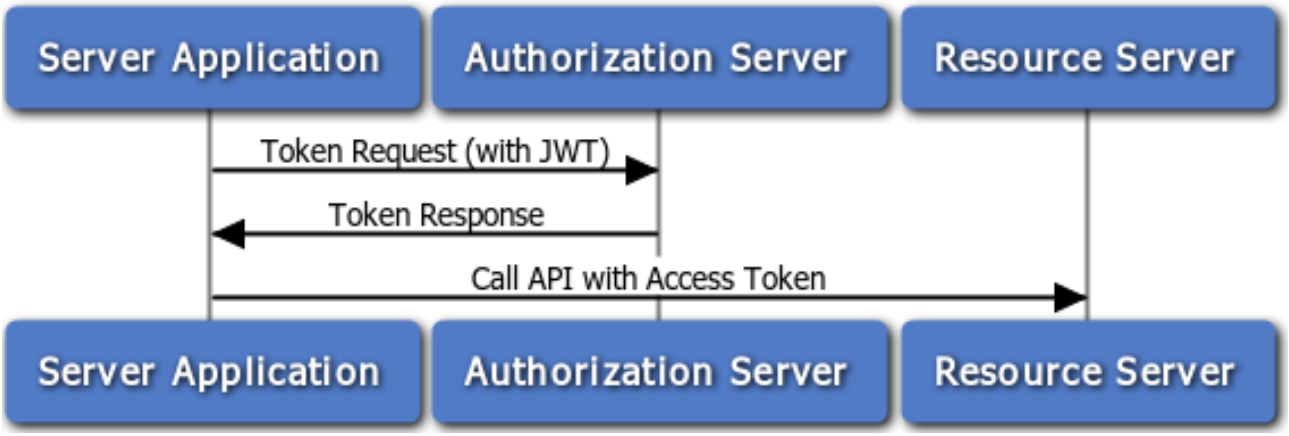
The outputs the following:

```
Sending up access token request using grant_type set to client_credentials
Response from access token request: 200
Parsing the json response
**********************ACCESS TOKEN RESPONSE*********************************
Access token received from authorization server
OjtVvNusLg2ujy3a6IXHhavqdEPtK7qSmIj9fLl8qywPyX8bKEsjqF
Access token type received from authorization server Bearer
Access token expiry time: 3599
****************************************************************************
Now we can try access the protected resource using the access token
Response from protected resource request is: 200
<html>Congrats! You've hit an OAuth protected resource</html>
```

**Further information**
For details on the API Gateway filter that supports this flow, see *Access Token Using Client Credentials*.

## JSON Web Token (JWT) Flow

A JSON Web Token (JWT) is a JSON-based security token encoding that enables identity and security information to be shared across security domains.




In the OAuth 2.0 JWT flow, the client application is assumed to be a confidential client that can store the client application's private key. The X.509 certificate that matches the client's private key must be registered in the Oracle Client Application Registry. The API Gateway uses this certificate to verify the signature of the JWT claim. For information on creating a private key and certificate, see the section called "Generating a Certificate and Private Key for a Client Application".

For more details on the OAuth 2.0 JWT flow, see
http://self-issued.info/docs/draft-ietf-oauth-jwt-bearer-00.html

**Creating a JWT bearer token**
To create a JWT bearer token, perform the following steps:

1. Construct a JWT header in the following format:

   ```
   {"alg":"RS256"}
   ```

2. Base64url encode the JWT Header as defined here, which results in the following:

   ```
   eyJhbGciOiJSUzI1NiJ9
   ```

3. Create a JWT Claims Set, which conforms to the following rules:
   - The issuer (`iss`) must be the OAuth `client_id` or the remote access application for which the developer registered their certificate.
   - The audience (`aud`) must match the value configured in the JWT filter. By default, this value is as follows:

     ```
     http://apigateway/api/oauth/token
     ```

   - The validity (`exp`) must be the expiration time of the assertion, within five minutes, expressed as the number of seconds from `1970-01-01T0:0:0Z` measured in UTC.
   - The time the assertion was issued (`iat`) measured in seconds after `00:00:00` UTC, January 1, 1970.
   - The JWT must be signed (using RSA SHA256).
   - The JWT must conform with the general format rules specified here:

http://tools.ietf.org/html/draft-jones-json-web-toke.
For example:

```
{
"iss": "SampleConfidentialApp",
    "aud": "http://apigateway/api/oauth/token",
    "exp": "1340452126",
    "iat": "1340451826"
}
```

4. Base64url encode the JWT Claims Set, resulting in:

```
eyJpc3MiOiJTYW1wbGVDb25maWRlbnRpYWxBcHAiLCJhdWQiOiJodHRwOi8vYXBpc2VydmV
  yL2FwaS9vYXV0aC90b2tlbiIsImV4cCI6IjEzNDA0NTIxMjYiLCJpYXQiOiIxMzQwNDUxODI2In0=
```

5. Create a new string from the encoded JWT header from step 2, and the encoded JWT Claims Set from step 4, and append them as follows:

```
Base64URLEncode(JWT Header) + . + Base64URLEncode(JWT Claims Set)
```

This results in a string as follows:

```
eyJhbGciOiJSUzI1NiJ9.eyJpc3MiOiAiU2FtcGxlQ29uZmlkZW50aWFsQXBwIiwgImF1ZCI6ICJodHRw
 Oi8vYXBpc2VydmVyL2FwaS9vYXV0aC90b2tlbiIsICJleHAiOiAiMTM0MTM1NDYwNSIsICJpYXQiOiAi
 MTM0MTM1NDMwNSJ9
```

6. Sign the resulting string in step 5 using SHA256 with RSA. The signature must then be Base64url encoded. The signature is then concatenated with a `.` character to the end of the Base64url representation of the input string. The result is the following JWT (line breaks added for clarity):

```
{Base64url encoded header}.
{Base64url encoded claim set}.
```

This results in a string as follows:

```
eyJhbGciOiJSUzI1NiJ9.eyJpc3MiOiAiU2FtcGxlQ29uZmlkZW50aWFsQXBwIiwgImF1ZCI6ICJodHR
wOi8vYXBpc2VydmVyL2FwaS9vYXV0aC90b2tlbiIsICJleHAiOiAiMTM0MTM1NDYwNSIsICJpYXQiOiA
iMTM0MTM1NDMwNSJ9.ilWR8O8OlbQtT5zBaGIQjveOZFIWGTkdVC6LofJ8dN0akvvD0m7IvUZtPp4dx3
KdEDj4YcsyCEAPhfopUlZO3LE-iNPlbxB5dsmizbFIc2oGZr7Zo4IlDf92OJHq9DGqwQosJ-s9GcIRQk
-IUPF4lVy1Q7PidPWKR9ohm3c2gt8
```

**Requesting an access token**
The JWT bearer token should be sent in an HTTP `POST` to the Token Endpoint with the following parameters:

| Parameter | Description |
|---|---|
| `grant_type` | Required. Must be set to `urn:ietf:params:oauth:grant-type:jwt-bearer`. |
| `assertion` | Required. Must be set to the JWT bearer token, base64url-encoded. |
| `format` | Optional. Expected return format. The default is `json`. Possible values are:<br>• `urlencoded`<br>• `json`<br>• `xml` |

The following is an example POST request:

```
POST /api/oauth/token HTTP/1.1
Content-Length: 424
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Host: 192.168.0.48:8080
grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Ajwt-bearer&assertion=eyJhbGciOiJS
 UzI1NiJ9.eyJpc3MiOiAiU2FtcGxlQ29uZmlkZW50aWFsQXBwIiwgImF1ZCI6ICJodHRwOi8vYXBpc2Vy
 dmVyL2FwaS9vYXV0aC90b2tlbiIsICJleHAiOiAiMTM0MTM1NDYwNSIsICJpYXQiOiAiMTM0MTM1NDMwN
 SJ9.ilWR8O8OlbQtT5zBaGIQjveOZFIWGTkdVC6LofJ8dN0akvvD0m7IvUZtPp4dx3KdEDj4YcsyCEAPh
 fopUlZO3LE-iNPlbxB5dsmizbFIc2oGZr7Zo4IlDf92OJHq9DGqwQosJ-s9GcIRQk-IUPF4lVy1Q7PidP
 WKR9ohm3c2gt8
```

**Handling the response**
The API Gateway returns an access token if the JWT claim and access token request are properly formed, and the JWT has been signed by the private key matching the registered certificate for the client application in the Oracle Client Application Registry.

For example, a valid response is as follows:

```
HTTP/1.1 200 OK
Cache-Control: no-store
Content-Type: application/json
Pragma: no-cache
{
    "access_token": “O91G451HZ0V83opz6udiSEjchPynd2Ss9......",
    "token_type": "Bearer",
    "expires_in": "3600",
}
```

**Running the sample client**
The following Jython sample creates and sends a JWT Bearer token to the Authorization Server:

```
INSTALL_DIR/samples/scripts/oauth/jwt.py
```

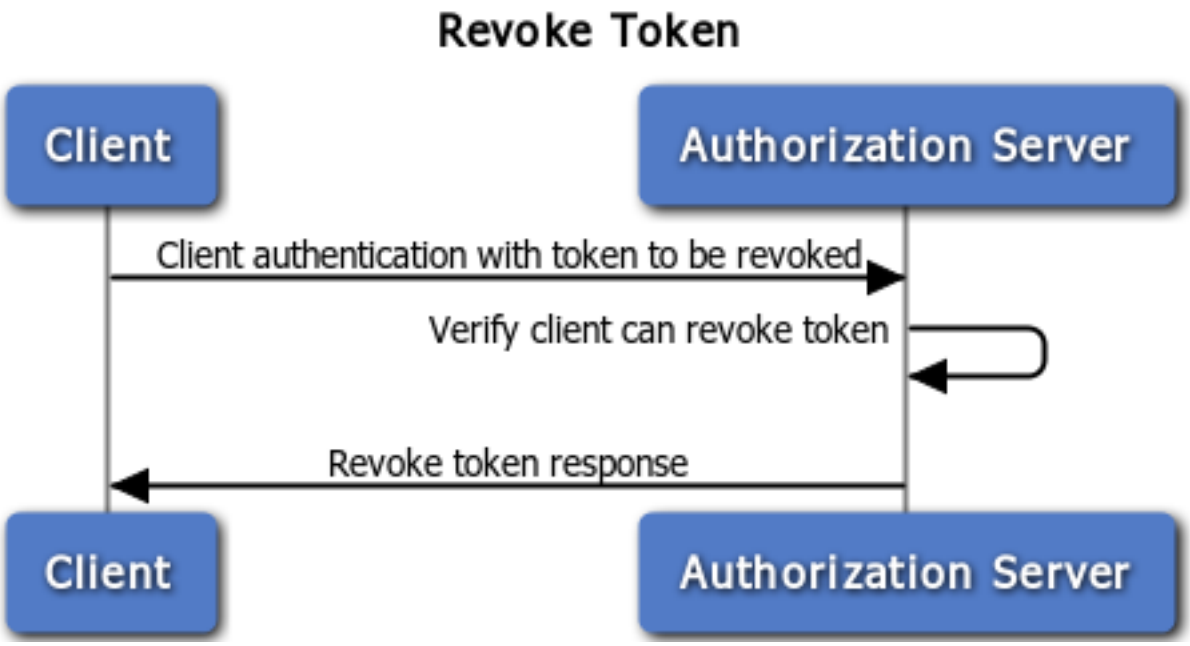
To run the sample, open a shell prompt at `INSTALL_DIR/samples/scripts`, and execute the following command:

```
> run oauth/jwt.py
```

**Further information**
For details on the API Gateway filter that supports this flow, see *Access Token Using JWT*.

## Revoke Token

In some cases a user may wish to revoke access given to an application. An access token can be revoked by calling the API Gateway revoke service and providing the access token to be revoked. A revoke token request causes the removal of the client permissions associated with the particular token to access the end-user's protected resources.




The endpoint for revoke token requests is as follows:

```
https://<API Gateway>:8089/api/oauth/revoke
```

The token to be revoked should be sent to the revoke token endpoint in an HTTP `POST` with the following parameter:

| Parameter | Description |
|---|---|
| `token` | Required. A token to be revoked (for example, `4eclEUX1N6oVIOoZBbaDTI977SV3T9KqJ3ayOvs4gqhGA4`). |

The following is an example POST request:

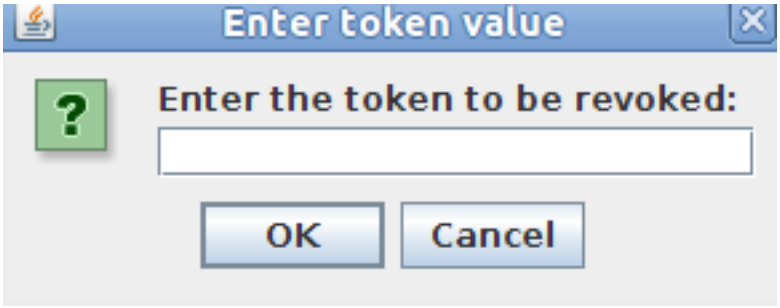
```
POST /api/oauth/revoke HTTP/1.1
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Host: 192.168.0.48:8080
Authorization: Basic U2FtcGxlQ29uZmlkZW50aWFsQXBwOjY4MDhkNGI2LWVmMDktNGIwZC04ZjI4LT
NiMDVkYTljNDhlYw==token=4eclEUX1N6oVIOoZBbaDTI977SV3T9KqJ3ayOvs4gqhGA4
```

**Running the sample client**
The following Jython sample client creates a token revoke request to the Authorization Server:

```
INSTALL_DIR/samples/scripts/oauth/revoke_token.py
```

To run the sample, open a shell prompt at `INSTALL_DIR/samples/scripts`, and execute the following command:

```
> run oauth/revoke_token.py
```




When the Authorization Server receives the token revocation request, it first validates the client credentials and verifies whether the client is authorized to revoke the particular token based on the client identity.

### Note

Only the client that was issued the token can revoke it.

The Authorization Server decides whether the token is an access token or a refresh token:

- If it is an access token, this token is revoked.
- If it is a refresh token, all access tokens issued for the refresh token are invalidated, and the refresh token is revoked.

**Response codes**
The following HTTP status response codes are returned:

- HTTP 200 if processing is successful.
- HTTP 401 if client authentication failed.
- HTTP 403 if the client is not authorized to revoke the token.

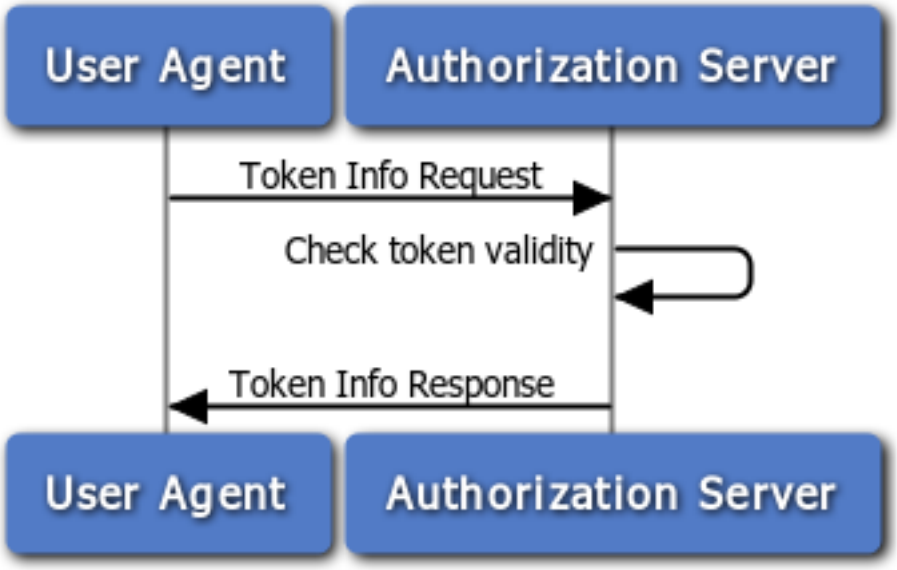
The following is an example response:

```
Token to be revoked: 3eXnUZzkODNGb9D94Qk5XhiV4W4gu9muZ56VAYoZiot4WNhIZ72D3
Revoking token..............
Response from revoke token request is: 200
Successfully revoked token
```

**Further information**
For details on the API Gateway filter that supports this flow, see *Revoke a Token*.

## Token Info Service

You can use the **Token Info Service** to validate that an access token issued by the API Gateway. A request to the `tokenInfo` service is an HTTP `GET` request for information in a specified OAuth 2.0 access token.




The endpoint for the token information service is as follows:

```
https://<apigateway>:8089/api/oauth/tokeninfo
```

Getting information about a token from the Authorization Server only requires a `GET` request to the tokeninfo endpoint. For example:

```
GET /api/oauth/tokeninfo HTTP/1.1
Host: 192.168.0.48:8080
access_token=4eclEUX1N6oVIOoZBbaDTI977SV3T9KqJ3ayOvs4gqhGA4
```

This request includes the following parameter:

| Parameter | Description |
|---|---|
| `access_token` | Required. A token that you want information about (for example: `4eclEUX1N6oVIOoZBbaDTI977SV3T9KqJ3ayOvs4gqhGA4`) |

The following example uses this parameter:

```
https://apigateway/api/oauth/tokeninfo?access_token=4eclEUX1N6oVIOoZBba
 DTI977SV3T9KqJ3ayOvs4gqhGA4
```

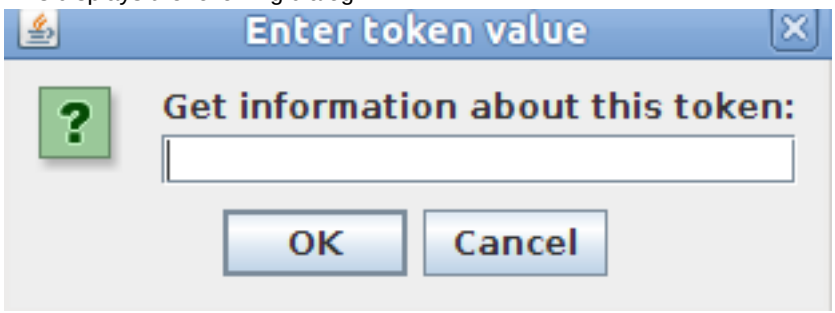
**Running the sample client**
The following Jython sample client creates a token revoke request to the Authorization Server:

```
INSTALL_DIR/samples/scripts/oauth/token_info.py
```

To run the sample, open a shell prompt at `INSTALL_DIR/samples/scripts`, and execute the following command:

```
> run oauth/token_info.py
```

This displays the following dialog:




When the Authorization Server receives the Token Info request, it first ensures the token is in its cache (EhCache or Database), and ensures the token is valid and has not expired.

The following is an example response:

```
Get token info for this token: BcYGjPOQSCrtbEc1F0ag8zf6OT9rCaMLiI1dYjFLT5zhxz3x5ScrdN
Response from token info request is: 200
**********************TOKEN INFO RESPONSE*********************************
Token audience received from authorization server: SampleConfidentialApp
Scopes user consented to: https://localhost:8090/auth/userinfo.email
Token expiry time: 3566
User id : admin
*************************************************************************
```

**Response codes**
The following HTTP Status codes are returned:

- 200 if processing is successful
- 400 on failure

The response is sent back as a JSON message. For example:

```
{
 "audience" : "SampleConfidentialApp",
   "user_id" : "admin",
   "scope" : "https://localhost:8090/auth/userinfo.email",
   "expires_in" : 2518
 }
```

You can get additional information about the access token using message attributes. For more details, see the section called “Querying OAuth 2.0 Message Attributes”.

**Further information**
For details on the API Gateway filter that supports this flow, see *OAuth Access Token Information*.

# OAuth Access Token Information

## Overview

The OAuth 2.0 **Access Token Information** filter is used to return a JSON description of the specified OAuth 2.0 access token. OAuth access tokens are used to grant access to specific resources in an HTTP service for a specific period of time (for example, photos on a photo sharing website). This enables users to grant third-party applications access to their resources without sharing all of their data and access permissions.

An OAuth access token can be sent to the Resource Server to access the protected resources of the Resource Owner (user). This token is a string that denotes a specific scope, lifetime, and other access attributes. For details on supported OAuth flows, see *API Gateway OAuth 2.0 Authentication Flows*.

## Access Token Info Settings

Configure the following fields on this tab:

**Token to verify can be found here:**
Click the browse button to select the location of the access token to verify (for example, in the default **OAuth Access Token Store**). To add a store, right-click **Access Token Stores**, and select **Add Access Token Store**. You can store tokens in a cache, in a relational database, or in an embedded Cassandra database. For more details, see the section called “Managing Access Tokens and Authorization Codes”.

**Where to get access token from?:**
Select one of the following:

- **In Query String**:
  This is the default setting. Defaults to the `access_token` parameter.
- **In a selector**:
  Defaults to the `${http.client.getCgiArgument('access_token')}` selector. For more details on API Gateway selectors, see the *API Gateway User Guide*.

## Monitoring

The settings on this tab configure service-level monitoring options such as whether to store usage metrics data to a database. This information can be used by the web-based API Gateway Manager tool to display service use, and by the API Gateway Analytics tool to produce reports on how the service is used.

- **Monitor service usage:**
  Select this option if you want to store message metrics for this service.
- **Monitor service usage per client:**
  Select this option if you want to generate reports monitoring which authenticated clients are calling which services.
- **Monitor client usage:**
  If you want to generate reports on authenticated clients, but are not interested in which services they are calling, select this option and deselect **Monitoring service usage per client**.
- **Which attribute is used to identify the client?:**
  Enter the message attribute to use to identify authenticated clients. The default is `authentication.subject.id`, which stores the identifier of the authenticated user (for example, the username or user's X.509 Distinguished Name).
- **Composite Context:**
  This setting enables you to select a service context as a composite context in which multiple service contexts are monitored during the processing of a message. This setting is not selected by default.

  For example, the API Gateway receives a message, and sends it to `serviceA` first, and then to `serviceB`. Monit-

oring is performed separately for each service by default. However, you can set a composite service context before `serviceA` and `serviceB` that includes both services. This composite service passes if both services complete successfully, and monitoring is also performed on the composite service context.

## Advanced

The settings on this tab include the following:

**Return additional Access Token parameters:**
Click **Add** to return additional access token parameters, and enter the **Name** and **Value** in the dialog. For example, you could enter `Department` in **Name**, and the following selector in **Value**:

```
${accesstoken.getAdditionalInformation().get("Department")
```

# Access Token Using Authorization Code

## Overview

The OAuth 2.0 **Access Token using Authorization Code** filter is used to get a new access token using the authorization code. This supports the OAuth 2.0 Authorization Code Grant or Web server authentication flow, which is used by applications that are hosted on a secure server. A critical aspect of this flow is that the server must be able to protect the issued client application's secret. For more details on supported OAuth flows, see *API Gateway OAuth 2.0 Authentication Flows*.

OAuth access tokens are used to grant access to specific resources in an HTTP service for a specific period of time (for example, photos on a photo sharing website). This enables users to grant third-party applications access to their resources without sharing all of their data and access permissions. An OAuth access token can be sent to the Resource Server to access the protected resources of the Resource Owner (user). This token is a string that denotes a specific scope, lifetime, and other access attributes.

## Application Validation

Configure the following fields on this tab:

**Use this store to validate the Authorization Code:**
Click the browse button to select the store in which to validate the authorization code (for example, in the default **Authz Code Store**). To add a store, right-click **Authorization Code Stores**, and select **Add Authorization Code Store**. You can store codes in a cache, in a relational database, or in an embedded Cassandra database. For more details, see the section called “Managing Access Tokens and Authorization Codes”.

**Find client application information from message:**
Select one of the following:

- **In Authorization Header**
  This is the default setting.
- **In Query String**:
  The **Client Id** defaults to `client_id`, and **Client Secret** defaults to `client_secret`.

## Access Token

Configure the following fields on the this tab:

**Access Token will be stored here:**
Click the browse button to select where to store the access token (for example, in the default **OAuth Access Token Store**). To add an access token store, right-click **Access Token Stores**, and select **Add Access Token Store**. You can store tokens in a cache, in a relational database, or in an embedded Cassandra database. For more details, see the section called “Managing Access Tokens and Authorization Codes”.

**Access Token Expiry (in secs):**
Enter the number of seconds before the access token expires. Defaults to `3600` (one hour).

**Access Token Length:**
Enter the number of characters in the access token. Defaults to `54`.

**Access Token Type:**
Enter the access token type. This provides the client with information required to use the access token to make a protected resource request. The client cannot use an access token if it does not understand the token type. Defaults to `Bearer`.

**Include Refresh Token:**
Select whether to include a refresh token. This is a token issued by the Authorization Server to the client that can be used to obtain a new access token. This setting is selected by default.

**Refresh Token Expiry (in secs):**
When **Include Refresh Token** is selected, enter the number of seconds before the refresh token expires. Defaults to `43200` (twelve hours).

**Refresh Token Length:**
When **Include Refresh Token** is selected, enter the number of characters in the refresh token. Defaults to `46`.

**Store additional Access Token parameters:**
Click **Add** to store additional access token parameters, and enter the **Name** and **Value** in the dialog (for example, `Department`, `Engineering`).

## Monitoring

The settings on this tab configure service-level monitoring options such as whether to store usage metrics data to a database. This information can be used by the web-based API Gateway Manager tool to display service use, and by the API Gateway Analytics tool to produce reports on how the service is used. For details on the fields on this tab, see the section called “Monitoring” in *OAuth Access Token Information*.

# Access Token Using Client Credentials

## Overview

The OAuth 2.0 **Access Token using Client Credentials** filter enables an OAuth client to request an access token using only its client credentials. This supports the OAuth 2.0 Client Credentials flow, which is used when the client application needs to directly access its own resources on the Resource Server. Only the client application's credentials or public/private key pair are used in the this flow. The Resource Owner's credentials are not required. For more details on supported OAuth flows, see *API Gateway OAuth 2.0 Authentication Flows*.

OAuth access tokens are used to grant access to specific resources in an HTTP service for a specific period of time (for example, photos on a photo sharing website). This enables users to grant third-party applications access to their resources without sharing all of their data and access permissions. An OAuth access token can be sent to the Resource Server to access the protected resources of the Resource Owner (user). This token is a string that denotes a specific scope, lifetime, and other access attributes.

## Application Validation

Configure the following fields on this tab:

**Find client application information from message:**
Select one of the following:

- **In Authorization Header:**
  This is the default setting.
- **In Query String:**
  The **Client Id** defaults to `client_id`, and **Client Secret** defaults to `client_secret`.

## Access Token

Configure the following fields on the this tab:

**Access Token will be stored here:**
Click the browse button to select where to store the access token (for example, in the default **OAuth Access Token Store**). To add an access token store, right-click **Access Token Stores**, and select **Add Access Token Store**. You can store tokens in a cache, in a relational database, or in an embedded Cassandra database. For more details, see the section called “Managing Access Tokens and Authorization Codes”.

**Access Token Expiry (in secs):**
Enter the number of seconds before the access token expires. Defaults to `3600` (one hour).

**Access Token Length:**
Enter the number of characters in the access token. Defaults to `54`.

**Access Token Type:**
Enter the access token type. This provides the client with information required to use the access token to make a protected resource request. The client cannot use an access token if it does not understand the token type. Defaults to `Bearer`.

**Include Refresh Token:**
Select whether to include a refresh token. This is a token issued by the Authorization Server to the client that can be used to obtain a new access token. This setting is selected by default.

**Refresh Token Expiry (in secs):**
When **Include Refresh Token** is selected, enter the number of seconds before the refresh token expires. Defaults to

`43200` (twelve hours).

**Refresh Token Length:**
When **Include Refresh Token** is selected, enter the number of characters in the refresh token. Defaults to `46`.

**Store additional Access Token parameters:**
Click **Add** to store additional access token parameters, and enter the **Name** and **Value** in the dialog (for example, `Department` and `Engineering`).

**Generate Token Scopes:**
When requesting a token from the Authorization Server, you can specify a parameter for the OAuth scopes that you wish to access. When scopes are sent in the request, you can select whether the access token is generated only if the scopes in the request match all or any scopes registered for the application. Alternatively, for extra flexibility you can get the scopes by calling out to a policy.

Select one of the following options to configure how access tokens are generated based on specified scopes:

- **Get scopes from a registered application**:
  Select whether the scopes must match **Any** or **All** of the scopes registered for the application in the Client Application Registry. Defaults to **Any**. If no scopes are sent in the request, the token is generated with the scopes registered for the application.
- **Get scopes by calling policy**:
  Select a pre-configured policy to get the scopes, and enter the attribute that stores the scopes in the **Scopes approved for token are stored in the attribute** textbox. Defaults to `scopes.for.token`. The configured filter requires the scopes as set of strings on the message whiteboard.

## Monitoring

The settings on this tab configure service-level monitoring options such as whether to store usage metrics data to a database. This information can be used by the web-based API Gateway Manager tool to display service use, and by the API Gateway Analytics tool to produce reports on how the service is used. For details on the fields on this tab, see the section called “Monitoring” in *OAuth Access Token Information*.

# Access Token Using JWT

## Overview

The OAuth 2.0 **Access Token using JWT** filter enables an OAuth client to request an access token using only a JSON Web Token (JWT). This supports the OAuth 2.0 JWT flow, which is used when the client application needs to directly access its own resources on the Resource Server. Only the client JWT token is used in this flow, the Resource Owner's credentials are not required. A JWT token is a JSON-based security token encoding that enables identity and security information to be shared across security domains. For more details on supported OAuth flows, see *API Gateway OAuth 2.0 Authentication Flows*.

OAuth access tokens are used to grant access to specific resources in an HTTP service for a specific period of time (for example, photos on a photo sharing website). This enables users to grant third-party applications access to their resources without sharing all of their data and access permissions. An OAuth access token can be sent to the Resource Server to access the protected resources of the Resource Owner (user). This token is a string that denotes a specific scope, lifetime, and other access attributes.

## Application Validation

Configure the following fields on this tab:

**Audience (aud) must contain the following URI:**
Enter the JWT `aud` (intended audience). The JWT must contain an `aud` URI that identifies the Authorization Server, or service provider domain, as an intended audience. The Authorization Server must also verify that it is an intended audience for the JWT. Defaults to `http://apiserver/api/oauth/token`.

## Access Token

Configure the following fields on the this tab:

**Access Token will be stored here:**
Click the browse button to select where to cache the access token (for example, in the default **OAuth Access Token Store**). To add an access token store, right-click **Access Token Stores**, and select **Add Access Token Store**. You can store tokens in a cache, in a relational database, or in an embedded Cassandra database. For more details, see the section called “Managing Access Tokens and Authorization Codes”

**Access Token Expiry (in secs):**
Enter the number of seconds before the access token expires. Defaults to `3600` (one hour).

**Access Token Length:**
Enter the number of characters in the access token. Defaults to `54`.

**Access Token Type:**
Enter the access token type. This provides the client with information required to use the access token to make a protected resource request. The client cannot use an access token if it does not understand the token type. Defaults to `Bearer`.

**Include Refresh Token:**
Select whether to include a refresh token. This is a token issued by the Authorization Server to the client that can be used to obtain a new access token. This setting is unselected by default.

**Refresh Token Expiry (in secs):**
When **Include Refresh Token** is selected, enter the number of seconds before the refresh token expires. Defaults to `43200` (twelve hours).

**Refresh Token Length:**

When **Include Refresh Token** is selected, enter the number of characters in the refresh token. Defaults to 46.

**Store additional Access Token parameters:**
Click **Add** to store additional access token parameters, and enter the **Name** and **Value** in the dialog (for example, `Department` and `Engineering`).

**Generate Token Scopes:**
When requesting a token from the Authorization Server, you can specify a parameter for the OAuth scopes that you wish to access. When scopes are sent in the request, you can select whether the access token is generated only if the scopes in the request match all or any scopes registered for the application. Alternatively, for extra flexibility, you can get the scopes by calling out to a policy.

Select one of the following options to configure how access tokens are generated based on specified scopes:

- **Get scopes from a registered application**:
  Select whether the scopes must match **Any** or **All** of the scopes registered for the application in the Client Application Registry. Defaults to **Any**. If no scopes are sent in the request, the token is generated with the scopes registered for the application.
- **Get scopes by calling policy**:
  Select a pre-configured policy to get the scopes, and enter the attribute that stores the scopes in the **Scopes approved for token are stored in the attribute** textbox. Defaults to `scopes.for.token`. The configured filter requires the scopes as set of strings on the message whiteboard.

## Monitoring

The settings on this tab configure service-level monitoring options such as whether to store usage metrics data to a database. This information can be used by the web-based API Gateway Manager tool to display service use, and by the API Gateway Analytics tool to produce reports on how the service is used. For details on the fields on this tab, see the section called “Monitoring” in *OAuth Access Token Information*.

# Authorization Code Flow

## Overview

The OAuth 2.0 **Authorization Code Flow** filter is used to consume OAuth authorization requests, and is also known as the **Authorization Request** filter. This filter supports the OAuth 2.0 Authorization Code Grant (Web server) authentication flow, which is used by applications hosted on a secure server. A critical aspect of this flow is that the server must be able to protect the issued client application's secret. The Web server flow is suitable for clients capable of interacting with the end-user's user-agent (typically a Web browser), and capable of receiving incoming requests from the Authorization Server (acting as an HTTP server). The Authorization Code Grant flow is also known as the *Three-Legged OAuth Flow*.

The OAuth 2.0 Authorization Code Grant flow is as follows:

1. The Web server redirects the user to the API Gateway acting as an Authorization Server to authenticate and authorize the server to access data on their behalf.
2. After the user approves access, the Web server receives a callback with an authorization code.
3. After obtaining the authorization code, the Web server passes back the authorization code to obtain an access token response.
4. After validating the authorization code, the API Gateway passes back a token response to the Web server.
5. After the token is granted, the Web server accesses their data.

OAuth access tokens are used to grant access to specific resources in an HTTP service for a specific period of time (for example, photos on a photo sharing website). This enables users to grant third-party applications access to their resources without sharing all of their data and access permissions. An OAuth access token can be sent to the Resource Server to access the protected resources of the Resource Owner (user). This token is a string that denotes a specific scope, lifetime, and other access attributes.

The OAuth 2.0 **Authorization Request** filter also supports the Implicit Grant (User Agent) flow. This is used by client applications (consumers) residing in the user's device (for example, in a browser using JavaScript, or from a mobile device or desktop application). These consumers cannot keep the client secret confidential (application password or private key).

For more details on supported OAuth flows, see *API Gateway OAuth 2.0 Authentication Flows*.

## Validation/Templates

Configure the following fields on this tab:

**Authorize Resource Owner:**
Select one of the following:

- **Use internal flow**
  Uses the internal API Gateway flow to authorize the Resource Owner. This is the default setting. The internal flow authenticates the user against the API Gateway user store, and redirects the user to the **Authorize Transaction** filter to use sample template files for login and Resource Owner scope authorization.

  ### Note

  If you wish to store additional information with the authorization code (for Authorization Code flow), or with an access token (for Implicit Grant flow), you must set additional parameters in the **Authorize Transaction** flow filter.

- **Call this policy**
  Click the browse button to select a policy to authorize the Resource Owner. You can use the **Policy will store sub-**

**ject in selector** text box to specify where the policy is stored. Defaults to the `${authentication.subject.id}` message attribute. For more details on selectors, see the *API Gateway User Guide*.

### Note

If you wish to store additional information with the authorization code (for Authorization Code flow), or with an access token (for Implicit Grant flow), you must set additional parameters in the **Authorization Code Flow** filter.

## Authz Code Details

Configure the following fields on the this tab:

**Authorization Code will be stored here:**
Click the browse button to select where to cache the access token (for example, in the default **Authz Code Store**). To add an access token store, right-click **Authorization Code Stores**, and select **Add Authorization Code Store**. You can store codes in a cache, in a relational database, or in an embedded Cassandra database. For more details, see the section called “Managing Access Tokens and Authorization Codes”.

**Location of Access Code redirect page:**
Enter the full path to the HTML page used for the access code HTTP redirect. Defaults to the following:

```
${environment.VDISTDIR}/samples/oauth/templates/showAccessCode.html
```

`VDISTDIR` specifies the directory in which the API Gateway is installed.

**Length:**
Enter the number of characters in the authorization code. Defaults to `30`.

**Expiry (in secs):**
Enter the number of seconds before the authorization code expires. Defaults to `600` (ten minutes).

**Additional parameters to store for this Authorization Code:**
If you wish to store additional metadata with the authorization code, click **Add**, and enter the **Name** and **Value** in the dialog (for example, `Department` and `Engineering`). When additional data is set, it is then available in the **Access Token using Authorization Code** filter when the authorization code is exchanged for an access token. You can also specify the fields in this table using selectors. For more details, see the *API Gateway User Guide*.

### Note

If you entered parameters for the authorization code and parameters for the access token, the data will be merged. Data in the **Access Token using Authorization Code** filter may overwrite parameters stored with the authorization code. For example, if you set `Name:John` and `Department:Engineering` in the **Authorization Request** filter, and set `Department:HR` in the **Access Token using Authorization Code** filter, the token is created with `Name:John` and `Department:HR`.

## Access Token Details

Configure the following fields on the this tab:

**Access Token will be stored here:**
Click the browse button to select where to cache the access token (for example, in the default `OAuth Access Token Store`). To add an access token store, right-click **Access Token Stores**, and select **Add Access Token Store**. You can store tokens in a cache, in a relational database, or in an embedded Cassandra database. For more details, see the

section called “Managing Access Tokens and Authorization Codes”.

**Expiry (in secs):**
Enter the number of seconds before the access token expires. Defaults to `3600` (one hour).

**Length:**
Enter the number of characters in the access token. Defaults to `54`.

**Type:**
Enter the access token type. This provides the client with information required to use the access token to make a protected resource request. The client cannot use an access token if it does not understand the token type. Defaults to `Bearer`.

**Additional parameters to store for this Access Token:**
Click **Add** to store additional access token parameters, and enter the **Name** and **Value** in the dialog (for example, `Department`, `Engineering`).

**Generate Token Scopes:**
When requesting a token from the Authorization Server, you can specify a parameter for the OAuth scopes that you wish to access. You can select whether the access token is generated only if the scopes in the request match all or any scopes registered for the application. Alternatively, for extra flexibility you can get the scopes by calling out to a policy.

Select one of the following options to configure how access tokens are generated based on specified scopes:

- **Get scopes from a registered application**:
  Select whether the scopes must match **Any** or **All** of the scopes registered for the application in the Client Application Registry. Defaults to **Any**. If no scopes are sent in the request, the token is generated with the scopes registered for the application.
- **Get scopes by calling policy**:
  Select a pre-configured policy to get the scopes, and enter the attribute that stores the scopes in the **Scopes approved for token are stored in the attribute** textbox. Defaults to `scopes.for.token`. The configured filter requires the scopes as set of strings on the message whiteboard.

## Monitoring

The settings on this tab configure service-level monitoring options such as whether to store usage metrics data to a database. This information can be used by the web-based API Gateway Manager tool to display service use, and by the API Gateway Analytics tool to produce reports on how the service is used.

**Monitoring Options**
For details on the **Monitoring Options** fields on this tab, see the section called “Monitoring” in *OAuth Access Token Information*.

**Record Outbound Transactions**
Select whether to record outbound message traffic. You can use this setting to override the **Record Outbound Transactions** setting on the **System Settings** -> **Traffic Monitor** screen. This setting is selected by default.

# Authorize Transaction

## Overview

The OAuth 2.0 **Authorize Transaction** filter is used to authorize the Resource Owner and grant (allow/deny) client access to the resources. This supports the OAuth 2.0 Authorization Code Grant or Web server authentication flow, which is used by applications hosted on a secure server. A critical aspect of this flow is that the server must be able to protect the issued client application's secret. The Web server flow is suitable for clients capable of interacting with the end-user's user-agent (typically a Web browser), and capable of receiving incoming requests from the Authorization Server (acting as an HTTP server).

For more details on supported OAuth flows, see *API Gateway OAuth 2.0 Authentication Flows*.

## Validation/Templates

Configure the following fields on this tab:

**HTML Templates:**
Specify the following templates for HTML forms:

- **Login Form:**
  Enter the full path to the HTML form that the Resource Owner can use to log in. Defaults to the following:

  ```
  ${environment.VDISTDIR}/samples/oauth/templates/login.html
  ```

- **Authorization Form:**
  Enter the full path to the HTML form that the Resource Owner can use to grant (allow/deny) client access to the resources. Defaults to the following:

  ```
  ${environment.VDISTDIR}/samples/oauth/templates/requestAccess.html
  ```

`VDISTDIR` specifies the directory in which the API Gateway is installed.

## Authz Code Details

Configure the following fields on the this tab:

**Authorization Code will be stored here:**
Click the browse button to select where to cache the access token (for example, in the default `Authz Code Store`). To add an access token store, right-click **Authorization Code Stores**, and select **Add Authorization Code Store**. You can store codes in a cache, in a relational database, or in an embedded Cassandra database. For more details, see the section called "Managing Access Tokens and Authorization Codes".

**Location of Access Code redirect page:**
Enter the full path to the HTML page used for the access code HTTP redirect. Defaults to the following:

```
${environment.VDISTDIR}/samples/oauth/templates/showAccessCode.html
```

**Length:**
Enter the number of characters in the authorization code. Defaults to `30`.

**Expiry (in secs):**
Enter the number of seconds before the authorization code expires. Defaults to `600` (ten minutes).

**Additional parameters to store for this Authorization Code:**
If you wish to store additional metadata with the authorization code, click **Add**, and enter the **Name** and **Value** in the dialog (for example, `Department` and `Engineering`). When additional data is set, it is then available in the **Access Token using Authorization Code** filter when the authorization code is exchanged for an access token. You can also specify the fields in this table using selectors. For more details, see the *API Gateway User Guide*.

### Note

If you entered parameters for the authorization code and parameters for the access token, the data will be merged. Data in the **Access Token using Authorization Code** filter may overwrite parameters stored with the authorization code. For example, if you set `Name:John` and `Department:Engineering` in the **Authorize Transaction** filter, and set `Department:HR` in the **Access Token using Authorization Code** filter, the token is created with `Name:John` and `Department:HR`.

## Access Token Details

Configure the following fields on the this tab:

**Access Token will be stored here:**
Click the browse button to select where to cache the access token (for example, in the default `OAuth Access Token Store`). To add an access token store, right-click **Access Token Stores**, and select **Add Access Token Store**. You can store tokens in a cache, in a relational database, or in an embedded Cassandra database. For more details, see the section called “Managing Access Tokens and Authorization Codes”.

**Expiry (in secs):**
Enter the number of seconds before the access token expires. Defaults to `3600` (one hour).

**Length:**
Enter the number of characters in the access token. Defaults to `54`.

**Type:**
Enter the access token type. This provides the client with information required to use the access token to make a protected resource request. The client cannot use an access token if it does not understand the token type. Defaults to `Bearer`.

**Additional parameters to store for this Access Token:**
Click **Add** to store additional access token parameters, and enter the **Name** and **Value** in the dialog (for example, `Department`, `Engineering`).

**Generate Token Scopes:**
When requesting a token from the Authorization Server, you can specify a parameter for the OAuth scopes that you wish to access. When scopes are sent in the request, you can select whether the access token is generated only if the scopes in the request match all or any scopes registered for the application. Alternatively, for extra flexibility you can get the scopes by calling out to a policy.

Select one of the following options to configure how access tokens are generated based on specified scopes:

- **Get scopes from a registered application**:
  Select whether the scopes must match **Any** or **All** of the scopes registered for the application in the Client Application Registry. Defaults to **Any**. If no scopes are sent in the request, the token is generated with the scopes registered for the application.
- **Get scopes by calling policy**:
  Select a pre-configured policy to get the scopes, and enter the attribute that stores the scopes in the **Scopes approved for token are stored in the attribute** textbox. Defaults to `scopes.for.token`. The configured filter requires the scopes as set of strings on the message whiteboard.

## Monitoring

The settings on this tab configure service-level monitoring options such as whether to store usage metrics data to a database. This information can be used by the web-based API Gateway Manager tool to display service use, and by the API Gateway Analytics tool to produce reports on how the service is used. For details on the fields on this tab, see the section called “Monitoring” in *OAuth Access Token Information*.

# Refresh Access Token

## Overview

The OAuth 2.0 **Refresh Access Token** filter enables an OAuth client to get a new access token using a refresh token. This filter supports the OAuth 2.0 Refresh Token flow. After the client consumer has been authorized for access, they can use a refresh token to get a new access token (session ID). This is only done after the consumer already has received an access token using either the Web Server or User-Agent flow. For more details on supported OAuth flows, see *API Gateway OAuth 2.0 Authentication Flows*.

## Application Validation

Configure the following fields on this tab:

**Find client application information from message:**
Select one of the following:

- **In Authorization Header:**
  This is the default setting.
- **In Query String:**
  The **Client Id** defaults to `client_id`, and **Client Secret** defaults to `client_secret`.

## Access Token

Configure the following fields on the this tab:

**Access Token will be stored here:**
Click the browse button to select where to cache the access token (for example, in the default **OAuth Access Token Store**). To add an access token store, right-click **Access Token Stores**, and select **Add Access Token Store**. You can store tokens in a cache, in a relational database, or in an embedded Cassandra database. For more details, see the section called “Managing Access Tokens and Authorization Codes”.

**Access Token Expiry (in secs):**
Enter the number of seconds before the access token expires. Defaults to `3600` (one hour).

**Access Token Length:**
Enter the number of characters in the access token. Defaults to `54`.

**Access Token Type:**
Enter the access token type. This provides the client with information required to use the access token to make a protected resource request. The client cannot use an access token if it does not understand the token type. Defaults to `Bearer`.

**Include Refresh Token:**
Select whether to include a refresh token. This is a token issued by the Authorization Server to the client that can be used to obtain a new access token. This setting is selected by default.

**Refresh Token Expiry (in secs):**
When **Include Refresh Token** is selected, enter the number of seconds before the refresh token expires. Defaults to `43200` (twelve hours).

**Refresh Token Length:**
When **Include Refresh Token** is selected, enter the number of characters in the refresh token. Defaults to `46`.

**Store additional Access Token parameters:**

Click **Add** to store additional access token parameters, and enter the **Name** and **Value** in the dialog (for example, `Department` and `Engineering`).

## Monitoring

The settings on this tab configure service-level monitoring options such as whether to store usage metrics data to a database. This information can be used by the web-based API Gateway Manager tool to display service use, and by the API Gateway Analytics tool to produce reports on how the service is used. For details on the fields on this tab, see the section called “Monitoring” in *OAuth Access Token Information*.

# Resource Owner Credentials

## Overview

The OAuth 2.0 **Resource Owner Credentials** filter is used to directly obtain an access token and an optional refresh token. This supports the OAuth 2.0 Resource Owner Password Credentials flow, which can be used as a replacement for an existing login when the consumer client already has the user's credentials. For more details on supported OAuth flows, see *API Gateway OAuth 2.0 Authentication Flows*.

OAuth access tokens are used to grant access to specific resources in an HTTP service for a specific period of time (for example, photos on a photo sharing website). This enables users to grant third-party applications access to their resources without sharing all of their data and access permissions. An OAuth access token can be sent to the Resource Server to access the protected resources of the Resource Owner (user). This token is a string that denotes a specific scope, lifetime, and other access attributes.

## Application Validation

Configure the following fields on this tab:

**Authenticate Resource Owner**
Select one of the following:

- **Authenticate credentials using this repository:**
  Select one of the following from the list:
  - `Simple Active Directory Repository`
  - `Local User Store`
- **Call this policy:**
  Click the browse button to select a policy to authenticate the Resource Owner. You can use the **Policy will store subject in selector** text box to specify where the policy is stored. Defaults to the `${authentication.subject.id}` message attribute. For more details on selectors, see the *API Gateway User Guide*.

**Find client application information from message:**
Select one of the following:

- **In Authorization Header:**
  This is the default setting.
- **In Query String:**
  The **Client Id** defaults to `client_id`, and **Client Secret** defaults to `client_secret`.

## Access Token

Configure the following fields on the this tab:

**Access Token will be stored here:**
Click the browse button to select where to cache the access token (for example, in the default **OAuth Access Token Store**). To add an access token store, right-click **Access Token Stores**, and select **Add Access Token Store**. You can store tokens in a cache, in a relational database, or in an embedded Cassandra database. For more details, see the section called "Managing Access Tokens and Authorization Codes".

**Access Token Expiry (in secs):**
Enter the number of seconds before the access token expires. Defaults to `3600` (one hour).

**Access Token Length:**
Enter the number of characters in the access token. Defaults to `54`.

**Access Token Type:**
Enter the access token type. This provides the client with information required to use the access token to make a protected resource request. The client cannot use an access token if it does not understand the token type. Defaults to `Bearer`.

**Include Refresh Token:**
Select whether to include a refresh token. This is a token issued by the Authorization Server to the client that can be used to obtain a new access token. This setting is selected by default.

**Refresh Token Expiry (in secs):**
When **Include Refresh Token** is selected, enter the number of seconds before the refresh token expires. Defaults to `43200` (twelve hours).

**Refresh Token Length:**
When **Include Refresh Token** is selected, enter the number of characters in the refresh token. Defaults to `46`.

**Store additional Access Token parameters:**
Click **Add** to store additional access token parameters, and enter the **Name** and **Value** in the dialog (for example, `Department` and `Engineering`).

**Generate Token Scopes:**
When requesting a token from the Authorization Server, you can specify a parameter for the OAuth scopes that you wish to access. You can select whether the access token is generated only if the scopes in the request match all or any scopes registered for the application. Alternatively, for extra flexibility you can get the scopes by calling out to a policy.

Select one of the following options to configure how access tokens are generated based on specified scopes:

- **Get scopes from a registered application**:
  Select whether the scopes must match **Any** or **All** of the scopes registered for the application in the Client Application Registry. Defaults to **Any**. If no scopes are sent in the request, the token is generated with the scopes registered for the application.
- **Get scopes by calling policy**:
  Select a pre-configured policy to get the scopes, and enter the attribute that stores the scopes in the **Scopes approved for token are stored in the attribute** textbox. Defaults to `scopes.for.token`. The configured filter requires the scopes as set of strings on the message whiteboard.

## Monitoring

The settings on this tab configure service-level monitoring options such as whether to store usage metrics data to a database. This information can be used by the web-based API Gateway Manager tool to display service use, and by the API Gateway Analytics tool to produce reports on how the service is used.

**Monitoring Options**
For details on the **Monitoring Options** fields on this tab, see the section called “Monitoring” in *OAuth Access Token Information*.

**Record Outbound Transactions**
Select whether to record outbound message traffic. You can use this setting to override the **Record Outbound Transactions** setting on the **System Settings** -> **Traffic Monitor** screen. This setting is selected by default.

# Revoke a Token

## Overview

The OAuth 2.0 **Revoke a Token** filter is used to revoke a specified OAuth 2.0 access or refresh token. A revoke token request causes the removal of the client permissions associated with the specified token used to access the user's protected resources. For more details on supported OAuth flows, see *API Gateway OAuth 2.0 Authentication Flows*.

OAuth access tokens are used to grant access to specific resources in an HTTP service for a specific period of time (for example, photos on a photo sharing website). This enables users to grant third-party applications access to their resources without sharing all of their data and access permissions. OAuth refresh tokens are tokens issued by the Authorization Server to the client that can be used to obtain a new access token.

## Revoke Token Settings

Configure the following fields on this tab:

**Token to be revoked can be found here:**
Click the browse button to select the cache to revoke the token from (for example, the default **OAuth Access Token Store**). To add an access token store, right-click **Access Token Stores**, and select **Add Access Token Store**. You can store tokens in a cache, in a relational database, or in an embedded Cassandra database. For more details, see the section called “Managing Access Tokens and Authorization Codes”.

**Find client application information from message:**
Select one of the following:

- **In Authorization Header:**
  This is the default setting.
- **In Query String:**
  The **Client Id** defaults to `client_id`, and **Client Secret** defaults to `client_secret`.

## Monitoring

The settings on this tab configure service-level monitoring options such as whether to store usage metrics data to a database. This information can be used by the web-based API Gateway Manager tool to display service use, and by the API Gateway Analytics tool to produce reports on how the service is used. For details on the fields on this tab, see the section called “Monitoring” in *OAuth Access Token Information*.

# Validate Access Token

## Overview

The OAuth 2.0 **Validate Access Token** filter is used to validate a specified access token contained in persistent storage. OAuth access tokens are used to grant access to specific resources in an HTTP service for a specific period of time (for example, photos on a photo sharing website). This enables users to grant third-party applications access to their resources without sharing all of their data and access permissions.

For more details on supported OAuth flows, see *API Gateway OAuth 2.0 Authentication Flows*.

## Configuration

Configure the following fields on this tab:

**Name:**
Enter a suitable name for this filter.

**Verify access token is in cache:**
Click the browse button to select the cache in which to verify access token (for example, in the default **OAuth Access Token Store**). To add an access token store, right-click **Access Token Stores**, and select **Add Access Token Store**. You can store tokens in a cache, in a relational database, or in an embedded Cassandra database. For more details, see the section called “Managing Access Tokens and Authorization Codes”.

**Location of access token:**
Select one of the following:

- **In Authorization Header with prefix:**
  The access token is in the Authorization header with the selected prefix. Defaults to `Bearer`. This is the default option.
- **In query string/form body with name:**
  The access token is in the HTTP query string with the name specified in the text box.
- **In Attribute:**
  The access token is in the API Gateway message attribute specified in the text box.

**Validate Scopes**:
Select one of the following options to configure how access tokens are accepted based on the validation of specified OAuth scopes:

- **Get scopes from list**:
  Select whether scopes match **Any** or **All** of the configured scopes in the table, and click **Add** to add an OAuth scope. The default scopes are found in `${http.request.uri}`.
- **Get scopes by calling policy**:
  Select a pre-configured policy to get the scopes, and enter the attribute that stores the scopes in the **Scopes required to access the resource are stored in the attribute** textbox. Defaults to `${scopes.required}`.

  Because the access token is in a message attribute on the whiteboard, you can use this policy to get the scopes for the access token and validate them against a scope list. In the event of a scope validation failure, you can set the `${scopes.required}` message attribute. This ensures that the end-user sees a list of required scopes to access the resource in the response.

For example, the default scopes used in the OAuth demos are as follows:

```
https://localhost:8090/auth/user.photos
https://localhost:8090/auth/userinfo.email
```