OAuth 2.0 Guide

/ ForgeRock Access Management 7.1.4

Latest update: 7.1.4

ForgeRock AS.
201 Mission St., Suite 2900
San Francisco, CA 94105, USA
+1 415-599-1100 (US)
www.forgerock.com

Abstract

Guide showing you how to use OAuth 2.0 with ForgeRock® Access Management (AM). ForgeRock Access Management provides intelligent authentication, authorization, federation, and single sign-on functionality.

Table of Contents

Overview

This guide covers concepts, configuration, and usage procedures for working with OAuth 2.0 and ForgeRock Access Management.

Quick Start

 AM as the Authorization Server Learn about OAuth 2.0 and how AM can take the role of the authorization server, what is supported, and the particulars of AM's implementation.	 Configure AM as an authorization server Configure AM as an OAuth 2.0 authorization server.	 OAuth 2.0 Grant Flows Discover the OAuth 2.0 flows and how to implement them in AM.
 OAuth 2.0 Endpoints Learn about the different endpoints AM exposes as an OAuth 2.0 authorization server.	 OAuth 2.0 Consent Allow OAuth 2.0 clients to skip consent, and users to save and revoke consent.	 OAuth 2.0 Scopes Learn what scopes are, how to configure them in AM, and how to create a custom scope validator tailored to your environment.

About ForgeRock Identity Platform™ Software

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

Chapter 1

AM as the Authorization Server

In the role of the authorization server, AM authenticates resource owners and obtains their authorization in order to return access tokens to clients.

+ *OAuth 2.0 Concepts*

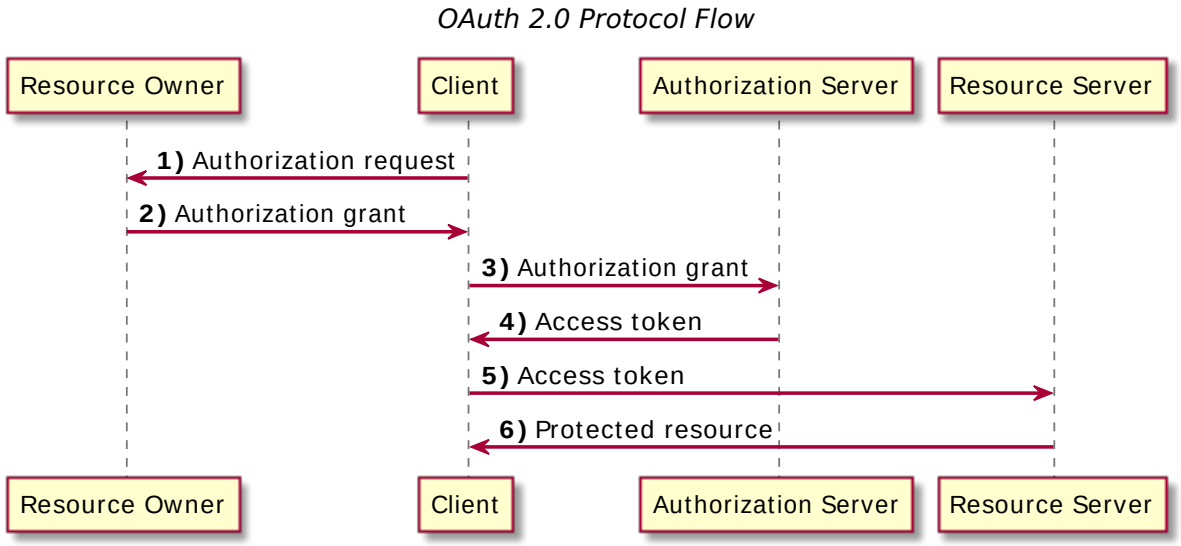
RFC 6749, *The OAuth 2.0 Authorization Framework* lets a third-party application obtain limited access to a resource (usually user data), either on behalf of the resource owner, or in the application's own behalf.

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

OAuth 2.0 Framework Actors

Actor	Description
Resource Owner (RO)	The owner of the resources. For example, a user that has several photos stored in a photo-sharing service. The resource owner uses a *user-agent*, usually a web-browser, to communicate with the client.
Client	The third-party application that wants to obtain access to the resources. The client makes requests on behalf of the resource owner and with their authorization. For example, a printing service that needs to access the resource owner's photos to print them. AM can act as a client.
Authorization Server (AS)	The authorization service that authenticates the resource owner and/or the client, issues access tokens to the client, and tracks their validity. Access tokens prove that the resource owner authorizes the client to act on their behalf over specific resources during a limited amount of time. AM can act as the authorization server.
Resource Server (RS)	The service hosting the protected resources. For example, a photo-sharing service. The resource server must be able to validate the tokens issued by the authorization server. A website protected by a web or a Java agent can act as the resource server.

The following sequence diagram demonstrates the basic OAuth 2.0 flow:



Before configuring OAuth 2.0 in your environment, ensure you are familiar with the OAuth 2.0 authorization framework and the RFCs, Internet-Drafts, and standards in the *Reference* that AM supports relating to OAuth 2.0.

When using AM as the authorization server, you can register confidential or public clients in the AM console, or clients can register themselves with AM dynamically. For more information, see "*Client Registration*".

As the authorization server, AM supports the following:

Grant Types

- **Authorization Code**
- **Implicit**
- **Resource Owner Password Credentials**
- **Client Credentials**
- **Device Flow**
- **SAML v2.0 Profile for Authorization Grant**
- **JWT Profile for OAuth 2.0 Authorization Grants**

For more information, see "*OAuth 2.0 Grant Flows*".

Client Authentication Standards

- **JWT Profile for OAuth 2.0 Client Authentication**
- **Mutual TLS**

For more information, see "*OAuth 2.0 Client Authentication*".

Token Exchange Standards

OAuth 2.0 Token Exchange

For more information, see "*OAuth 2.0 Token Exchange*".

Other OAuth 2.0 Standards

- **JWT Proof-of-Possession**

  For more information, see "JWK-Based Proof-of-Possession".

- **Certificate-based Proof-of-Possesion**

  For more information, see "Certificate-Bound Proof-of-Possession".

- **User Managed Access (UMA) 2.0**

  For more information, see the *ForgeRock Access Management UMA 2.0 Guide*.

- **OpenID Connect**

  For more information, see the ForgeRock Access Management OpenID Connect 1.0 Guide.

Tip

See the complete list with links to RFCs and Internet drafts in the *Reference*.

Moreover, AM as an authorization server supports the following capabilities:

- **Remote consent services**, which allows the consent-gathering part of an OAuth 2.0 flow to be handed off to a separate service.

  For more information, see "*Remote Consent*".

- **Dynamic Scopes**, which allows customization of how scopes are granted to the client regardless of the grant flow used. You can configure AM to grant scopes statically or dynamically:

  - Statically (Default). You configure several OAuth 2.0 clients with different subsets of scopes and resource owners are redirected to a specific client depending on the scopes required. As long as the resource owner can authenticate and the client can deliver the same or a subset of the requested scopes, AM issues the token with the scopes requested. Therefore, two different users requesting scopes A and B to the same client will always receive scopes A and B.

- Dynamically. You configure an OAuth 2.0 client with a comprehensive list of scopes and resource owners authenticate against it. When AM receives a request for scopes, AM's Authorization Service grants or denies access scopes dynamically by evaluating authorization policies at runtime. Therefore, two different users requesting scopes A and B to the same client can receive different scopes based on policy conditions.

For more information about granting scopes dynamically, see "*About Authorization and Policy Decisions*" and "*Dynamic OAuth 2.0 Authorization*" in the *Authorization Guide*.

Security Considerations

OAuth 2.0 messages involve credentials and access tokens that allow the bearer to retrieve protected resources. Therefore, do not let an attacker capture requests or responses. Protect the messages going across the network.

RFC 6749 includes a number of Security Considerations, and also requires Transport Layer Security (TLS) to protect sensitive messages. Make sure you read the section covering *Security Considerations*, and that you can implement them in your deployment.

Also, especially when deploying a mix of other clients and resource servers, take into account the points covered in the Internet-Draft, *OAuth 2.0 Threat Model and Security Considerations*, before putting your service into production.

OAuth 2.0 Sample Mobile Applications

To try the capabilities of AM as an authorization server, you can download the sample mobile application.

For information on downloading and building AM sample source code, see How do I access and build the sample code provided for AM (All versions)? in the *Knowledge Base*.

Related information:

- "About Token Storage Location"

About Token Storage Location

AM OAuth 2.0-related services are stateless unless otherwise indicated; they do not hold any token information local to the AM instances. Instead, they either store the OAuth 2.0/OpenID Connect tokens in the CTS token store, or present them to the client. This architecture lets you scale your AM infrastructure horizontally, since any server in the deployment can satisfy any token request.

OAuth 2.0 token storage location is a property of the OAuth 2.0 service, which is configured by realm. You can configure each realm to store tokens in the CTS token store, or to hand the tokens to the clients as required.

Both CTS-based and client-based token configurations support all OAuth 2.0 features present in AM. Before you decide to keep CTS-based tokens or to configure client-based tokens, consider the information in the following list:

CTS-Based OAuth 2.0 tokens (previously referred to as stateful tokens)

- The CTS token store is the authoritative source for the tokens. AM returns to the client a reference to the token, but that reference does not contain any of the token information. In the following example, the reference is stored in the `access_token` property:

```
{
  "access_token": "sbQZuveFumUDV5R1vVBl6QAGNB8",
  "scope": "write",
  "token_type": "Bearer",
  "expires_in": 3599
}
```

- CTS-based tokens are configured by default for all realms.
- Clients cannot access the tokens other than to introspect them, making tokens less vulnerable to tampering attacks.
- AM does not cache CTS-based tokens in memory. Therefore, every time a client presents a token ID in a request, AM checks if the token exists in the CTS token store (in case it has been revoked) and, if available, retrieves its information.

  Reading from and writing to the CTS token store incurs overhead for the CTS DS instances.
- If you need to add an additional layer of security for the stored tokens, consider one of the following alternatives:
  - Configure AM to encrypt the tokens before storing them in the CTS token store.
  - Configure DS to encrypt the CTS token store data.

  For more information about both options, see "*Managing CTS Tokens*" in the *Core Token Service Guide (CTS)*.
- Tokens can only be introspected using a call to the authorization server.

Client-Based OAuth 2.0 tokens (previously referred to as stateless tokens)

- AM returns the token to the client after successfully completing one of the grant flows. In the following example, the token is stored in the `access_token` property:

```
{
   "access_token":"eyJ0eXAiOiJKV1QiLOT05FIiwiYWxnQY1lIjoxNTM5MDEzMzYyLsbSI6Ii8iLCj....",
   "scope":"write",
   "token_type":"Bearer",
   "expires_in":3599
}
```

A decoded access token produces JSON structures similar to the following:

```
{
     typ: "JWT",
     zip: "NONE",
     alg: "HS256"
}
{
     sub: "(usr!myClient)",
     subname" "myClient",
     cts: "OAUTH2_STATELESS_GRANT",
     auditTrackingId: "f20f4099-5248-4399-a7f0-2d54d4020099-108676",
     iss: "https://openam.example.com:8443/openam/oauth2",
     tokenName: "access_token",
     token_type: "Bearer",
     authGrantId: "1LUgI8zcDWqcfEnnLdZDnNqA2wc",
     aud: "myClient",
     nbf: 1539075967,
     grant_type: "client_credentials",
     scope: [
      "write"
     ],
     auth_time: 1539075967,
     realm: "/alpha",
     exp: 1539079567,
     iat: 1539075967,
     expires_in: 3600,
     jti: "FTQT6eZkDhm6PHEaSthORoTLB80"
}
[signature]
```

- Token size may be a concern if tokens need to be sent in a header, since they are larger than the token reference returned for CTS-based tokens.

  The size of the client-based tokens also increases when you customize AM to store additional attributes in the tokens. You are responsible for ensuring that the size of the token does not exceed the maximum header size allowed by your end users' browsers.

- Can be configured by realm.

- Tokens are presented to the client after successfully completing an OAuth 2.0 grant flow. Therefore, tokens are vulnerable to tampering attacks and you should configure AM to **sign and encrypt** them.

- AM does not store the decrypt sequence of the token in memory, so decrypting and verifying tokens incurs overhead for the AM instances.

- *Token blacklisting* is a feature that maintains a list of revoked tokens and authorization codes stored in the CTS token store. This feature protects against replay attacks, and it is always enabled for client-based tokens.

  Every time a client presents a client-based token in a request, AM checks in the CTS token store if the token has been blacklisted (revoked). If it has not, then AM decrypts it to retrieve its information.

Note

Client-based refresh tokens have corresponding entries in a CTS whitelist, rather than a blacklist. When presenting a client-based refresh token, AM will check that a matching entry is found in the CTS whitelist, and prevent reissue if the record does not exist.

Adding a client-based OAuth 2.0 token to the blacklist will also remove associated refresh tokens from the whitelist.

- Clients can introspect the tokens without calling the authorization server. This can be advantageous in global deployments where keeping the CTS token store replication in sync fast enough to serve clients at any time by any server proves difficult.

For more information about configuring client-based OAuth 2.0, see "Configuring AM for Client-Based OAuth 2.0 Tokens".

Chapter 2

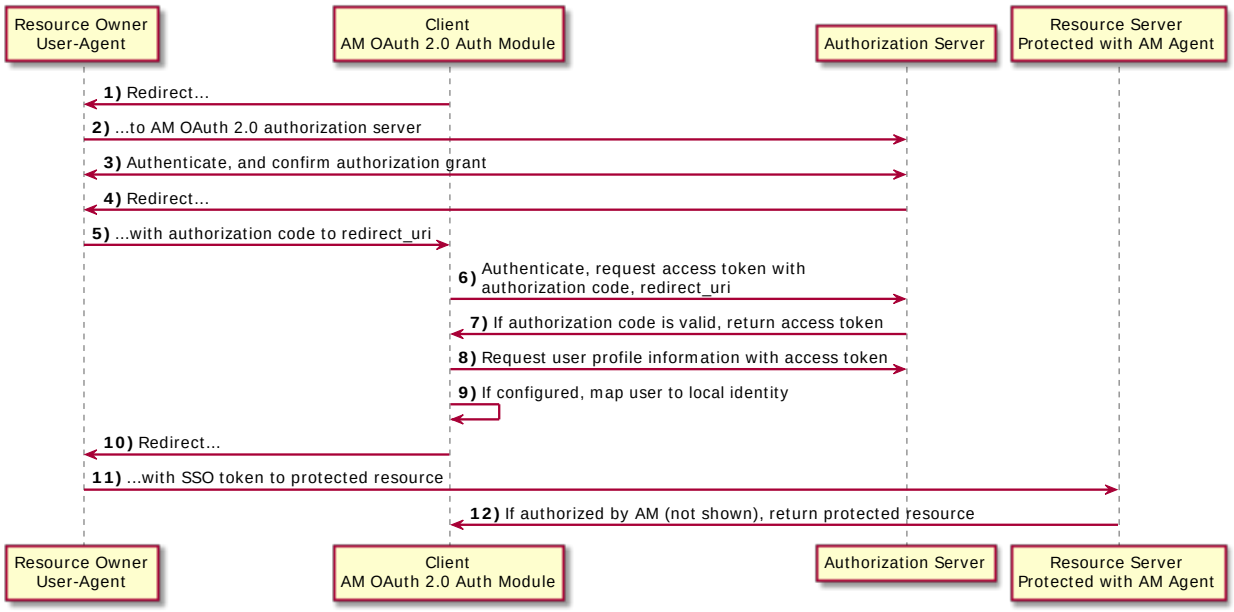
AM as Client and Resource Server

When AM functions as an OAuth 2.0 client, it provides an session after successfully authenticating the resource owner and obtaining authorization. This means the client can then access resources protected by agents.

To configure AM as an OAuth 2.0 client, use OAuth 2.0/OpenID Connect nodes or modules as part of the authentication journey.

The following sequence diagram shows how the client gains access to protected resources in the scenario where AM functions as both authorization server and client:

OAuth 2.0 Client and Authorization Server



As the OAuth 2.0 client functionality is implemented as an AM authentication module or nodes, you do not need to deploy your own resource server implementation when using AM as an OAuth 2.0 client. Use web or Java agents or IG to protect resources.

Using Your Own Client and Resource Server

AM returns bearer tokens as described in RFC 6750, *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. Notice in the following example JSON response to an access token request that AM returns a refresh token with the access token. The client can use the refresh token to get a new access token as described in RFC 6749:

```
{
  "expires_in": 599,
  "token_type": "Bearer",
  "refresh_token": "f6dcf133-f00b-4943-a8d4-ee939fc1bf29",
  "access_token": "f9063e26-3a29-41ec-86de-1d0d68aa85e9"
}
```

In addition to implementing your client, the resource server must also implement the logic for handling access tokens. The resource server can use the `/oauth2/introspect` endpoint to determine whether the access token is still valid, and to retrieve the scopes associated with the access token. For an example of the values returned by the endpoint, see "/oauth2/introspect".

AM is designed to let you plug in your own scopes implementation if the default implementation does not do what your deployment requires. See "Customizing OAuth 2.0 Scope Handling" for an example.

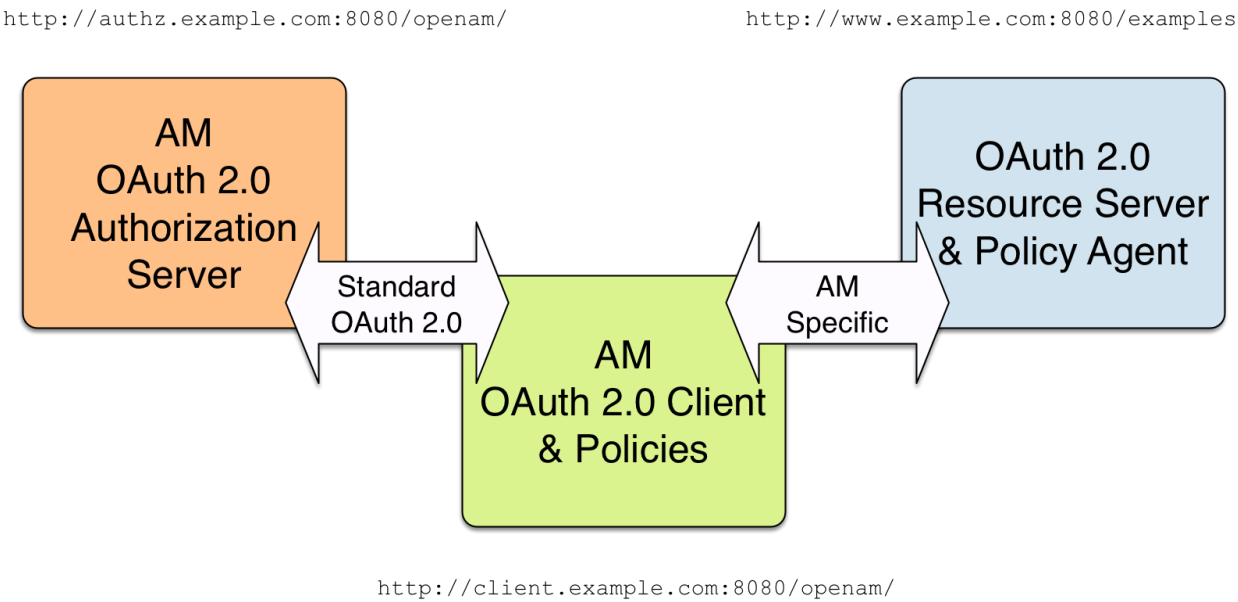
Related information:

- "*Social Authentication*" in the *Authentication and Single Sign-On Guide*
- "Configuring AM as an Authorization Server and Client"

Configuring AM as an Authorization Server and Client

This section takes a high-level look at how to set up AM both as an OAuth 2.0 authorization server and also as an OAuth 2.0 client, in order to protect resources on a resource server by using an AM web agent.

Authorization Server, Client, and Resource Server



The example in this section uses three servers, `http://authz.example.com:8080/openam` as the OAuth 2.0 authorization server, `http://client.example.com:8080/openam` as the OAuth 2.0 client, which also handles policy, `http://www.example.com:8080/` as the OAuth 2.0 resource server protected with an AM web agent where the resources to protect are deployed in Apache Tomcat. The two AM servers communicate using OAuth 2.0. The web agent on the resource server communicates with AM as agents normally do, using AM specific requests. The resource server in this example does not need to support OAuth 2.0.

The high-level configuration steps are as follows:

1. On the AM server that you will configure to act as an OAuth 2.0 client, configure an agent profile, and the policy used to protect the resources.

   On the web server or application container that will act as an OAuth 2.0 resource server, install and configure an AM web agent.

   Make sure that you can access the resources when you log in through an authentication module that you know is working, such as the default DataStore authentication module.

   In this example, you would try to access `http://www.example.com:8080/examples/`. The web agent should redirect you to the AM login page. After you log in successfully as a user with access rights to the resource, AM should redirect you back to `http://www.example.com:8080/examples/`, and the web agent should allow access.

Fix any problems you have in accessing the resources before you try to set up access through an OAuth 2.0 or OpenID Connect authentication module.

2. Configure one AM server as an OAuth 2.0 authorization service, which is described in "*Authorization Server Configuration*".
3. Configure the other AM server, the one with the agent profile and policy, as an OAuth 2.0 client, by setting up an OAuth 2.0 or OpenID Connect authentication module according to "Social Authentication Modules" in the *Authentication and Single Sign-On Guide*.
4. On the authorization server, register the OAuth 2.0 or OpenID Connect authentication module as an OAuth 2.0 client, which is described in "*Client Registration*".
5. Log out and access the protected resources to see the process in action.

Example: Protecting a Web Site With OAuth 2.0

This example pulls everything together (except security considerations), using AM servers both as the OAuth 2.0 authorization server, and also as the OAuth 2.0 client, with an AM web or Java agent on the resource server requesting policy decisions from AM as OAuth 2.0 client. In this way, any server protected by an agent that is connected to an AM OAuth 2.0 client can act as an OAuth 2.0 resource server:

1. On the AM server that will be configured as an OAuth 2.0 client, set up an AM web or Java agent and policy in the Top Level Realm, `/`, to protect resources.

   See the *ForgeRock Web Agents User Guide* or the *ForgeRock Java Agents User Guide* for instructions on installing an agent. This example relies on the Tomcat Java agent, configured to protect resources in Apache Tomcat (Tomcat) at `http://www.example.com:8080/`.

   The policies for this example protect the Tomcat examples under `http://www.example.com:8080/examples/`, allowing GET and POST operations by all authenticated users. For more information, see "*Dynamic OAuth 2.0 Authorization*" in the *Authorization Guide*.

   After setting up the web or Java agent and the policy, you can make sure everything is working by attempting to access a protected resource, in this case, `http://www.example.com:8080/examples/`. The agent should redirect you to AM to authenticate with the default authentication module, where you can login as user `demo` password `Ch4ng31t`. After successful authentication, AM redirects your browser back to the protected resource and the Java agent lets you get the protected resource, in this case, the Tomcat examples top page.

Accessing the Apache Tomcat Examples

2. On the AM server to be configured as an OAuth 2.0 authorization server, configure AM's OAuth 2.0 authorization service as described in "*Authorization Server Configuration*".

   The authorization endpoint to protect in this example is at `http://authz.example.com:8080/openam/oauth2/realms/root/authorize`.

3. On the AM server to be configured as an OAuth 2.0 client, configure an AM OAuth 2.0 or OpenID Connect social authentication module instance for the Top Level Realm:

   Under Realms > Top Level Realm > Authentication > Modules, click Add Module. Name the module `OAuth2`, and select the Social Auth OAuth2 type, then click Create. The module configuration page appears. This page offers numerous options. The key settings for this example are the following:

   Client Id

   This is the client identifier used to register your client with AM's authorization server, and then used when your client must authenticate to AM.

   Set this to `myClientID` for this example.

   Client Secret

   This is the client password used to register your client with AM's authorization server, and then used when your client must authenticate to AM.

   Set this to `password` for this example. Make sure you use strong passwords when you actually deploy OAuth 2.0.

   Authentication Endpoint URL

   In this example, `http://authz.example.com:8080/openam/oauth2/realms/root/authorize`.

   This AM endpoint can take additional parameters. In particular, you must specify the realm if the AM OAuth 2.0 provider is configured for a subrealm rather than for the Top Level Realm.

   When making a REST API call, specify the realm in the path component of the endpoint. You must specify the entire hierarchy of the realm, starting at the Top Level Realm. Prefix each realm in the hierarchy with the `realms/` keyword. For example `/realms/root/realms/customers/realms/europe` For example `/realms/root/realms/alpha`.

   For example, if the OAuth 2.0 provider is configured for the realm `customers` within the Top Level Realm, then use the following URL: `http://authz.example.com:8080/openam/oauth2/realms/root/realms/customers/authorize`.

   The `/oauth2/authorize` endpoint can also take `module` and `service` parameters. Use either as described in "*Authenticating (Browser)*" in the *Authentication and Single Sign-On Guide*, where `module` specifies the authentication module instance to use or `service` specifies the authentication chain to use when authenticating the resource owner.

Access Token Endpoint URL

In this example, `http://authz.example.com:8080/openam/oauth2/realms/root/access_token`.

This AM endpoint can take additional parameters. In particular, you must specify the realm if the AM OAuth 2.0 provider is configured for a subrealm rather than the Top Level Realm (/).

When making a REST API call, specify the realm in the path component of the endpoint. You must specify the entire hierarchy of the realm, starting at the Top Level Realm. Prefix each realm in the hierarchy with the `realms/` keyword. For example `/realms/root/realms/customers/realms/europe` For example `/realms/root/realms/alpha`.

For example, if the OAuth 2.0 provider is configured for the realm `/customers`, then use the following URL: `http://authz.example.com:8080/openam/oauth2/realms/root/realms/customers/access_token`.

User Profile Service URL

In this example, `http://authz.example.com:8080/openam/oauth2/realms/root/tokeninfo`.

Scope

In this example, `cn`.

The demo user has common name `demo` by default, so by setting this to `cn|Read your user name`, AM can get the value of the attribute without the need to create additional identities, or to update existing identities. The description, `Read your user name`, is shown to the resource owner in the consent page.

Subject Property

In this example, `cn`.

Proxy URL

The client redirect URL, which in this example is `http://client.example.com:8080/openam/oauth2c/OAuthProxy.jsp`.

Account Mapper

In this example, `org.forgerock.openam.authentication.modules.common.mapping.JsonAttributeMapper`.

Account Mapper Configuration

In this example, `cn=cn`.

Attribute Mapper

In this example, `org.forgerock.openam.authentication.modules.common.mapping.JsonAttributeMapper`.

Attribute Mapper Configuration

In this example, `cn=cn`.

Create account if it does not exist

In this example, disable this functionality.

AM can create local accounts based on the account information returned by the authorization server.

4. On the AM server configured to act as an OAuth 2.0 authorization server, register the Social Auth OAuth2 authentication module as an OAuth 2.0 confidential client, which is described in "*Client Registration*".

   Under Realms > Top Level Realm > Applications > OAuth 2.0 > `myClientID`, adjust the following settings:

   Client type

   In this example, `confidential`. AM protects its credentials as an OAuth 2.0 client.

   Redirection URIs

   In this example, `http://client.example.com:8080/openam/oauth2c/OAuthProxy.jsp`.

   If any Redirection URI scheme, host, or port differs from that of AM, add it to the global validation service to ensure that it is pre-approved, as described in "Configuring Success and Failure Redirection URLs" in the *Authentication and Single Sign-On Guide*. Otherwise, AM rejects the redirection URI, even if it matches the client profile, and redirection fails.

   Scopes

   In this example, `cn`.

5. Before you try it out, on the AM server configured to act as an OAuth 2.0 client, you must make the following additional change to the configuration.

   Your AM OAuth 2.0 client authentication module is not part of the default chain, and therefore AM does not call it unless you specifically request the OAuth 2.0 client authentication module.

   To cause the Java agent to request your OAuth 2.0 client authentication module explicitly, navigate to your *agent profile configuration*, in this case Realms > Top Level Realm > Applications > Agents > Java > *Agent Name* > AM Services > AM Login URL, and add `http://client.example.com:8080/openam/XUI/?realm=/&module=OAuth2`, moving it to the top of the list.

   Save your work.

This ensures that the Java agent directs the resource owner to AM with the instructions to authenticate using the `OAuth2` authentication module.

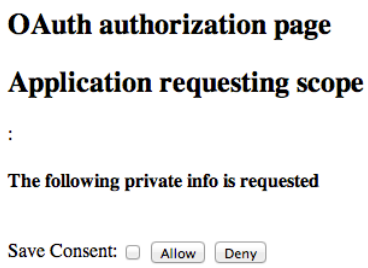
6. Try it out.

   First, make sure you are logged out of AM. For example, by browsing to the logout URL, in this case `http://client.example.com:8080/openam/XUI/?realm=/#logout`.

   Next attempt to access the protected resource, in this case `http://www.example.com:8080/examples/`.

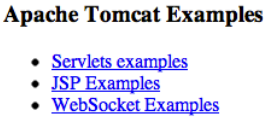
   If everything is set up properly, the Java agent redirects your browser to the login page of AM with `module=OAuth2` among other query string parameters. After you authenticate (for example, as user `demo`, password `Ch4ng31t`), AM displays an authorization decision page.

Presenting Authorization Decision Page to Resource Owner



When you click Allow, the authorization service creates an SSO session, and redirects the client back to the resource, thus letting the client access the protected resource. If you configured an attribute on which to store the saved consent decision, and you choose to save the consent decision for this authorization, then AM can use that saved decision to avoid prompting you for authorization next time the client accesses the resource, but only ensure that you have authenticated and have a valid session.

Successfully Accessing the Apache Tomcat Examples



Chapter 3

Authorization Server Configuration

Configure the OAuth2 Provider Service in a realm to expose the endpoints specified in "*OAuth 2.0 Endpoints*" and "*OAuth 2.0 Administration and Supporting REST Endpoints*".

To Configure the OAuth 2.0 Provider Service

1. In the AM console, go to Realms > *Realm Name* > Services, and click on the Add a Service button.
2. From the drop-down menu, select the OAuth2 Provider service. Then, click the Create button without filling any other field.

   The OAuth 2.0 provider page appears.
3. Go to the Advanced tab.
4. Configure the Grant Types that clients will be able to use to request access, refresh, and ID tokens.

   + *Grant Types Reference*

   ```
   Implicit
   SAML2
   Refresh Token
   Resource Owner Password Credentials
   Client Credentials
   Device Code
   Authorization Code
   Back Channel Request
   UMA
   JWT Bearer
   Token Exchange
   ```

   Related information:

   - "*OAuth 2.0 Grant Flows*"
   - "*OpenID Connect Grant Flows*" in the *OpenID Connect 1.0 Guide*
   - "*The UMA Grant Flow*" in the *User-Managed Access (UMA) 2.0 Guide*

- "*OAuth 2.0 Token Exchange*"

5. Configure the Response Type Plugins you need in your environment based on the grant type flows you will allow in your environment. Response plugins let the provider issue access tokens, ID tokens, authorization codes, and others, when the client requests a flow that interacts with the `/oauth2/authorize` endpoint.

   + *Response Type Plugin Reference*

```
code|org.forgerock.oauth2.core.AuthorizationCodeResponseTypeHandler
id_token|org.forgerock.openidconnect.IdTokenResponseTypeHandler
device_code|org.forgerock.oauth2.core.TokenResponseTypeHandler
token|org.forgerock.oauth2.core.TokenResponseTypeHandler
none|org.forgerock.oauth2.core.NoneResponseTypeHandler
```

   - The `id_token` and `none` response types are used in OpenID Connect flows.
   - The `code` response type is used in the OAuth 2.0 and OpenID Connect Authorization Code grant flows.
   - The `device_code` response type was used for the Device grant flow, but it is not required in the current implementation and will be removed in a future release of AM.
   - The `token` response type is used in the OAuth 2.0 and OpenID Connect Implicit flows.

6. (Optional) For configuration options, see "Additional Configuration".

Additional Configuration

The OAuth 2.0 provider is highly configurable:

- To access the OAuth 2.0 provider configuration in the AM console, go to Realms > *Realm Name* > Services, and then select OAuth2 Provider.
- To adjust global defaults, in the AM console, go to Configure > Global Services, and then click OAuth2 Provider.

See the "OAuth2 Provider" in the *Reference* reference section for details on each of the fields in the provider.

OAuth 2.0 Provider Configuration Options

Task	Resources
Configure the authorization server to issue refresh tokens	"*Refresh Tokens*"

Task	Resources
Learn why refresh tokens are useful in your environment, how to configure AM to issue them, and how to request them.	
Adjust the lifetimes of tokens and codes If necessary, adjust the lifetimes for authorization codes (a lifetime of 10 minutes or less is recommended in RFC 6749), access tokens, and refresh tokens. Configure them on the Core tab of the provider.	N/A.
Configure a custom scope validator implementation Keep the default scope implementation, whereby scopes are taken to be resource owner profile attribute names, unless you have a custom scope validator implementation. If you have a custom scope validator implementation, copy it to the AM classpath (for example, `/path/to/tomcat/webapps/openam/WEB-INF/lib/`) and specify the class name in the Scope Implementation Class field.	"Customizing OAuth 2.0 Scope Handling"
Configure the OAuth 2.0 provider for token exchange Following the OAuth 2.0 Token Exchange specification, the provider can let your clients exchange tokens.	"*OAuth 2.0 Token Exchange*"
Configure the OAuth 2.0 service to provide scopes dynamically The OAuth 2.0 provider can leverage the AM Authorization service to grant or deny scopes dynamically.	"*Dynamic OAuth 2.0 Authorization*" in the *Authorization Guide*
Configure a custom response plugin To configure a custom response type plugin, put it on the AM classpath, and then add the custom response types and the plugin class names to the list of Response Type Plugins field, on the Advanced tab.	N/A
Decide how scopes appear in the consent pages To change how scopes appear, configure the Client Registration Scope Whitelist field on the Advanced tab of the OAuth 2.0 provider. Scopes may be entered as simple strings or pipe-separated strings representing the internal scope name, locale, and localized description. For example: `read|en|Permission to view email messages in your account`.	"*About Scopes*"
Decide how to manage consent You can: • Allow users to save consent so the OAuth 2.0 provider remembers their consented scopes.	"*About Consent*"

Task	Resources
• Allow clients to skip consent so no consent page is displayed to the resource owners. • Allow clients to revoke consent.	
Configure a remote consent server This is useful, for example, when your environment must hand off the consent-gathering part of the OAuth 2.0 flows to a separate service.	"*Remote Consent*"
Configure the attribute AM uses to retrieve the user profile This is useful, for example, in cases where the resource owner should log in with their email address instead of with a username.	"To Change the Attribute Used to Retrieve the User Profile"
Configure client-based tokens Configure client-based tokens so that resource servers can directly introspect the tokens without making a call to AM.	"Configuring AM for Client-Based OAuth 2.0 Tokens"
Configure OpenID-Connect specific options UMA providers also use these options.	"*OpenID Provider Configuration*" in the *OpenID Connect 1.0 Guide*

To Change the Attribute Used to Retrieve the User Profile

If you use an external identity repository where resource owners log in not with their user ID, but instead with their mail address or some other profile attribute, you must configure AM authentication to allow it.

For example, to configure AM so OAuth 2.0 resource owners can log in using their email address, stored on the LDAP profile attribute, `mail`, perform the following steps:

1. On the OAuth2 provider Advanced tab, add the LDAP profile attribute to the User Profile Attribute(s) the Resource Owner is Authenticated On list, and save your changes.

2. Navigate to Realms > *Realm Name* > Identity Stores > *Identity Store Name* > Authentication Configuration.

3. Set the value of the Authentication Naming Attribute field to the LDAP attribute required. For example, `mail`.

   Warning

   If you change the value of Authentication Naming Attribute after you have deployed and configured AM, you must update or recreate all existing identities to refresh user DNs.

Failure to do so could result in unsuccessful authentication or risk of impersonation attacks.

4. Create an LDAP authentication module or an LDAP decision node to use with the identity repository.

   In both cases, configure the following fields:

   a. In the Attribute Used to Retrieve User Profile field, set the attribute to `mail`.

   b. In the Attributes Used to Search for a User to be Authenticated list, add the `mail` attribute.

   c. Save your changes.

5. Ensure the resource owners use the authentication tree or chain where you configured the LDAP module or node.

   Specify the chain or tree by using one or more of the methods below. AM checks for the configured value in the following order, using the first value found:

   1. For a specific access token REST request.

      Set the `auth_chain` parameter.

   2. Individually for a realm, overriding the realm-level setting below.

      Navigate to Realms > *Realm Name* > Services > OAuth2 Provider > Advanced, and set the Password Grant Authentication Service property.

   3. Individually for a realm.

      Navigate to Realms > *Realm Name* > Authentication > Settings > Core, and set the Organization Authentication Configuration property.

   4. Globally, for all realms.

      Navigate to Configure > Authentication > Core Attributes > Core, and set the Organization Authentication Configuration property.

   For more information, see Configure Sensible Default Authentication Services in the *Security Guide*.

   For more information about authentication trees and chains, see "*Configuring AM for Authentication*" in the *Authentication and Single Sign-On Guide*.

Configuring AM for Client-Based OAuth 2.0 Tokens

When configured for client-based tokens, AM returns a token (instead of the token reference it returns when configured for CTS-based tokens) to the client after successfully completing one of the grant flows. For more information about client-based and CTS-based tokens, see "About Token Storage Location".

To configure client-based tokens, perform the following tasks:

- Enable client-based tokens
- Configure client-based token blacklisting
- Configure AM to encrypt client-based tokens
- Configure AM to sign client-based tokens

Enabling Client-Based OAuth 2.0 Tokens

Perform the steps in the following procedure to configure AM to issue client-based access and refresh tokens:

To Enable Client-Based OAuth 2.0 Tokens

1. In the AM console, go to Realms > *Realm Name* > Services > OAuth2 Provider.
2. On the Core tab, enable Use Client-Based Access & Refresh Tokens.
3. (Optional) Enable Issue Refresh Tokens and/or Issue Refresh Tokens on Refreshing Access Tokens.
4. Save your changes.
5. Configure client-based token blacklisting. For more information, see "Configuring Client-Based OAuth 2.0 Token Blacklisting".
6. Configure either client-based token signature or client-based token encryption.

   Token signature is enabled by default when client-based tokens are enabled. By default, token signature is configured using a demo key that you must change in production environments. Enabling token encryption disables token signing as encryption is performed using direct symmetric encryption.

   For more information, see "Configuring Client-Based OAuth 2.0 Token Encryption" and "Configuring Client-Based OAuth 2.0 Token Digital Signatures".

   Client-based access and refresh tokens are ready for use.

Configuring Client-Based OAuth 2.0 Token Blacklisting

AM provides a blacklisting feature that prevents client-based tokens from being reused if the authorization code has been replayed or tokens have been revoked by either the client or resource owner.

Note

Client-based refresh tokens have corresponding entries in a CTS whitelist, rather than a blacklist. When presenting a client-based refresh token AM will check that a matching entry is found in the CTS whitelist, and prevent reissue if the record does not exist.

Adding a client-based OAuth 2.0 token to the blacklist will also remove associated refresh tokens from the whitelist.

To Configure Client-Based OAuth 2.0 Token Blacklisting

Perform the following steps to configure client-based token blacklisting:

1. In the AM console, go to Configure > Global Services > Global > OAuth2 Provider.
2. Under Global Attributes, enter the number of blacklisted tokens in the Token Blacklisting Cache Size field.

   Token Blacklisting Cache Size determines the number of blacklisted tokens to cache in memory to speed up blacklist checks. You can enter a number based on the estimated number of token revocations that a client will issue (for example, when the user gives up access or an administrator revokes a client's access).

   Default: 10000
3. In the Blacklist Poll Interval field, enter the interval in seconds for AM to check for token blacklist changes from the CTS data store.

   The longer the polling interval, the more time a malicious user has to connect to other AM servers in a cluster and make use of a stolen OAuth v2.0 access token. Shortening the polling interval improves the security for revoked tokens but might incur a minimal decrease in overall AM performance due to increased network activity.

   Default: 60 seconds
4. In the Blacklist Purge Delay field, enter the length of time in minutes that blacklist tokens can exist before being purged beyond their expiration time.

   When client-based token blacklisting is enabled, AM tracks OAuth v2.0 access tokens over the configured lifetime of those tokens plus the blacklist purge delay. For example, if the access token lifetime is set to 6000 seconds and the blacklist purge delay is one minute, then AM tracks the access token for 101 minutes. You can increase the blacklist purge delay if you expect system

clock skews in an AM server cluster to be greater than one minute. There is no need to increase the blacklist purge delay for servers running a clock synchronization protocol, such as Network Time Protocol.

Default: 1 minute

5. Click Save to apply your changes.

Configuring Client-Based OAuth 2.0 Token Encryption

To protect OAuth 2.0 client-based access and refresh tokens, AM supports encrypting their JWTs using AES authenticated encryption. Since this encryption also protects the integrity of the JWT, you only need to configure AM to sign OAuth 2.0 client-based tokens if token encryption is disabled.

To Enable Client-Based OAuth 2.0 Token Encryption

1. Navigate to Realms > *Realm Name* > Services > OAuth2 Provider.
2. On the Core tab, enable Use Client-Based Access & Refresh Tokens.
3. On the Advanced tab, enable Client-Based Token Encryption.

   Note that the alias mapped to the algorithm is defined in the secret stores, as shown in the table below:

   + *Secret ID Mappings for Encrypting Client-Based OAuth 2.0 Tokens*

     The following table shows the secret ID mapping used to encrypt client-based access tokens:

Secret ID	Default Alias	Algorithms
am.services.oauth2.stateless.token.encryption	directentest	A128CBC-HS256

   By default, secret IDs are mapped to demo keys contained in the default keystore provided with AM and mapped to the `default-keystore` keystore secret store. Use these keys for demo and test purposes only. For production environments, replace the secrets as required and create mappings for them in a secret store configured in AM.

   For more information about managing secret stores and mapping secret IDs to aliases, see "*Configuring Secrets, Certificates, and Keys*" in the *Security Guide*.

4. Save your changes.

   Client-based OAuth 2.0 access and refresh tokens will now be encrypted.

Configuring Client-Based OAuth 2.0 Token Digital Signatures

AM supports digital signature algorithms that secure the integrity of client-based tokens.

Important

Client-based tokens must be signed and/or encrypted for security reasons. If your environment does not support encrypting OAuth 2.0 tokens, you must configure signing to protect them against tampering.

AM exposes the public key to validate client-based token signatures in its JWK URI. See "/oauth2/connect/jwk_uri" in the *OpenID Connect 1.0 Guide*.

To Configure the OAuth 2.0 Provider to Sign Client-Based Tokens

Perform the steps in this procedure to configure the OAuth 2.0 provider to sign client-based tokens:

1. Navigate to Realms > *Realm Name* > Services, and then click OAuth2 Provider.
2. On the Advanced tab, in the OAuth2 Token Signing Algorithm drop-down list, select the signing algorithm to use for signing client-based tokens.

   Note that the alias mapped to the algorithm is defined in the secret stores, as shown in the table below:

   + *Secret ID Mappings for Signing Client-Based OAuth 2.0 Tokens*

   The following table shows the secret ID mappings used to sign client-based access tokens:

Secret ID	Default Alias	Algorithms
am.services.oauth2.stateless.signing.ES256	es256test	ES256
am.services.oauth2.stateless.signing.ES384	es384test	ES384
am.services.oauth2.stateless.signing.ES512	es512test	ES512
am.services.oauth2.stateless.signing.HMAC	hmacsigningtest	HS256 HS384 HS512
am.services.oauth2.stateless.signing.RSA	rsajwtsigningkey	PS256 PS384 PS512 RS256 RS384 RS512

   By default, secret IDs are mapped to demo keys contained in the default keystore provided with AM and mapped to the `default-keystore` keystore secret store. Use these keys for demo and

test purposes only. For production environments, replace the secrets as required and create mappings for them in a secret store configured in AM.

For more information about managing secret stores and mapping secret IDs to aliases, see "*Configuring Secrets, Certificates, and Keys*" in the *Security Guide*.

3. Save your changes.

   Client-based OAuth 2.0 access and refresh tokens will now be signed.

Chapter 4

About Scopes

OAuth 2.0 flows require scopes to limit the client's access to the resource owner's resources.

+ *What Are Scopes?*

  Scopes are a way to restrict client access to the resource owner's resources, as defined in the *OAuth 2.0 Authorization Framework*.

  Scopes are not associated with data and, in practice, they are just concepts specified as strings that the resource server must interpret in order to provide the required access or resources to the client. The OAuth 2.0 framework does not define any particular value for scopes since they are dependent on the architecture of your environment.

  For example, a client may request the `write` scope, which the resource server may interpret as that the client wants to save some new information in the user's account, such as images or documents.

  A client can request one or more scopes, which AM may display in the consent screen. If the resource owner agrees to share access to their resources, scopes are included in the access token.

For security reasons, AM only accepts scopes preconfigured in the Scope(s) or in the Default Scope(s) fields in the client profile (Realms > *Realm Name* > Applications > OAuth 2.0 > Clients > *Client Name* > Core).

AM checks the requested scopes against the Scope(s) field of the client's profile. If the client requests a scope that is not preconfigured, AM returns an error, such as `Unknown/invalid scope(s)`.

If a client does not request any scopes, AM uses the scopes configured in the Default Scope(s) field of the client's profile. If none are configured, AM uses those configured in the Default Scope(s) field of the OAuth 2.0 provider.

If no scopes are configured by default, AM returns the `No scope requested` error. AM does not use the default scopes in any other circumstance.

Tip

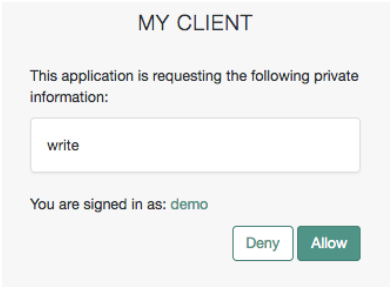
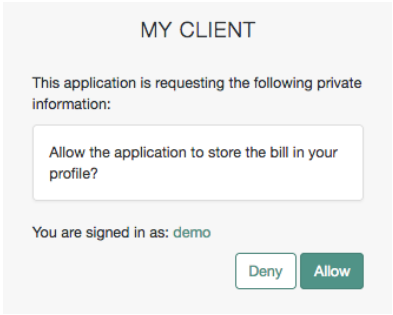
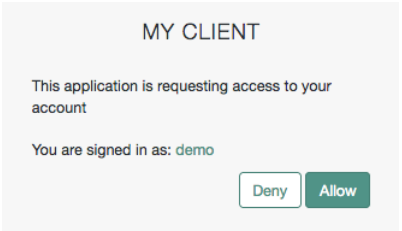
The Client Registration Scope Whitelist field of the OAuth 2.0 provider restricts the scopes a client can register with. In that sense, it is used for OpenID Connect discovery and dynamic client registration *only*.

You can use this field, however, to configure how AM presents the scopes in the AM consent screen. By default, scopes are not configured to display in the consent screen. You can either disable the consent pages, or configure the scopes for display as described below.

Since scope names are arbitrary, in some cases they would not be descriptive enough for the resource owner to understand their purpose. In other cases, you may not want the resource owner to see a particular scope because it is for internal uses only.

Configuring Scopes in the Consent Screen

You can configure the AM consent screen to show, for each scope, one of the following options:

The Scope Itself	A Localized Description	Neither the Scope nor a Description
		

Configure how scopes appear in the consent screen by client or by realm (in the OAuth 2.0 provider service). For examples, see the Client Registration Scope Whitelist field in the provider's "Advanced" in the *Reference* reference section or the Scope(s) field in Core Properties.

Client level configuration overrides that at provider level.

Special Scopes

AM reserves the following special scopes that *cannot be added during dynamic client registration*:

am-introspect-all-tokens

Add this scope to the Scopes(s) field in a client profile to let the client introspect tokens issued to other clients, as long as all clients are registered in the same realm.

For example:

1. Client A is registered in the `/customers/NA` realm, and it is issued a token there.
2. Client B is registered in the `/customers` realm. It cannot introspect Client A's token because they are not in the same realm. Client B can only introspect tokens from other clients registered in the `/customers` realm.

am-introspect-all-tokens-any-realm

Add this scope to the Scopes(s) field in a client profile to let the client introspect tokens issued to other clients, as long as they are registered in the realm of the introspecting client, or in a subrealm of it.

For example:

1. Client A is registered in the `/customers/NA` realm, and it is issued a token there.
2. Client B is registered in the `/customers` realm. It can introspect Client A's token because the `/customers/NA` realm is a subrealm of the `/customers` realm.

   Client B can introspect tokens for any client registered in the `/customer` realm, or any subrealm of it.

For security reasons, give these scopes only to the clients that need them.

Related information:

- For examples of requesting scopes from the authorization server, see any of the grant flows in "*OAuth 2.0 Grant Flows*".
- To create your own implementation of the scope handler, see "Customizing OAuth 2.0 Scope Handling".
- To configure AM to grant scopes dynamically by evaluating authorization policies at runtime, see "*About Authorization and Policy Decisions*" and "*Dynamic OAuth 2.0 Authorization*" in the *Authorization Guide*.

Customizing OAuth 2.0 Scope Handling

RFC 6749, *The OAuth 2.0 Authorization Framework*, describes access token scopes as a set of case-sensitive strings defined by the authorization server. Clients can request scopes, and resource owners can authorize them.

The default scopes implementation in AM treats scopes as per RFC 7662, while the legacy `/oauth2/tokeninfo` endpoint populates the scopes with profile attribute values. For example, if one of the scopes is `mail`, AM sets `mail` to the resource owner's email address in the token information returned.

You can change the scope implementation behavior by writing your own scope validator plugin. This section shows how to write a custom OAuth 2.0 scope validator plugin for use in an OAuth 2.0 provider (authorization server) configuration.

Tip

The default scope validator calls the script that lets AM modify the key pairs contained inside an access token before issuing it. If you intend to use this functionality, you must incorporate this call into your custom scope validator implementation.

About the Scope Validator Plugin Sample

A scope validator plugin implements the `org.forgerock.oauth2.core.ScopeValidator` interface. As described in the API specification, the `ScopeValidator` interface has several methods that your plugin overrides.

This plugin, taken from the *openam-scope-sample* example, sets whether `read` and `write` permissions were granted:

+ *CustomScopeValidator.java*

```
/*
 * Copyright 2014-2022 ForgeRock AS. All Rights Reserved
 *
 * Use of this code requires a commercial software license with ForgeRock AS.
 * or with one of its affiliates. All use shall be exclusively subject
 * to such license between the licensee and ForgeRock AS.
 */

package org.forgerock.openam.examples;

import org.forgerock.oauth2.core.AccessToken;
import org.forgerock.oauth2.core.ClientRegistration;
import org.forgerock.oauth2.core.OAuth2Request;
import org.forgerock.oauth2.core.ScopeValidator;
import org.forgerock.oauth2.core.Token;
import org.forgerock.oauth2.core.UserInfoClaims;
import org.forgerock.oauth2.core.exceptions.InvalidClientException;
import org.forgerock.oauth2.core.exceptions.ServerException;
import org.forgerock.oauth2.core.exceptions.UnauthorizedClientException;

import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

/**
 * Custom scope validators implement the
 * {@link org.forgerock.oauth2.core.ScopeValidator} interface.
 *
 * <p>
 * This example sets read and write permissions according to the scopes set.
 * </p>
 *
 * <ul>
 *
 * <li>
 * The {@code validateAuthorizationScope} method
```

```
 * adds default scopes, or any allowed scopes provided.
 * </li>
 *
 * <li>
 * The {@code validateAccessTokenScope} method
 * adds default scopes, or any allowed scopes provided.
 * </li>
 *
 * <li>
 * The {@code validateRefreshTokenScope} method
 * adds the scopes from the access token,
 * or any requested scopes provided that are also in the access token scopes.
 * </li>
 *
 * <li>
 * The {@code getUserInfo} method
 * populates scope values and sets the resource owner ID to return.
 * </li>
 *
 * <li>
 * The {@code evaluateScope} method
 * populates scope values to return.
 * </li>
 *
 * <li>
 * The {@code additionalDataToReturnFromAuthorizeEndpoint} method
 * returns no additional data (an empty Map).
 * </li>
 *
 * <li>
 * The {@code additionalDataToReturnFromTokenEndpoint} method
 * adds no additional data.
 * </li>
 *
 * </ul>
 */
public class CustomScopeValidator implements ScopeValidator {
    @Override
    public Set<String> validateAuthorizationScope(
            ClientRegistration clientRegistration,
            Set<String> scope,
            OAuth2Request oAuth2Request) {
        if (scope == null || scope.isEmpty()) {
            return clientRegistration.getDefaultScopes();
        }

        Set<String> scopes = new HashSet<String>(
                clientRegistration.getAllowedScopes());
        scopes.retainAll(scope);
        return scopes;
    }

    @Override
    public Set<String> validateAccessTokenScope(
            ClientRegistration clientRegistration,
            Set<String> scope,
            OAuth2Request request) {
        if (scope == null || scope.isEmpty()) {
            return clientRegistration.getDefaultScopes();
```

```
        }

        Set<String> scopes = new HashSet<String>(
                clientRegistration.getAllowedScopes());
        scopes.retainAll(scope);
        return scopes;
    }

    @Override
    public Set<String> validateRefreshTokenScope(
            ClientRegistration clientRegistration,
            Set<String> requestedScope,
            Set<String> tokenScope,
            OAuth2Request request) {
        if (requestedScope == null || requestedScope.isEmpty()) {
            return tokenScope;
        }

        Set<String> scopes = new HashSet<String>(tokenScope);
        scopes.retainAll(requestedScope);
        return scopes;
    }

    /**
     * Set read and write permissions according to scope.
     *
     * @param token The access token presented for validation.
     * @return The map of read and write permissions,
     *         with permissions set to {@code true} or {@code false},
     *         as appropriate.
     */
    private Map<String,Object> mapScopes(AccessToken token) {
        Set<String> scopes = token.getScope();
        Map<String, Object> map = new HashMap<String, Object>();
        final String[] permissions = {"read", "write"};

        for (String scope : permissions) {
            if (scopes.contains(scope)) {
                map.put(scope, true);
            } else {
                map.put(scope, false);
            }
        }
        return map;
    }

    @Override
    public UserInfoClaims getUserInfo(
            AccessToken token,
            OAuth2Request request)
            throws UnauthorizedClientException {
        Map<String, Object> response = mapScopes(token);
        response.put("sub", token.getResourceOwnerId());
        UserInfoClaims userInfoClaims = new UserInfoClaims(response, null);
        return userInfoClaims;
    }

    @Override
    public Map<String, Object> evaluateScope(AccessToken token) {
```

```
        return mapScopes(token);
    }

    @Override
    public Map<String, String> additionalDataToReturnFromAuthorizeEndpoint(
            Map<String, Token> tokens,
            OAuth2Request request) {
        return new HashMap<String, String>(); // No special handling
    }

    @Override
    public void additionalDataToReturnFromTokenEndpoint(
            AccessToken token,
            OAuth2Request request)
            throws ServerException, InvalidClientException {
        // No special handling
    }
}
```

For information on downloading and building AM sample source code, see How do I access and build the sample code provided for AM (All versions)? in the *Knowledge Base*.

Get a local clone so that you can try the sample on your system.

+ *Files Included in the Sample*

`pom.xml`

Apache Maven project file for the module

This file specifies how to build the sample scope validator plugin, and also specifies its dependencies on AM components.

`src/main/java/org/forgerock/openam/examples/CustomScopeValidator.java`

Core class for the sample OAuth 2.0 scope validator plugin

See "About the Scope Validator Plugin Sample" for a listing.

After you successfully build the project, you find the `openam-scope-sample-7.1.jar` in the `/path/to/openam-samples-external/openam-scope-sample/target` directory of the project.

Configuring an Instance to Use the Plugin

After building your plugin .jar file, copy the .jar file under `WEB-INF/lib/` where you deployed AM.

Restart AM or the container in which it runs.

In the AM console, you can either configure a specific OAuth 2.0 provider to use your plugin, or configure your plugin as the default for new OAuth 2.0 providers. In either case, you need the

class name of your plugin. The class name for the sample plugin is `org.forgerock.openam.examples.CustomScopeValidator`.

- To configure a specific OAuth 2.0 provider to use your plugin, navigate to Realms > *Realm Name* > Services, click OAuth2 Provider, and enter the class name of your scopes plugin to the Scope Implementation Class field.
- To configure your plugin as the default for new OAuth 2.0 providers, add the class name of your scopes plugin. Navigate to Configure > Global Services, click OAuth2 Provider, and set Scope Implementation Class.

Trying the Sample Plugin

In order to try the sample plugin, make sure you have configured an OAuth 2.0 provider to use the sample plugin. Also, set up an OAuth 2.0 client of the provider that takes scopes `read` and `write`.

Next try the provider as shown in the following example:

```
$ curl \
--request POST \
--data "grant_type=client_credentials \
&client_id=myClientID&client_secret=password&scope=read" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token"
{
    "scope": "read",
    "expires_in": 59,
    "token_type": "Bearer",
    "access_token": "c8860442-daba-4af0-a1d9-b607c03e5a0b"
}

$ curl https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/tokeninfo\
?access_token=0d492486-11a7-4175-b116-2fc1cbff6d78
{
    "scope": [
        "read"
    ],
    "grant_type": "client_credentials",
    "realm": "/alpha",
    "write": false,
    "read": true,
    "token_type": "Bearer",
    "expires_in": 24,
    "access_token": "c8860442-daba-4af0-a1d9-b607c03e5a0b"
}
```

As seen in this example, the requested scope `read` is authorized, but the `write` scope has not been authorized.

Chapter 5

About Consent

Many of the OAuth 2.0/OpenID Connect flows require the user to explicitly agree to provide the client with access to their resources. This act of trust is one of the pillars of OAuth 2.0 and OpenID Connect.

Users grant consent based on scopes. In OAuth 2.0, scopes are a concept that limits the information to share with the client or the actions the client can do with the user's data. In OpenID Connect, scopes can be mapped to specific user data, too. For example, AM maps the `profile` scope to a number of user profile attributes.

Note

AM has built-in consent pages in its UI, but you can hand off the consent-gathering part of the flow to a separate service by configuring the "*Remote Consent*".

By default, scopes are not configured to display in the consent pages. You can either disable the consent pages, or manually add scopes for display in the OAuth 2.0 provider configuration in the *Reference*.

For OpenID Connect, customize claims for display in the provider configuration in the *Reference* or at the client level.

AM let clients store the scopes to which the user has given consent to improve user experience. This is useful, for example, to minimize customer interaction. In the same way, AM let users revoke consent at any point in time.

In some circumstances, however, clients may need a mechanism to skip consent altogether; for example, for trusted application-to-application or service-to-service interaction.

Tasks:

- "Allowing Clients To Skip Consent"
- "Allowing the OAuth 2.0 Provider to Save Consent"
- "Allowing Users to Revoke Consent"

Allowing Clients To Skip Consent

Companies that have internal applications that use OAuth 2.0 or OpenID Connect can allow clients to skip consent and make consent confirmation optional so as not to disrupt their online experience.

To Allow Clients To Skip Consent

Perform the following steps to configure the OAuth 2.0 service and an OAuth 2.0 client to skip consent:

1. In the AM console, configure the OAuth 2.0 provider to allow clients to skip consent:
   a. Navigate to Realms > *Realm Name* > Services > OAuth 2.0 provider > Consent.
   b. Enable Allow Clients to Skip Consent.
   c. Click Save Changes.
2. Configure the OAuth 2.0 client to skip consent:
   a. Navigate to Realms > *Realm Name* > Applications > OAuth 2.0 > *Client Name* > Advanced.
   b. Enable Implied consent.
   c. Save your changes.

   AM will now treat the requests from this client as if the resource owner/end user has already consented, and will not display consent pages during the flow.

Allowing the OAuth 2.0 Provider to Save Consent

Requesting resource owners/end users consent to sharing their data is extremely important. However, that does not mean that your company needs to be asking for consent every time the user wants to use your services.

To provide a better user experience, AM can store the scopes for which they have given consent in their user profile.

When the client requests a scope combination, AM checks if the user has already consented each scope within the combination. If AM can find the scopes across multiple saved consent entries, AM will not require the user to consent. If part of the requested scope combination is not found in any entry, AM will require the user to consent.

Consider an example where the user grants consent to the `read` scope on a first request and to the `email` and `profile` scopes on a second request. AM will not require consent for a request for the `read` and `profile` scopes.

Tip

To request the user to provide consent even if it is already saved, add the `prompt=consent` parameter to the request.

Resource owners/end users can also revoke consent provided on requests for access tokens at any given time. For more information, see "Allowing Users to Revoke Consent".

To Configure AM to Save Consent

Perform the following steps to configure AM to save consent:

1. Create a multi-valued string syntax attribute in your identity store to save consent entries. For example, `oauth2Consent`.

   To create the attribute and configure it in AM, see "To Update the Identity Repository for the New Attribute" in the *Setup Guide*.

2. In the AM console, go to Realms > *Realm Name* > Services > OAuth 2.0 provider > Consent.

3. In the Saved Consent Attribute field, add the name of the attribute you created in the identity store.

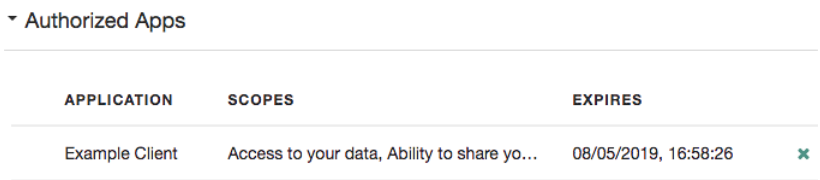
4. Save your changes.

   AM will now save the consented scopes in the identity repository and will only request consent when it cannot find the requested scopes.

Allowing Users to Revoke Consent

Users of OAuth 2.0 clients can manage their authorized applications on their user page in the AM console. For example, the user logs in to the AM console as `demo`, and then clicks the Dashboard link on the Profile page. In the Authorized Apps section, the users can view the client application and the scopes they saved consent during requests for access tokens. Clicking the **x** button will remove consent for those scopes.

OAuth 2.0 Self-Service



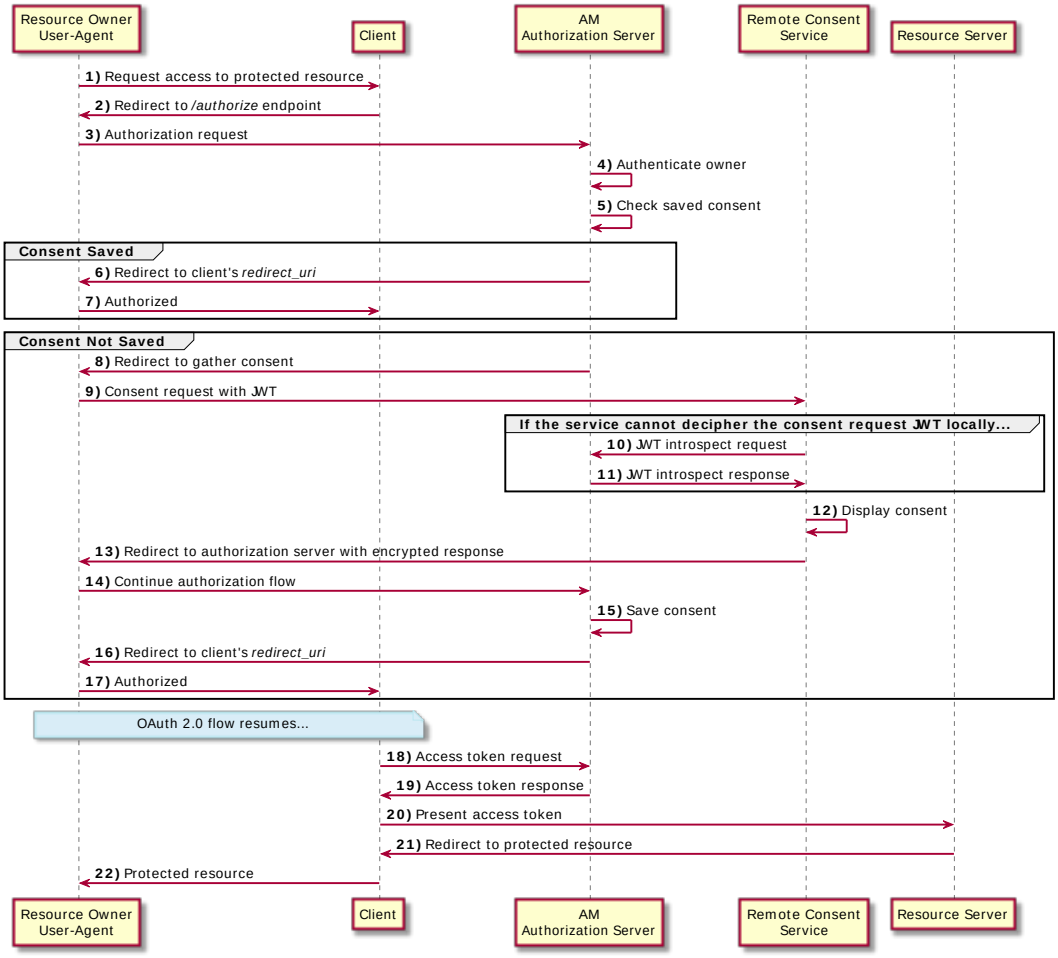
For information about the dashboard service, see "Implementing the Dashboard Service" in the *Setup Guide*.

Chapter 6

Remote Consent

AM supports OAuth 2.0 remote consent services, which allow the consent-gathering part of an OAuth 2.0 flow to be handed off to a separate service.

A remote consent service renders a consent page, gathers the result, signs and encrypts the result, and returns it to the authorization server.



During an OAuth 2.0 flow that requires user consent, AM can create a *consent request* JWT that contains the necessary information to render a consent gathering page. It does not send the actual values of the requested scopes.

+ *Consent Request JWT Example and Properties*

```
{
  "clientId": "myClient",
  "iss": "https://openam.example.com:8443/openam/oauth2",
  "csrf": "gjeH2C43nFJwW+Ir1zL3hl8kux9oatSZRso7aCzI0vk=",
  "client_description": "",
  "aud": "rcs",
  "save_consent_enabled": true,
  "claims": {},
  "scopes": {
      "write": null
  },
  "exp": 1536229486,
  "iat": 1536229306,
  "client_name": "My Client",
  "consentApprovalRedirectUri": "https://openam.example.com:8443/openam/oauth2/authorize?
client_id=MyClient&response_type=code&redirect_uri=https://application.example.com:8443/
callback&scope=write&state=1234zy",
  "username": "demo"
}
```

`iat`

Specifies the creation time of the JWT.

`iss`

Specifies the name of the issuer - configured in the OAuth 2.0 Provider Service in AM.

`aud`

Specifies the name of the expected recipient of the JWT, in this case, the remote consent service.

`exp`

Specifies the expiration time of the JWT.

Use short expiration times, for example, 180 seconds, as the JWT is intended for use in machine-to-machine interactions.

`csrf`

Specifies a unique string that must be returned in the response to help prevent cross-site request forgery (CSRF) attacks.

AM generates this string from a hash of the user's session ID.

`client_id`

Specifies the ID of the OAuth 2.0 client making the request.

`client_name`

Specifies the display name of the OAuth 2.0 client making the request.

`client_description`

Specifies a description of the OAuth 2.0 client making the request.

`username`

Specifies the username of the logged-in user.

Tip

Ensure you encrypt the JWT if the username could be considered personally identifiable information.

`scopes`

Specifies the requested scopes.

`claims`

Specifies the claims the request is making.

Use the claims field for additional information to display on the remote consent page that helps the user to determine if consent should be granted. For example, Open Banking OAuth 2.0 flows may include identifiers for a money transaction.

`save_consent_enabled`

Specifies whether to provide the user the option to save their consent decision.

If set to `false`, the value of the `save_consent` property in the consent response from the RCS must also be `false`.

`consentApprovalRedirectUri`

Specifies the URI to return the resource owner to after they have provided consent. The response JWT must be sent as a `consent_response` form parameter in a POST operation to this URI.

Acting as the authorization server, AM signs and encrypts the JWT.

The remote consent service decrypts the JWT, verifies the signature and other details, such as the validity of the `aud`, `iss` and `exp` properties, and renders the consent page to the resource owner.

After the remote consent service gathers the user's consent, it creates a *consent response* JWT, encrypts and signs the response, and returns it to AM for processing.

+ *Consent Response JWT Example and Properties*

```
{
  "consent_response" : {
    "clientId": "myClient",
    "iss": "rcs",
    "csrf": "gjeH2C43nFJwW+Ir1zL3hl8kux9oatSZRso7aCzI0vk=",
    "client_description": "",
    "aud": "https://openam.example.com:8443/openam/oauth2",
    "save_consent": true,
    "claims": {},
    "scopes": "[write]",
    "exp": 1536229430,
    "iat": 1536229250,
    "client_name": "My Client",
    "consentApprovalRedirectUri": "https://openam.example.com:8443/openam/oauth2/authorize?
client_id=MyClient&response_type=code&redirect_uri=https://application.example.com:8443/
callback&scope=write&state=1234zy",
    "username": "demo",
    "decision": true
  },
}
```

`iat`

Specifies the creation time of the JWT.

`iss`

Specifies the name of the remote consent service.

Must match the value of the `aud` property received from AM.

`aud`

Specifies the name of the expected recipient of the JWT, in this case, AM acting as the AS.

Must match the value of the `iss` property received from AM.

`exp`

Specifies the expiration time of the JWT.

Use short expiration times, for example, 180 seconds, as the JWT is intended for use in machine-to-machine interactions.

`decision`

Specifies `true` if consent was provided, or `false` if consent was withheld.

`client_id`

Specifies the ID of the OAuth 2.0 client making the request, matching the value provided in the request.

`client_name`

Specifies the display name of the OAuth 2.0 client making the request.

`client_description`

Specifies a description of the OAuth 2.0 client making the request.

`scopes`

Specifies an array of allowed scopes.

Must be equal to, or a subset of the array of scopes in the request.

`save_consent`

Specifies `true` if the user chose to save their consent decision, or `false` if they did not.

If `save_consent_enabled` was set to `false` in the request, `save_consent` must also be `false`.

`consentApprovalRedirectUri`

Specifies the URI to return the resource owner to after they have provided consent.

AM decrypts and verifies the signature of the consent response and other details, such as the validity of the `aud`, `iss` and `exp` properties, and processes the response. For example, it may save the consent decision if configured to do so.

Note

If the remote consent service compresses the consent response JWT, note that by default, AM rejects JWTs that expand to a size larger than 32 KiB (32768 bytes).For more information, see "Controlling the Maximum Size of Compressed JWTs" in the *Security Guide*.

AM and the remote consent service make their required public keys available fromm two `jwk_uri`s, to enable signing and encryption between the two servers.

Configuring a remote consent service requires completion of these high-level tasks:

Task	Resources
Add the details of the remote consent service as an agent profile in AM You can configure a single remote consent service in a realm, by adding the details to a Remote Consent Agent profile. The profile defines properties for signing and encrypting the consent request and consent response, redirect URI, and the `jwk_uri` URI details of the remote consent service.	"To Configure AM to use a Remote Consent Service"
Enable remote consent and specify the agent profile in AM's OAuth 2.0 provider service.	"To Configure the OAuth 2.0 Provider to Use a Remote Consent Agent Profile"

Task	Resources
Configure the remote consent service with AM's `jwk_uri` URI details The remote consent service must be able to obtain the required signature and decryption keys from AM.	N/A

Note

AM includes an example remote consent service. Do not use the example in production environments.

See "To Configure the AM Example Remote Consent Service".

To Configure AM to use a Remote Consent Service

To add the details of the remote consent service as an agent profile:

1. In the AM console, go to Realms > *Realm Name* > Applications > OAuth 2.0 > Remote Consent, and select Add Remote Consent Agent.
2. Enter an Agent ID, for example, `myRCSAgent`, and then select Create.
3. (Optional) If you will be using an HMAC algorithm for signing the JWTs, enter the shared symmetric key in the Remote Consent Service secret field.

   This step is not required when using other algorithms.
4. Select the Remote Consent Agent, and then configure the properties as required.

   + *Remote Consent Agent Properties*

   Group

   Configure several remote consent agent profiles by assigning them to a group.

   Default value: `none`

   amster attribute: `agentgroup`

   Remote Consent Service secret

   If the remote consent agent needs to authenticate to AM, enter the password it will use. Reenter the password in the Remote Consent Service secret (confirm) property.

   amster attribute: `userpassword`

   Redirect URL

   Specify the URL to which the user should be redirected during the OAuth 2.0 flow to obtain their consent.

The AM example remote consent service provides an `/oauth2/consent` path to obtain consent from the user.

Example: `https://rcs.example.com:8443/openam/oauth2/consent`

amster attribute: `remoteConsentRedirectUrl`

Consent Request Signing Algorithm

Specify the algorithm used to sign the consent request JWT sent to the Remote Consent Service.

The signing key used depends on the algorithm chosen. The relevant secret IDs and the default signing key aliases are shown in the table below:

+ *Secret ID Mappings for Signing Remote Consent Requests*

The following table shows the secret ID mappings used to sign remote consent requests:

Secret ID	Default Alias	Algorithms [a]
am.applications.agents.remote.consent.request.signing.ES256	es256test	ES256
am.applications.agents.remote.consent.request.signing.ES384	es384test	ES384
am.applications.agents.remote.consent.request.signing.ES512	es512test	ES512
am.applications.agents.remote.consent.request.signing.RSA	rsajwtsigningkey	RS256 RS384 RS512 PS256 PS384 PS512

[a] If you select an HMAC algorithm for signing consent requests (`HS256`, `HS384`, or `HS512`), the value of the Remote Consent Service secret property is used, instead of an entry from the secret stores.

Since the HMAC secret is shared between AM and the remote consent client, a malicious user compromising the client could potentially create tokens that AM would trust. Therefore, to protect against misuse, AM also signs the token using a non-shared signing key configured in the `am.services.oauth2.jwt.authenticity.signing` secret ID.

By default, secret IDs are mapped to demo keys contained in the default keystore provided with AM and mapped to the `default-keystore` keystore secret store. Use these keys for demo and test purposes only. For production environments, replace the secrets as required and create mappings for them in a secret store configured in AM.

For more information about managing secret stores and mapping secret IDs to aliases, see "*Configuring Secrets, Certificates, and Keys*" in the *Security Guide*.

Default value: `RS256`

amster attribute: `remoteConsentRequestSigningAlgorithm`

Enable consent request Encryption

Specify whether to encrypt the consent request JWT sent to the Remote Consent Service.

Default: `true`

amster attribute: `remoteConsentRequestEncryptionEnabled`

Consent request Encryption Algorithm

Specify the encryption algorithm used to encrypt the consent request JWT sent to the Remote Consent Service.

AM supports the following encryption algorithms:

- `A128KW` - AES Key Wrapping with 128-bit key derived from the client secret.
- `A192KW` - AES Key Wrapping with 192-bit key derived from the client secret.
- `A256KW` - AES Key Wrapping with 256-bit key derived from the client secret.
- `RSA-OAEP` - RSA with Optimal Asymmetric Encryption Padding (OAEP) with SHA-1 and MGF-1.
- `RSA-OAEP-256` - RSA with OAEP with SHA-256 and MGF-1.
- `RSA1_5` - RSA with PKCS#1 v1.5 padding.
- `dir` - Direct encryption with AES using the hashed client secret.

Default value: `RSA-OAEP-256`

amster attribute: `remoteConsentRequestEncryptionAlgorithm`

Consent request Encryption Method

Specify the encryption method used to encrypt the consent request JWT sent to the Remote Consent Service.

AM supports the following encryption methods:

- `A128GCM`, `A192GCM`, and `A256GCM` - AES in Galois Counter Mode (GCM) authenticated encryption mode.
- `A128CBC-HS256`, `A192CBC-HS384`, and `A256CBC-HS512` - AES encryption in CBC mode, with HMAC-SHA-2 for integrity.

Default value: `A128GCM`

amster attribute: `remoteConsentRequestEncryptionMethod`

Consent response signing algorithm

Specify the algorithm used to verify a signed consent response JWT received from the Remote Consent Service.

AM supports signing algorithms listed in JSON Web Algorithms (JWA): "alg" (Algorithm) Header Parameter Values for JWS:

- `ES256` - ECDSA with SHA-256 and NIST standard P-256 elliptic curve.
- `ES384` - ECDSA with SHA-384 and NIST standard P-384 elliptic curve.
- `ES512` - ECDSA with SHA-512 and NIST standard P-521 elliptic curve.
- `HS256` - HMAC with SHA-256.
- `HS384` - HMAC with SHA-384.
- `HS512` - HMAC with SHA-512.
- `RS256` - RSASSA-PKCS-v1_5 using SHA-256.

Default value: `RS256`

amster attribute: `remoteConsentResponseSigningAlg`

Consent response encryption algorithm

Specify the encryption algorithm used to decrypt the consent response JWT received from the Remote Consent Service.

AM supports the following encryption algorithms:

- `A128KW` - AES Key Wrapping with 128-bit key derived from the client secret.
- `A192KW` - AES Key Wrapping with 192-bit key derived from the client secret.
- `A256KW` - AES Key Wrapping with 256-bit key derived from the client secret.
- `RSA-OAEP-256` - RSA with OAEP with SHA-256 and MGF-1.
- `dir` - Direct encryption with AES using the hashed client secret.

The decryption key used depends on the algorithm chosen. The relevant secret IDs and the default decryption key aliases are shown in the table below:

+ *Secret ID Mappings for Decrypting Remote Consent Responses*

The following table shows the secret ID mapping used to decrypt remote consent responses:

Secret ID	Default Alias	Algorithms[a]
am.services.oauth2.remote.consent.response.decryption	test	RSA-OAEP-256

[a] If you select an algorithm other than `RSA-OAEP-256` for decrypting consent responses, the value of the Remote Consent Service secret property is used, instead of an entry from the secret stores.

By default, secret IDs are mapped to demo keys contained in the default keystore provided with AM and mapped to the `default-keystore` keystore secret store. Use these keys for demo and test purposes only. For production environments, replace the secrets as required and create mappings for them in a secret store configured in AM.

For more information about managing secret stores and mapping secret IDs to aliases, see "*Configuring Secrets, Certificates, and Keys*" in the *Security Guide*.

Default value: `RSA-OAEP-256`

amster attribute: `remoteConsentResponseEncryptionAlgorithm`

Consent response encryption method

Specify the encryption method used to decrypt the consent response JWT received from the Remote Consent Service.

AM supports the following encryption methods:

- `A128GCM`, `A192GCM`, and `A256GCM` - AES in Galois Counter Mode (GCM) authenticated encryption mode.
- `A128CBC-HS256`, `A192CBC-HS384`, and `A256CBC-HS512` - AES encryption in CBC mode, with HMAC-SHA-2 for integrity.

Default value: `A128GCM`

amster attribute: `remoteConsentResponseEncryptionMethod`

Public key selector

Specify whether the remote consent service provides its public keys using a `JWKs_URI`, or manually in `JWKs` format.

If you select `JWKs`, enter the keys in the Json Web Key property. Otherwise complete the JWKs URI-related properties.

Default: `JWKs_URI`

amster attribute: `remoteConsentRedirectUrl`

Json Web Key URI

Specify the URI from which AM can obtain the Remote Consent Service's public keys.

The example remote consent service uses the URI `/oauth2/consent/jwk_uri` to provide its public keys.

Example: `http://rcs.example.com:8080/openam/oauth2/consent/jwk_uri`

amster attribute: `jwksUri`

JWKs URI content cache timeout in ms

Specify the amount of time, in milliseconds, that the content of the JWKs' URI is cached for before being refreshed. Caching the content avoids fetching it for every token encryption or validation.

Default: `3600000`

amster attribute: `com.forgerock.openam.oauth2provider.jwksCacheTimeout`

JWKs URI content cache miss cache time

Specify the amount of time, in milliseconds, that AM waits before fetching the URI's content again when a key ID (`kid`) is not in the JWKs that are already cached.

For example, if a request comes in with a `kid` that is not in the cached JWKs, AM checks the value of JWKs' URI content cache miss cache time. If the amount of time specified in this property has already passed since the last time AM fetched the JWKs, AM fetches them again. Otherwise, the request is rejected.

Use this property as a rate limit to prevent denial-of-service attacks against the URI.

Default: `60000`

amster attribute: `com.forgerock.openam.oauth2provider.jwkStoreCacheMissCacheTime`

Json Web Key

If the Public key selector: property is set to `JWKs`, specify the Remote Consent Service's public keys, in JSON Web Key format.

Example:

```
{
        "keys": [
        {
        "kty": "RSA",
        "kid": "RemA6Gw0...LzsJ5zG3E=",
        "use": "enc",
        "alg": "RSA-OAEP-256",
        "n": "AL4kjz74rDo3VQ3Wx...nhch4qJRGt2QnCF7M0",
        "e": "AQAB"
        },
        {
        "kty": "RSA",
        "kid": "wUy3ifIIaL...eM1rP1QM=",
        "use": "sig",
        "alg": "RS256",
        "n": "ANdIhkOZeSHagT9Ze...ciOACVuGUoNTzztlCUk",
        "e": "AQAB"
        }
        ]
        }
```

amster attribute: `jwkSet`

Consent Request Time Limit

Specify the amount of time, in seconds, for which the consent request JWT sent to the Remote Consent Service should be considered valid.

Default: `180`

amster attribute: `requestTimeLimit`

5. Save your changes.

   The Remote Consent Agent profile is now available for selection in the OAuth 2.0 provider. See "To Configure the OAuth 2.0 Provider to Use a Remote Consent Agent Profile".

To Configure the OAuth 2.0 Provider to Use a Remote Consent Agent Profile

To add the details of the Remote Consent Agent profile to an OAuth 2.0 provider service:

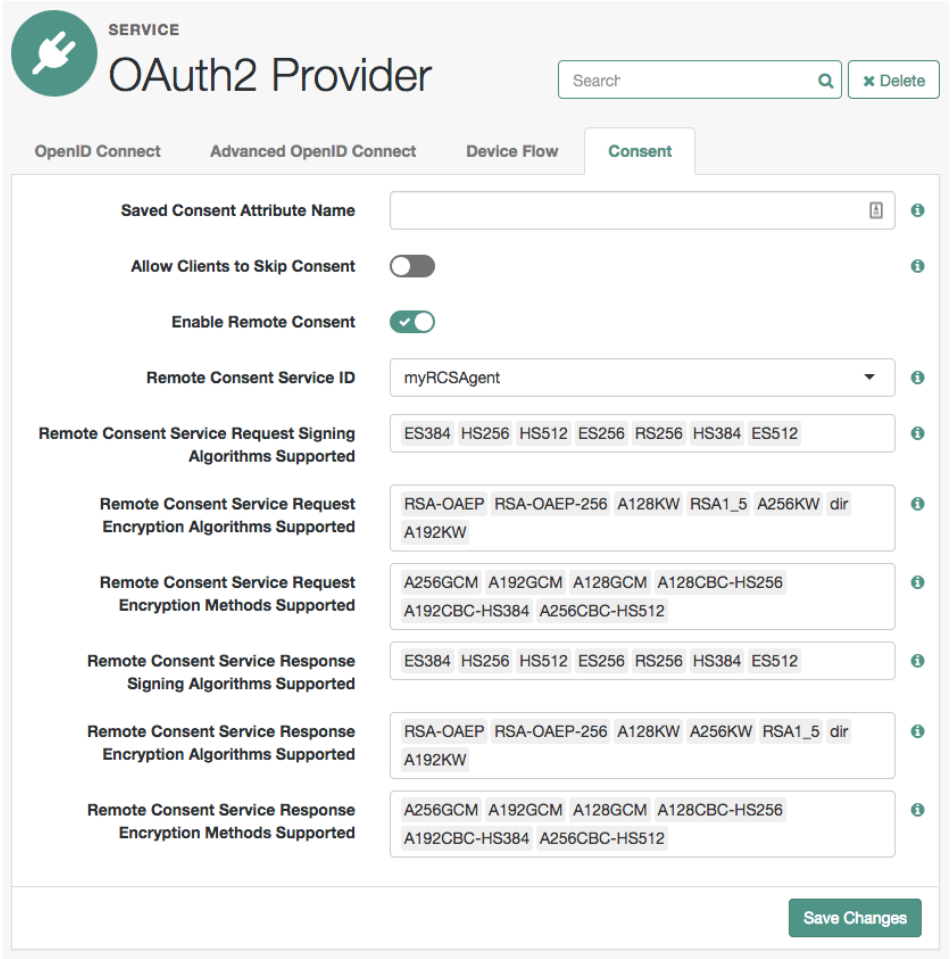
1. In the AM console, go to Realms > *Realm Name* > Services, and then select OAuth2 Provider.
2. On the Consent tab:
   a. Select Enable Remote Consent.
   b. In the Remote Consent Service ID drop-down list, select the Agent ID of the Remote Consent Agent. For example, `myRCSAgent`.

3. (Optional) If required, modify the supported signing and encryption methods and algorithms used for the consent request and consent response JSON web tokens.

   For more information on the available properties, see "Consent" in the *Reference*.

   + *Example*

Configuring RCS in an OAuth 2.0 Provider



4. Save your changes.

OAuth 2.0 flows by any client in the realm will now use the remote consent service. OAuth 2.0 clients in other realms are unaffected.

To Configure the AM Example Remote Consent Service

AM includes an example Remote Consent Service that lets you demonstrate and test remote consent.

Note

The example remote consent service is not intended for use in production environments, because the encryption and signing algorithms are not configurable. It is just an example that shows how you can configure AM to use a custom remote consent service.

The following example uses two instances of AM:

- One instance that acts as the authorization server. For example, `https://openam.example.com:8443/openam`.
- One instance that acts as the example remote consent service. For example, `https://rcs.exampe.com:8443/openam`.

Perform the following steps to configure your environment:

1. As an administrative user, for example, `amAdmin` log in to the instance that acts as the example remote consent service with.
2. Go to Realms > *Realm Name* > Services, and click Add a Service.
3. From the Choose a service type drop-down list, select Remote Consent Service.
4. Perform the following steps to configure the Remote Consent Service:
   a. In Client Name, enter the Agent ID given to the Remote Consent Agent profile in AM.

      In this example, enter `myRCSAgent`.

   b. In the Authorization Server jwk_uri field, enter the URI where the remote consent service will obtain the keys that the authorization service uses to sign and encrypt the consent request. These keys include:

      - The public signing key, used to sign the consent request that is sent to the remote consent server, so that it can be validated on the remote consent server.
      - The public encryption key for the consent response, so that the response can be encrypted (if encryption is enabled).

      The default JWKs URI for remote consent clients is `/oauth2/consent_agents/jwk_uri`.

For example, `https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/consent_agents/jwk_uri`.

c. Select Create.

d. Verify the configuration. For more information about the available properties, see "Remote Consent Service" in the *Reference*.

5. Configure the secret IDs for encrypting the consent request and signing the consent response.

   You can use an existing secret store at the global or realm level, or create a new secret store. This step assumes you have a keystore secret store in the realm where the Remote Consent Server is configured:

   a. Go to Realms > *Realm Name* > Secret Stores, and click on the name of the keystore secret store that contains the secrets you will use to sign and encrypt the consent response.

   b. On the Mappings tab, add the following mappings if not already present:

      - `am.services.oauth2.remote.consent.response.signing.RSA`: `rsajwtsigningkey`
      - `am.services.oauth2.remote.consent.request.encryption`: `test`

      These keys must match the configuration of the remote consent service agent.

      For more information about mapping secrets, see "Mapping and Rotating Secrets" in the *Security Guide*.

6. In the AM console of the instance acting as the OAuth 2.0 provider, configure a remote consent service agent by performing the steps in "To Configure AM to use a Remote Consent Service".

   Note

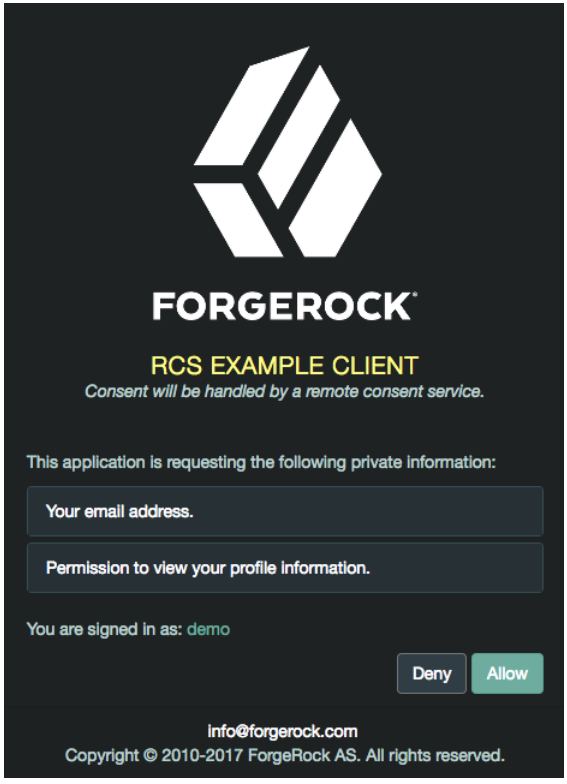
   The example remote consent service provides an `/oauth2/consent/jwk_uri` path to provide its public keys to the authorization server. In this example, configure `https://rcs.example.com:8443/openam/oauth2/consent/jwk_uri` in the Json Web Key URI field.

7. Configure the authorization server to use the remote consent service agent by performing the steps in "To Configure the OAuth 2.0 Provider to Use a Remote Consent Agent Profile".

8. Test your configuration.

   Performing an OAuth 2.0 flow on the AM instance that is acting as the authorization server will redirect the user to the second instance when user consent is required:

Example Remote Consent Service



Note that the *fr-dark-theme* has been applied to the AM instance acting as the remote consent service for the purpose of this demonstration.

For more information on customizing the user interface, see the UI Customization Guide.

Chapter 7
Client Registration

You can register OAuth 2.0/OpenID Connect clients with the AM OAuth 2.0 authorization service by creating and configuring a client profile. When creating a client profile, you must provide at least the client identifier and client secret.

Alternatively, you can register a client dynamically in the *OpenID Connect 1.0 Guide*. AM supports open registration, registration with an access token, and registration including a secure software statement issued by a software publisher.

You can also create an OAuth 2.0 client profile group. OAuth 2.0 clients within a group can specify one or more properties that inherit their values from the group, allowing configuration of multiple OAuth 2.0 clients simultaneously. For more information, see "To Configure a Client Profile Group".

To Create and Configure a Client Profile

1. In the AM console, go to Realms > *Realm Name* > Applications > OAuth 2.0 > Clients.
2. Click Add Client, and then provide the Client ID, Client secret, Redirection URIs, Scope(s), and Default Scope(s).

   Finally, click Create to create the profile.
3. Adjust the configuration as needed using the inline help for hints and the following documentation:

   + *Core Properties*

   Group

   Set this field if you have configured an OAuth 2.0 client group.

   Status

   Specify whether the client profile is active for use or inactive.

   Client secret

   Specify the client secret as described by RFC 6749 in the section, Client Password.

For OAuth 2.0/OpenID Connect 1.0 clients, AM uses the client password as the client shared secret key when signing the contents of the `request` parameter with HMAC-based algorithms, such as HS256.

Client type

Specify the client type.

Confidential clients can maintain the confidentiality of their credentials, such as a web application running on a server where its credentials are protected. *Public* clients run the risk of exposing their passwords to a host or user agent, such as a JavaScript client running in a browser.

Allow wildcard ports in redirection URIs

Specify whether AM allows the use of wildcards (* characters) in the redirection URI port to match one or more ports.

The URL configured in the redirection URI must be either `localhost`, `127.0.01`, or `::1`. For example, `http://localhost:*/`, `https://127.0.0.1:80*/`, or `https://[::1]:*443/`.

Enable this setting, for example, for desktop apps that start a web server on a random free port during the OAuth 2.0 flow.

Redirection URIs

Specify client redirection endpoint URIs as described by RFC 6749 in the section, Redirection Endpoint. AM's OAuth 2.0 authorization service redirects the resource owner's user-agent back to this endpoint during the authorization code grant process. If your client has more than one redirection URI, then it must specify the redirection URI to use in the authorization request.

Redirection URI values must NOT contain a fragment (**#**) and must be an exact match. Wildcards are only considered special characters for ports when the Allow wildcard ports in redirection URIs option is enabled.

OpenID Connect clients require redirection URIs.

Scope(s)

Specify scopes that are to be presented to the resource owner when the resource owner is asked to authorize client access to protected resources.

The `openid` scope is required for OpenID Connect clients. It indicates that the client is making an OpenID Connect request to the authorization server.

Scopes can be entered as simple strings, such as `openid`, `read`, `email`, `profile`, or as a pipe-separated string in the format: *`scope|locale|localized description`*. For example, `read|en|Permission to view email messages`.

Locale strings have the format: *language_country_variant*. For example, `en`, `en_GB`, or `en_US_WIN`. If the `locale` and pipe is omitted, the *localized description* is displayed to all users having undefined locales. If the *localized description* is omitted, nothing is displayed to all users. For example, a scope of `read|` would allow the client to use the `read` scope but would not display it to the user when requested.

AM reserves special scopes to let resource servers introspect tokens issued to other clients. For more information, see "Special Scopes".

For more information about scopes and default scopes, and how AM uses them, see "*About Scopes*".

Default Scope(s)

Scopes that AM uses when the client does not request any during a grant flow.

Specify scopes in *`scope`* or *`scope|locale|localized description`* format.

Scopes defined in this property take the same format as those defined in Scope(s).

For more information about scopes and default scopes, and how AM uses them, see "*About Scopes*".

Client Name

Specify a human-readable name for the client.

Authorization Code Lifetime (seconds)

Specify the time in seconds for an authorization code to be valid. If this field is set to zero, the authorization code lifetime of the OAuth2 provider is used.

Default: `0`

Refresh Token Lifetime (seconds)

Specify the time in seconds for a refresh token to be valid. If this field is set to zero, the refresh token lifetime of the OAuth2 provider is used. If the field is set to `-1`, the token will never expire.

Default: `0`

Access Token Lifetime (seconds)

Specify the time in seconds for an access token to be valid. If this field is set to zero, the access token lifetime of the OAuth2 provider is used.

Default: `0`

+ *Advanced Properties*

Token Exchange Auth Level

Specify the authentication level that will be set for tokens created as a result of token exchange when the subject token does not have an authentication level to begin with. For example, when exchanging an ID token for an access token.

Default: 0

Display name

Specify a client name to display to the resource owner when the resource owner is asked to authorize client access to protected resources. Valid formats include *`name`* or *`locale|localized name`* .

The Display name can be entered as a single string or as a pipe-separated string for locale and localized name, for example, `en|My Example Company`.

Locale strings have the format:*language_ country_ variant*. For example, `en`, `en_GB`, or `en_US_WIN`. If the `locale` is omitted, the name is displayed to all users having undefined locales.

Display description

Specify a client description to display to the resource owner when the resource owner is asked to authorize client access to protected resources. Valid formats include *`description`* or *`locale|localized description`*.

The Display description can be entered as a single string or as a pipe-separated string for locale and localized name, for example, `en|The company intranet is requesting the following access permission`.

Locale strings have the format:*language_ country_ variant*. For example, `en`, `en_GB`, or `en_US_WIN`. If the `locale` is omitted, the name is displayed to all users having undefined locales.

JavaScript Origins

Specify the origin URLs that the client allows to make cross-origin resource sharing (CORS) requests to AM.

Note

This property does *not* support using a non-standard header. To use a custom header, you must create a new CORS configuration in the *Security Guide*.

For example, you might add the URL of a resource server being protected by an app that uses the ForgeRock SDKs, so that it can access the OAuth 2.0 endpoints from a different domain than AM.

Wildcards are not supported; each value should be an exact match for the origin of the CORS request.

The global CORS service collects the value of this property from each OAuth 2.0 client, and combines it with its own configuration.

Important

Ensure that customers whitelist *all* headers for CORS and OAuth 2.0 client integration with AM.

For more information, refer to "Configuring CORS Support" in the *Security Guide*.

Request uris

Specify `request_uri` values that a dynamic client would pre-register.

URIs must be pre-registered in this field before the client can request them in the `request_uri` parameter.

Grant Types

Specify the set of OAuth 2.0 grant flows allowed for this client. The following flows are available:

- `Authorization Code`
- `Back Channel Request`
- `Implicit`
- `Resource Owner Password Credentials`
- `Client Credentials`
- `Refresh Token`
- `UMA`
- `Device Code`

- `SAML2`
- `Token Exchange`

When registering clients dynamically, if no grant types are specified in the registration request, then the default `Authorization Code` grant type is assumed, and configured in the client.

Any grant types selected in a client must also be enabled in the OAuth 2.0 provider service. See "OAuth2 Provider" in the *Reference* .

Default: `Authorization Code`

Response Types

Specify the response types that the client uses. The response type value specifies the flow that determine how the ID token and access token are returned to the client. For more information, see OAuth 2.0 Multiple Response Type Encoding Practices.

Configure this field only if the client uses OAuth 2.0 / OpenID Connect grant flows that interact with the `/oauth2/authorize` endpoint.

The following response types are available:

- `code`. Specifies that the client requests an authorization code during the OAuth 2.0 "Authorization Code Grant" or the OpenID Connect "Authorization Code Grant" in the *OpenID Connect 1.0 Guide* flows.

  This response type also applies to the Authorization Code grant with PKCE flows.

- `token`. Specifies that the client requests an access token during the "Implicit Grant" flow.

- `id_token`. Specifies that the client requests an ID token during the OpenID Connect "Implicit Grant" in the *OpenID Connect 1.0 Guide* flow.

- `code token`. Specifies that the client requests an access token and an authorization code during the OpenID Connect "Hybrid Grant" in the *OpenID Connect 1.0 Guide* flow.

- `code id_token`. Specifies that the client requests an authorization code and an ID token during the OpenID Connect "Hybrid Grant" in the *OpenID Connect 1.0 Guide* flow.

- `code token id_token`. Specifies that the client application requests an authorization code, an access token, and an ID token, during the OpenID Connect "Hybrid Grant" in the *OpenID Connect 1.0 Guide* flow.

- `token id_token`. Specifies that the client requests an access token and an ID token during the OpenID Connect "Implicit Grant" in the *OpenID Connect 1.0 Guide* flow.

Contacts

Specify the email addresses of users who administer the client.

Token Endpoint Authentication Method

Specify the authentication method with which a client authenticates to AM (as an authorization server) at the token endpoint. The authentication method applies to OIDC requests with scope `openid`.

- `client_secret_basic`. Clients authenticate with AM (as an authorization server) using the HTTP Basic authentication scheme after receiving a `client_secret` value.
- `client_secret_post`. Clients authenticate with AM (as an authorization server) by including the client credentials in the request body after receiving a `client_secret` value.
- `private_key_jwt`. Clients sign a JSON web token (JWT) with a registered public key.
- `tls_client_auth`. Clients use a CA-signed certificate for mutual TLS authentication.
- `self_signed_tls_client_auth`. Clients use a self-signed certificate for mutual TLS authentication.

For more information, see "*OAuth 2.0 Client Authentication*", and Client Authentication in the *OpenID Connect Core 1.0 incorporating errata set 1* specification.

Sector Identifier URI

Specify the host component of this URI, which is used in the computation of pairwise subject identifiers.

Subject Type

Specify the subject identifier type, which is a locally unique identifier that will be consumed by the client. Select one of two options:

- *public*. Provides the same `sub` (subject) value to all clients.
- *pairwise*. Provides a different `sub` (subject) value to each client.

Access Token

Specify the `registration_access_token` value that you provide when registering the client, and then subsequently when reading or updating the client profile.

Client URI

Specify the URI containing further information about this client. The URI is displayed as a link in user-facing pages, such as consent pages.

The URI can be made locale-specific by specifying a pipe-separated string in the format: *URI*|*locale*. For example, `https://www.example.es:8443/aplicacion/informacion.html|es`

Logo URI

Specify the URI of a logo for the client. The logo is displayed in user-facing pages, such as consent pages.

The logo can be made locale-specific by specifying a pipe-separated string in the format: *URI*|*locale*. For example, `https://www.example.es:8443/aplicacion/imagen.png|es`

Privacy Policy URI

Specify the URI containing the client's privacy policy documentation. The URI is displayed as a link in user-facing pages, such as consent pages.

The URI can be made locale-specific by specifying a pipe-separated string in the format: *URI*|*locale*. For example, `https://www.example.es:8443/aplicacion/legal.html|es`

Implied Consent

Enable the implied consent feature. When enabled, the resource owner will not be asked for consent during authorization flows. The OAuth2 Provider must also be configured to allow clients to skip consent.

OAuth 2.0 Mix-Up Mitigation enabled

Enable OAuth 2.0 mix-up mitigation on the authorization server side.

Enable this setting only if this OAuth 2.0 client supports the OAuth 2.0 Mix-Up Mitigation draft, otherwise AM will fail to validate access token requests received from this client.

+ *OpenID Connect Properties*

Claim(s)

Specify one or more claim name translations that will override those specified for the authentication session. Claims are values that are presented to the user to inform them what data is being made available to the client.

Claims can be in entered as simple strings, such as `name`, `email`, `profile`, or `sub`, or as a pipe-separated string in the format: *`scope|locale|localized description`*. For example, `name|en|Full name of user`.

Locale strings have the format:*language_ country_ variant*. For example, `en`, `en_GB`, or `en_US_WIN`. If the `locale` and pipe is omitted, the *localized description* is displayed to all users

having undefined locales. If the *localized description* is omitted, nothing is displayed to all users. For example, a claim of `name|` would allow the client to use the `name` claim but would not display it to the user when requested.

If a value is not given, the value is computed from the OAuth2 provider.

Post Logout Redirect URIs

Specify one or more allowable URIs to which the user-agent can be redirected to after the client logout process.

Client Session URI

Specify the relying party (client) URI to which the OpenID Connect Provider sends session changed notification messages using the HTML 5 `postMessage` API.

Default Max Age

Specify the maximum time in seconds that a user can be authenticated. If the user last authenticated earlier than this value, then the user must be authenticated again. If specified, the request parameter `max_age` overrides this setting.

Minimum value: `1`.

Default: `600`

Default Max Age Enabled

Enable the default max age feature.

Default ACR values

Default Authentication Context Class Reference values.

Specify strings that will be requested as Voluntary Claims by default in all incoming requests.

Values specified in the `acr_values` request parameter or an individual `acr` claim request override these default values.

OpenID Connect JWT Token Lifetime (seconds)

The time, in seconds, for a JWT to be valid. If this field is set to zero, the JWT token lifetime of the OAuth2 provider is used.

Default: `0`

Backchannel Logout URL

The URL to which AM sends the logout token during back-channel logout.

This URL can use the http or the https scheme, and may contain a port, a path, or query parameters, depending on the implementation of the relying party. For example, `https://my-rp.example.com:443/logout`.

For more information, see "Informing Relying Parties that a Session has Expired" in the *OpenID Connect 1.0 Guide*.

Backchannel Logout Session Required

Specify whether to add the session ID (`sid`) to the logout token. The session ID identifies the relying party's session with the provider.

For more information, see "Informing Relying Parties that a Session has Expired" in the *OpenID Connect 1.0 Guide*.

+ *Signing and Encryption Properties*

AM returns an error if the administrator tries to save a client profile configuration containing an unsupported signing or encryption algorithm on a client.

For example, upon saving the configuration, AM will return an error if there is a typo on an algorithm, or a symmetric signing or encryption algorithm is configured on a public client: these algorithms are derived from the client's secret, which public clients do not have.

Clients registering dynamically must also send supported algorithms as part of their configuration, or AM will reject the registration request.

Different features support different algorithms. Refer to the documentation or the UI for more information.

Json Web Key URI

Specify the URI that contains the client's public keys in JSON web key format.

When the client authenticates to AM using the `private_key_jwt` method, AM checks this field to find the public key to validate the JWT.

JWKs URI content cache timeout in ms

Specify the amount of time, in milliseconds, that the content of the JWKs' URI is cached for before being refreshed. Caching the content avoids fetching it for every token encryption or validation.

Default: `3600000`

JWKs URI content cache miss cache time

Specify the amount of time, in milliseconds, that AM waits before fetching the URI's content again when a key ID (`kid`) is not in the JWK set already cached.

For example, if a request comes in with a `kid` that is not in the cached JWKs, AM checks the value of JWKs' URI content cache miss cache time. If the amount of time specified in this property has already passed since the last time AM fetched the JWKs, AM fetches them again. Otherwise, the request is rejected.

Use this property as a rate limit to prevent denial-of-service attacks against the URI.

Default: `60000`

Token Endpoint Authentication Signing Algorithm

Specify the JWS algorithm that must be used for signing JWTs used to authenticate the client at the Token Endpoint.

JWTs that are *not* signed with the selected algorithm in token requests from the client using the `private_key_jwt` or `client_secret_jwt` authentication methods will be rejected.

Default: `RS256`

Json Web Key

Raw JSON web key value containing the client's public keys.

ID Token Signing Algorithm

Specify the signing algorithm that the ID token must be signed with.

Enable ID Token Encryption

Enable ID token encryption using the specified ID token encryption algorithm.

ID Token Encryption Algorithm

Specify the algorithm that the ID token must be encrypted with.

Default value: `RSA1_5` (RSAES-PKCS1-V1_5).

ID Token Encryption Method

Specify the method that the ID token must be encrypted with.

Default value: `A128CBC-HS256`.

Client ID Token Public Encryption Key

Specify the Base64-encoded public key for encrypting ID tokens.

Client JWT Bearer Public Key Certificate

Specify the base64-encoded X509 certificate in PEM format. The certificate is never used during the signing process, but is used to obtain the client's JWT bearer public key. The client uses the private key to sign client authentication and access token request JWTs, while AM uses the public key for verification.

The following is an example of the certificate:

```
-----BEGIN CERTIFICATE-----
MIIDETCCAfmgAwIBAgIEU8SXLj.....
-----END CERTIFICATE-----
```

You can generate a new key pair alias by using the Java **keytool** command. Follow the steps in "To Create Key Aliases in an Existing Keystore" in the *Security Guide*.

+ *How Do I Export a Certificate in PEM Format?*

```
$ keytool \
-list \
-alias myAlias \
-rfc \
-storetype JCEKS \
-keystore myKeystore.jceks \
-keypass myKeypass \
-storepass myStorepass

Alias name: myAlias
Creation date: Oct 27, 2020
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
-----BEGIN CERTIFICATE-----
MIIDETCCAfmgAwIBAgIEU8SXLj.....
-----END CERTIFICATE-----
```

For more information, see "Authenticating Clients Using JWT Profiles".

mTLS Self-Signed Certificate

Specify the base64-encoded X.509 certificate in PEM format that clients can use to authenticate to the `access_token` endpoint during mutual TLS authentication.

Only applies when clients use self-signed certificates to authenticate.

For more information, see "Mutual TLS Using Self-Signed X.509 Certificates"

mTLS Subject DN

Specify the distinguished name that must exactly match the subject field in the client certificate used for mutual TLS authentication. For example, `CN=myOauth2Client`.

Only applies when clients use CA-signed certificates to authenticate.

For more information, see "Mutual TLS Using Public Key Infrastructure".

Use Certificate-Bound Access Tokens

Specify that access tokens issued to this client should be bound to the X.509 certificate it uses to authenticate to the `access_token` endpoint.

If enabled, AM adds a confirmation key labeled `x5t#S256` to all access tokens. The confirmation key contains the SHA-256 hash of the client's certificate.

For more information, see "Certificate-Bound Proof-of-Possession"

Public key selector

Select the format of the public keys for JWT profile client authentication, ID token encryption, and mTLS self-signed certificate authentication. Valid formats are:

- `JWKs_URI`

  Configure a URI that exposes the client public keys in the Json Web Key URI field, and ensure the following related properties have sensible values for your environment:

  - JWKs URI content cache timeout in ms
  - JWKs URI content cache miss cache time

- `JWKs`

  Enter a JWK Set containing one or more keys in the Json Web Key field. For example:

  ```
  {
      "keys": [
          {
              "kty": "RSA",
              "n": ...
          },
          ...
      ]
  }
  ```

- `X509`

  Enter a key object or a single certificate in one of the following fields, depending on the feature you are configuring:

- (ID token encryption) Client ID Token Public Encryption Key. Requires an RSA public key object in X.509 PEM format. For example:

```
-----BEGIN PUBLIC KEY-----
......
-----END PUBLIC KEY-----
```

- (JWT client authentication) Client JWT Bearer Public Key. Requires a X.509 certificate in PEM format. For example:

```
-----BEGIN CERTIFICATE-----
.....
-----END CERTIFICATE-----
```

- (mTLS client authentication) mTLS Self-Signed Certificate. Requires a X.509 certificate in PEM format. For example:

```
-----BEGIN CERTIFICATE-----
.....
-----END CERTIFICATE-----
```

Default: `JWKs_URI`

User info response format.

Specify the output format from the UserInfo endpoint.

The supported output formats are as follows:

- User info JSON response format.
- User info encrypted JWT response format.
- User info signed JWT response format.
- User info signed then encrypted response format.

For more information on the output format of the UserInfo Response, see Successful UserInfo Response in the *OpenID Connect Core 1.0 incorporating errata set 1* specification.

Default: User info JSON response format.

User info signed response algorithm

Specify the JSON Web Signature (JWS) algorithm for signing UserInfo Responses. If specified, the response will be JSON Web Token (JWT) serialized, and signed using JWS.

The default, if omitted, is for the UserInfo Response to return the claims as a UTF-8-encoded JSON object using the `application/json` content type.

User info encrypted response algorithm

Specify the JSON Web Encryption (JWE) algorithm for encrypting UserInfo Responses.

If both signing and encryption are requested, the response will be signed then encrypted, with the result being a nested JWT.

The default, if omitted, is that no encryption is performed.

User info encrypted response encryption algorithm

Specify the JWE encryption method for encrypting UserInfo Responses. If specified, you must also specify an encryption algorithm in the *User info encrypted response algorithm* property.

AM supports the following encryption methods:

- `A128GCM`, `A192GCM`, and `A256GCM` - AES in Galois Counter Mode (GCM) authenticated encryption mode.
- `A128CBC-HS256`, `A192CBC-HS384`, and `A256CBC-HS512` - AES encryption in CBC mode, with HMAC-SHA-2 for integrity.

Default: `A128CBC-HS256`

Request parameter signing algorithm

Specify the JWS algorithm for signing the request parameter.

Must match one of the values configured in the *Request parameter Signing Algorithms supported* property of the OAuth2 Provider service. See "Advanced OpenID Connect" in the *Reference* .

Request parameter encryption algorithm

Specify the algorithm for encrypting the request parameter.

Must match one of the values configured in the *Request parameter Encryption Algorithms supported* property of the OAuth2 Provider service. See "Advanced OpenID Connect" in the *Reference*

Request parameter encryption method

Specify the method for encrypting the request parameter.

Must match one of the values configured in the *Request parameter Encryption Methods supported* property of the OAuth2 Provider service. See "Advanced OpenID Connect" in the *Reference* .

Default: `A128CBC-HS256`

Token introspection response format

Specifies the format of the token introspection response. The possible values for this property are:

- JSON response format
- Signed JWT response format
- Signed then encrypted JWT response format

Even if the client has configured the response to be JSON-formatted, it can request a signed JWT by adding the `"Accept: application/jwt"` header to the request. However, when a client that is configured to use either of the JWT-formatted responses requests a JSON response, AM returns an error. For an example, see "/oauth2/introspect".

The JWT response format follows the JWT Response for OAuth Token Introspection Internet Draft.

For related signing and encryption algorithms, see the following properties:

- Token introspection response signing algorithm
- Token introspection response encryption algorithm
- Token introspection response encryption method

Default: JSON response format

Token introspection response signing algorithm

Specifies the algorithm used to sign the token introspection response when it is formatted as a signed JWT.

Must match a value configured in the *Token Introspection Response Signing Algorithms Supported* property of the OAuth2 Provider service. See "Advanced OpenID Connect" in the *Reference* .

AM supports the following signing algorithms:

- `HS256` - HMAC with SHA-256.
- `HS384` - HMAC with SHA-384.
- `HS512` - HMAC with SHA-512.
- `ES256` - ECDSA with SHA-256 and NIST standard P-256 elliptic curve.

- `ES384` - ECDSA with SHA-384 and NIST standard P-384 elliptic curve.
- `ES512` - ECDSA with SHA-512 and NIST standard P-521 elliptic curve.
- `RS256` - RSASSA-PKCS-v1_5 using SHA-256.
- `RS384` - RSASSA-PKCS-v1_5 using SHA-384.
- `RS512` - RSASSA-PKCS-v1_5 using SHA-512.
- `EdDSA` - EdDSA with SHA-512.

The signing key used depends on the algorithm chosen. The relevant secret IDs and the default signing key aliases are shown in the table below:

+ *Secret ID Mappings for Signing OpenID Connect Tokens*

The following table shows the secret ID mapping used to sign OpenID Connect ID tokens and backchannel logout tokens:

Secret ID	Default Alias	Algorithms [a]
am.services.oauth2.oidc.signing.ES256	es256test	ES256
am.services.oauth2.oidc.signing.ES384	es384test	ES384
am.services.oauth2.oidc.signing.ES512	es512test	ES512
am.services.oauth2.oidc.signing.RSA	rsajwtsigningkey	PS256 PS384 PS512 RS256 RS384 RS512
am.services.oauth2.oidc.signing.EDDSA	–	EdDSA with SHA-512

[a] The following applies to confidential clients only:

If you select an HMAC algorithm for signing ID tokens (`HS256`, `HS384`, or `HS512`), the Client Secret property value in the OAuth 2.0 Client is used as the HMAC secret instead of an entry from the secret stores.

Since the HMAC secret is shared between AM and the client, a malicious user compromising the client could potentially create tokens that AM would trust. Therefore, to protect against misuse, AM also signs the token using a non-shared signing key configured in the `am.services.oauth2.jwt.authenticity.signing` secret ID.

Default: RS256

Token introspection response encryption algorithm

Specifies the algorithm used to encrypt the token introspection response when it is formatted as a signed then encrypted JWT.

Must match a value configured in the *Token Introspection Response Encryption Algorithms Supported* property of the OAuth2 Provider service. "Advanced OpenID Connect" in the *Reference* .

AM supports the following encryption algorithms:

- `A128KW` - AES Key Wrapping with 128-bit key derived from the client secret.
- `A192KW` - AES Key Wrapping with 192-bit key derived from the client secret.
- `A256KW` - AES Key Wrapping with 256-bit key derived from the client secret.
- `RSA-OAEP` - RSA with Optimal Asymmetric Encryption Padding (OAEP) with SHA-1 and MGF-1.
- `RSA-OAEP-256` - RSA with OAEP with SHA-256 and MGF-1.
- `RSA1_5` - RSA with PKCS#1 v1.5 padding.
- `dir` - Direct encryption with AES using the hashed client secret.
- `ECDH-ES` - Elliptic Curve Diffie-Hellman
- `ECDH-ES+A128KW` - Elliptic Curve Diffie-Hellman + AES Key Wrapping with 128-bit key.
- `ECDH-ES+A192KW` - Elliptic Curve Diffie-Hellman + AES Key Wrapping with 192-bit key.
- `ECDH-ES+A256KW` - Elliptic Curve Diffie-Hellman + AES Key Wrapping with 256-bit key.

The algorithms that are not specified as being derived from the client secret use the client's public keys. For more information, see the Public key selector property.

Default: RSA-OAEP-256

Token introspection response encryption method

Specifies the encryption method used to encrypt the token introspection response when it is formatted as a signed then encrypted JWT.

Must match a value configured in the *Token Introspection Response Encryption Methods Supported* property of the OAuth2 Provider service. "Advanced OpenID Connect" in the *Reference* .

AM supports the following encryption methods:

- `A128GCM, A192GCM, and A256GCM` - AES in Galois Counter Mode (GCM) authenticated encryption mode.
- `A128CBC-HS256, A192CBC-HS384, and A256CBC-HS512` - AES encryption in CBC mode, with HMAC-SHA-2 for integrity.

Default: A128CBC-HS256

+ *UMA Properties*

Client Redirection URIs

Note

This property is for future use, and not currently active.

Specify one or more allowable URIs to which the client can be redirected after the UMA claims collection process. The URIs must not contain a fragment (#).

If multiple URIs are registered, the client MUST specify the redirection URI to be redirected to following approval.

Some of the configuration of the clients will depend on the configuration of the authorization server, and the type of client you are registering.

+ *Configuration Tips*

Some basic points you must decide on are:

- Is the client public or confidential?
- What is its redirection URI?
- Which scopes does it need?
- What's the name this client will show as in the UI pages?
- Which grant types the client can use to request tokens?
- Which tokens can this client request?
- In the case of an OpenID Connect client:
  - If the client is confidential, which authentication method will it use?

- Which claims does the client need?

4. When finished, save your work.

To Configure a Client Profile Group

1. In the AM console, go to Realms > *Realm Name* > Applications > OAuth 2.0.
   - To create a new OAuth 2.0 client profile group:

     On the Groups tab, select Add Group, and then provide the Group ID. Finally, select Create.
   - To configure a OAuth 2.0 client profile group:

     On the Groups tab, select the group to configure.
2. Adjust the configuration as needed. See "To Create and Configure a Client Profile".
3. When finished, save your work.

   If the group is assigned to one or more OAuth 2.0 client profiles, changes to inherited properties in the group are also applied to the client profile.

   To assign a group to an OAuth 2.0 client profile, see "To Assign a Group to a Client Profile and Inherit Properties".

To Assign a Group to a Client Profile and Inherit Properties

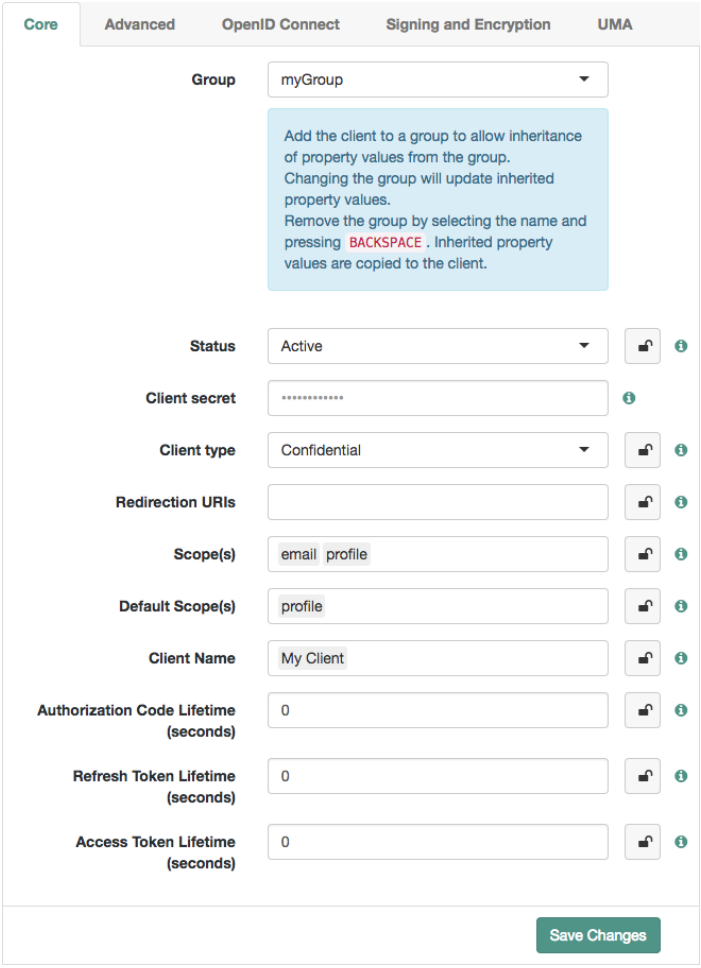
1. In the AM console, go to Realms > *Realm Name* > Applications > OAuth 2.0. On the Clients tab, select the client ID to which a group is to be assigned.
2. On the Core tab, select the group to assign to the client from the Group drop-down.

   Warning

   Adding or changing an assigned group will refresh the settings page. Unsaved property values will be lost.

   The inheritance (padlock) icons appear next to properties that support inheriting their value from the assigned group. Not all properties can inherit their value, such as, the Client secret property.

OAuth 2.0 Client Profile Group Inheritance



3. Inherit a property value from the group by selecting the inheritance button (the open padlock icon) next to the property.

   The value will be inherited from the group and the field will be locked.

> **Note**
>
> If you change the group, properties with inheritance enabled will inherit the value from the new group.

If you remove the group, inherited property values are written to the OAuth 2.0 client profile, and become editable.

4. When finished, save your work.

Chapter 8

OAuth 2.0 Client Authentication

AM can authenticate OAuth 2.0/OpenID Connect clients by using the following methods:

- "Authenticating Clients Using Form Parameters"
- "Authenticating Clients Using Authorization Headers"
- "Authenticating Clients Using JWT Profiles"
- "Authenticating Clients Using Mutual TLS"

Confidential clients holding a secret or a JWT bearer token assertion can authenticate with the authorization server using any of the above methods.

While confidential clients must always authenticate in one of the ways described in this section, public clients are not required to authenticate, because their information is intended to be public or they are used over insecure channels, so their secret could be easily snooped.

Important

OAuth 2.0 and OpenID Connect clients can use the same authentication methods. However, OpenID Connect clients must specify the method they are using in their client profiles.

See "*OpenID Connect Client Authentication*" in the *OpenID Connect 1.0 Guide*.

Authenticating Clients Using Form Parameters

Clients that have a client secret can send the client ID in the `client_id` form parameter and the secret in the `client_secret` form parameter in the body of the request. For example:

```
$ curl \
--request POST \
--data "client_id=myClient" \
--data "client_secret=forgerock" \
...
```

This is the simplest way to authenticate to any of the OAuth 2.0 endpoints, and the most insecure, since the client credentials are exposed. Ensure that communication with the authorization server happens over a secure protocol to protect the secret, and use this method in production only if the other methods are not available for your client.

Tip

OpenID Connect clients must also specify the authentication method they are using in their client profiles. See "*OpenID Connect Client Authentication*" in the *OpenID Connect 1.0 Guide*.

Authenticating Clients Using Authorization Headers

Clients that have a client secret can send the client ID and the secret in a basic authorization header with the base64-encoded value of `client_id:client_secret`. For example:

```
$ curl \
--header "Authorization: Basic bXlDbGllbnQ6Zm9yZ2Vyb2Nr" \
--request POST \
...
```

Note

If the client ID or client secret contains characters that have special meaning in URL-encoded strings, such as percent (**%**) or plus (**+**) characters, you must first URL-encode the string before combining them with the colon character and base64-encoding the result. URL-encoding characters that do not have special meaning in URL-encoded strings will still work, but is unnecessary.

For example, for a client named `example.com` with a client secret of `s=cr%t`:

1. URL-encode the client secret value and combine with the colon character. For example: `example.com:s%3Dcr%25t`.

   Note that you should not URL-encode the separating colon character.

2. Base64-encode the entire string to obtain the basic authorization header. For example, `ZXhhbXBsZS5jb206cyUzRGNyJTI1dA==`

Ensure that communication with the authorization server happens over a secure protocol to help protect the credentials.

Tip

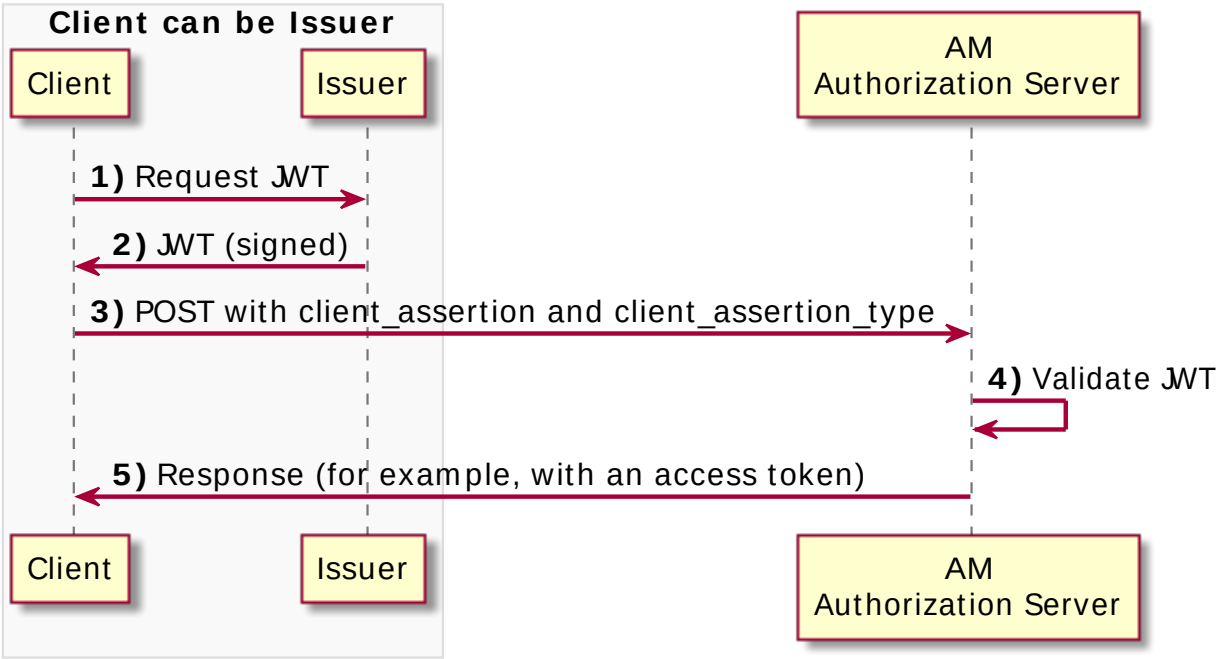
OpenID Connect clients must also specify the authentication method they are using in their client profiles. See "*OpenID Connect Client Authentication*" in the *OpenID Connect 1.0 Guide*.

Authenticating Clients Using JWT Profiles

Clients can send a signed JWT to the authorization server as credentials instead of the client ID and/or secret, as per *(RFC 7523) JWT Profile for OAuth 2.0 Client Authentication and Authorization Grants*. The authorization server must be able to validate the JWT to authenticate the client.

The following diagram demonstrates the JWT Bearer client authentication flow:

JWT Bearer Client Authentication



The steps in the diagram are described below:

1. The client requests a JWT from the issuer.

   Tip

   Clients usually generate their own JWTs before starting the OAuth 2.0/OpenID Connect flow, but they can delegate the task to a specific service in your environment if required. AM cannot generate JWTs for this purpose.

2. The issuer returns a signed JWT to the client. The JWT must contain, at least, the following claims in the payload:

   - **iss**. Specifies the unique identifier of the JWT *issuer*. This could also be the client, or a third party.
   - **sub**. Specifies the principal who is the *subject* of the JWT. Must be set to the client ID.

- **aud**. Specifies the authorization server that is the intended *audience* of the JWT. Must be set to the authorization server's token endpoint. For example, `https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token`.
- **exp**. Specifies the *expiration time*.

  Providing a JWT with an expiry time greater than 30 minutes causes AM to return a `JWT expiration time is unreasonable` error message.
- **jti**. Specifies a random, *unique identifier for the JWT*.

  This claim is *required* if the client requests the `openid` scope, and optional otherwise.

For more information about the JWT, read the RFC 7523 standard.

The JWT issuer must digitally sign the JWT or have a Message Authentication Code (MAC) applied by the issuer. When the issuer is also the client, the client can sign the JWT by using a private key.

AM ignores keys specified in JWT headers, such as `jku` and `jwe`. Regardless of who issues the JWT, you must configure the public key or HMAC secret in the client profile so AM can validate it:

+ *Configuring Certificates Represented as PEM Files*

  a. In the AM console, go to Realms > *Realm Name* > Applications > OAuth 2.0 > *Client Name* > Signing and Encryption.

  b. In the Client JWT Bearer Public Key field, enter the public certificate. For example:

  ```
  -----BEGIN CERTIFICATE-----
  MIIDETCCAfmgAwIBAgIEU8SXLjAN...
  -----END CERTIFICATE-----
  ```

  You can only enter one certificate.

  c. In the Public key selector drop-down list, select `X509`.

+ *Configuring Public Keys in JWK Format*

  You can either enter the JWK Set in the client profile, or store the JWK Set in a URI that exposes it to AM:

  - To store the JWK Set in the client profile:

    a. In the AM console, go to Realms > *Realm Name* > Applications > OAuth 2.0 > *Client Name* > Signing and Encryption.

    b. In the Json Web Key field, enter the JWK Set. For example:

```
{
  "keys": [
    {
      "alg": "RSA-OAEP-256",
      "kty": "RSA",
      "use": "sig",
      "kid": "RemA6Gw0...LzsJ5zG3E=",
      "n": "AL4kjz74rDo3VQ3Wx...nhch4qJRGt2QnCF7M0",
      "e": "AQAB"
    }
  ]
}
```

Enter a JWK Set with multiple JWKs if you plan to rotate certificates.

c. In the Public key selector drop-down list, select `JWKs`.

- To store the JWK Set in a URI:

a. In the AM console, go to Realms > *Realm Name* > Applications > OAuth 2.0 > *Client Name* > Signing and Encryption.

b. In the Json Web Key URI field, configure the URI that exposes the JWK Set. Ensure that the following related properties have sensible values for your environment:

- JWKs URI content cache timeout in ms
- JWKs URI content cache miss cache time

Store a JWK Set with multiple JWKs if you plan to rotate certificates.

c. In the Public key selector drop-down list, select `JWKs_URI`.

+ *Configuring HMAC Secrets*

a. In the AM console, go to Realms > *Realm Name* > Applications > OAuth 2.0 > *Client Name* > Core.

b. In the Client secret field, enter the HMAC secret. For more information about the lenght of the secret, see the Symmetric Key Entropy section of the OpenID Connect specification.

You can only enter one HMAC secret.

Tip

OpenID Connect clients must also specify the authentication method they are using in their client profiles. See "*OpenID Connect Client Authentication*" in the *OpenID Connect 1.0 Guide*.

3. The client includes the JWT and a client assertion type in the call to the OAuth 2.0 endpoint in the following parameters:

   - `client_assertion_type`=`urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer`
   - `client_assertion`=*my_JWT*

   For example:

   ```
   $ curl \
   --request POST \
   --data "client_id=myClient" \
   --data "client_assertion_type=urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer"
   --data "client_assertion=eyAiYWxnIjogIlJTMjU2IiB9.eyAic3ViIjogImp3..."
   ...
   ```

4. The authorization server validates the JWT with the public key stored in the client profile.
5. The authorization server issues a response to the client. This response may include, for example, an access token.

A sample Java-based client to test the JWT token bearer flow is provided.

For information on downloading and building AM sample source code, see How do I access and build the sample code provided for AM (All versions)? in the *Knowledge Base*.

Authenticating Clients Using Mutual TLS

Clients can authenticate to AM by using mutual TLS (or mTLS) and X.509 certificates that are either self-signed, or that use public key infrastructure (PKI), as per version 12 of the draft OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens specification.

Tip

AM also supports the Certificate Bound Access Tokens part of the specification. For more information, see "Certificate-Bound Proof-of-Possession".

Mutual TLS Using Public Key Infrastructure

This method of authenticating OAuth 2.0 clients requires that the certificate presented by the client contains a subject distinguished name that matches exactly a value specified in the client profile in AM.

The Certificate Authority specified in the chain must also be trusted by AM. You can configure secret mappings with secret ID `am.services.oauth2.tls.client.cert.authentication` to specify which certificate authorities AM trusts.

To Configure AM for Mutual TLS Using Public Key Infrastructure

Follow the steps in this procedure to configure AM to support mutual TLS using PKI.

1. If you have not already done so, create an OAuth 2.0 client profile in AM.

   For more information, see "*Client Registration*".

2. Setup a secret store in the same realm as the OAuth 2.0 client. AM maintains the details of trusted certificate authorities in this secret store.

   You can use an existing secret store, or create a new store, as follows:

   a. In the AM console, go to Realms > *Realm Name* > Secret Stores, and then click Add Secret Store.

   b. Enter an ID for the secret store (for example, `TrustStore`), select the store type, complete the required fields, and then click Create.

   Note

   You may need to configure the credentials for accessing the new store in one of the other configured secret stores.

   For more information on configuring secret stores, see "*Configuring Secrets, Certificates, and Keys*" in the *Security Guide*.

3. Import the certificates belonging to the certificate authorities you want the instance of AM to trust.

4. Map the aliases of the imported certificates to the `am.services.oauth2.tls.client.cert.authentication` secret ID:

a. In the AM console, go to Realms > *Realm Name* > Secret Stores > *Store Name* > Mappings, and then click Add Mapping.

b. In the Secret ID field, select `am.services.oauth2.tls.client.cert.authentication`.

c. In the Aliases field, enter the aliases of the imported CA certificate to trust, and then click the Add Alias (+) button.

d. Repeat the previous step to add the aliases of all the CA certificates to trust, and then click Create.

5. Add the subject distinguished name that must appear in the client certificate to be able to authenticate:

a. In the AM console, go to Realms > *Realm Name* > Applications > OAuth 2.0 > *Agent Name* > Signing and Encryption.

b. In the mTLS Subject DN field, enter the distinguished name that must exactly match the subject field in the client certificate. For example, `CN=myOauth2Client`.

> **Note**
>
> If this field is left empty, the default value that must be found in a CA-signed client certificate is `CN=Client ID`. For example, `CN=myMTLSClient`.

c. Save your changes.

6. (Optional) Configure the OAuth 2.0 provider to check whether the certificates presented by the authenticating clients have been revoked:

a. In the AM console, go to Realms > *Realm Name* > Services > OAuth2 Provider > Advanced.

b. Enable Check TLS Certificate Revocation Status.

c. (Optional) In the OCSP Responder URI field, enter the URI of the online certificate status protocol responder service. AM will use this service to check the certificates.

If not specified, AM determines the appropriate URI from the certificate.

d. (Optional) In the OCSP Responder Certificate field, enter the PEM-encoded certificate that AM will use to verify all OCSP responses.

If not specified, AM determines the appropriate certificate from the trusted CA certificates configured in the `am.services.oauth2.tls.client.cert.authentication` secret ID.

AM is now configured to accept CA-signed client certificate for authentication. For information on how to present the certificates when authenticating, see "Providing Client Certificates to AM".

Mutual TLS Using Self-Signed X.509 Certificates

This method of authenticating OAuth 2.0 clients requires that the self-signed X.509 certificate presented by the client matches exactly a certificate specified in the client profile in AM.

You can specify the expected self-signed X.509 certificate in the client profile using one of the following methods:

1. JSON Web Key Set (JWKS)

   Specify the X.509 certificates in the X.509 Certificate Chain (`x5c`) attribute of the one or more JSON Web Keys specified in the set.

2. JSON Web Key Set URI (JWKS_uri)

   AM periodically retrieves the JWKS from the specified URI, and uses the certificates provided in the X.509 Certificate Chain (`x5c`) attribute to verify the client certificate.

3. X.509

   Add content of the X.509 certificate as-is into the client profile.

   Unlike the other methods, only a single certificate can be specified using this method.

To Configure AM for Mutual TLS Using Self-Signed X.509 Certificates

Follow the steps in this procedure to configure AM to support mutual TLS using self-signed certificates.

1. If you have not already done so, create an OAuth 2.0 client profile in AM.

   For more information, see "*Client Registration*".

2. To provide the X.509 certificates the client will use to authenticate, navigate to Applications > OAuth 2.0 > *Agent Name* > Signing and Encryption, and then perform one of the following steps:

   - To use a JSON Web Key Set (JWKS) to specify the certificates:

     a. Set the Public key selector property to JWKs.

     b. Enter the contents of the JWKS in the Json Web Key property.

   - To use a JSON Web Key Set URI (JWKS_uri) to specify the certificates:

     a. Set the Public key selector property to JWKs_uri.

     b. Enter the JWKS URI in the Json Web Key URI property.

   - To use the contents of an X.509 certificate:

a. Set the Public key selector property to X509.

b. In the mTLS Self-Signed Certificate field, enter the content of the X.509 certificate, which must be in PEM format.

Tip

You can choose to include or exclude the `-----BEGIN CERTIFICATE-----` and `-----END CERTIFICATE-----` labels.

Tip

OpenID Connect clients must also specify the authentication method they are using in their client profiles. See "*OpenID Connect Client Authentication*" in the *OpenID Connect 1.0 Guide*.

3. Save your changes.

AM is now configured to accept self-signed client certificate for authentication. For information on how to present the certificates when authenticating, see"Providing Client Certificates to AM".

Providing Client Certificates to AM

The client can provide its certificate to AM by using either of methods below.

Important

You must configure the web container in which AM runs to use TLS connections, and to request and accept client certificates.

Consult the documentation for your web container to determine the appropriate actions to take.

1. Standard TLS Client Certificate Authentication

   The client provides its certificates in the standard servlet client certificate attribute.

   This is the preferred method, as the web container will verify that the client authenticated the TLS session with the private key associated with the certificate.

   After configuring AM to accept client certificates, the client can authenticate to the OAuth 2.0 `access_token` endpoint using one of the X.509 certificates registered in the client.

   Any of the OAuth 2.0 grant flows that makes a call to the `access_token` endpoint can authenticate clients using X.509 certificates. The following example uses `grant_type=client_credentials`, and attaches the client certificates to the request:

```
$ curl --request POST \
--data "client_id=myClient" \
--data "grant_type=client_credentials" \
--data "scope=write" \
--data "response_type=token" \
--cert "myClientCertificate.pem" \
--key "myClientCertificate.key.pem" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token"
{
  "access_token": "sbQZuveFumUDV5R1vVBl6QAGNB8",
  "scope": "write",
  "token_type": "Bearer",
  "expires_in": 3599
}
```

2. Trusted Headers

   AM receives the certificates in a configured, trusted HTTP header.

   This method is intended for cases where TLS is being terminated at a reverse proxy or load balancer, and therefore the container in which AM runs is not directly able to authenticate the client.

   You **must** configure the proxy or load balancer to:

   a. Forward the certificate to AM in the trusted header.

      AM supports receiving certificates in the following formats:

      - Raw PEM-encoded.
      - PEM-encoded first, then URL-encoded, for compatibility with the NGINX `$ssl_client_escaped_cert` variable.
      - PEM-encoded first, URL-encoded next, and then included as a field in a multi-field trusted header, for compatibility with the Envoy `x-forwarded-client-cert` headers.

      To specify the format of the trusted header, go to Realms > *Realm Name* > Services > OAuth2 Provider > Advanced, and choose the appropriate value in the TLS Client Certificate Header Format drop-down list:

      - Use `URLENCODED_PEM` for raw PEM and NGINX-like URL-encoded formats.
      - Use `X_FORWARDED_CLIENT_CERT` for the Envoy-like format.

   b. **Strip the trusted header from any incoming requests**. This is because AM has no way of authenticating the contents of this header, and so would trust whatever is present.

   To specify the name of the trusted header, in the AM console, go to Realms > *Realm Name* > Services > OAuth2 Provider > Advanced, and enter the header name in the Trusted TLS Client Certificate Header property.

Tip

It is recommended to specify a strong, random name for the trusted header. A misconfigured proxy or load balancer could let an attacker send malicious header values. A trusted header name that is difficult to guess makes this type of attack more difficult.

Chapter 9

Proof-of-Possession

Proof-of-possession is a means of ensuring that the client sending a request to the resource server is in possession of a particular cryptographic key. In other words, it is a way of proving the identity of the client.

Configure proof-of-possession to control which clients access your resources, or to mitigate against token theft; a malicious user with an access token must also present the cryptographic key to access the resources.

AM supports the following proof-of-possession methods:

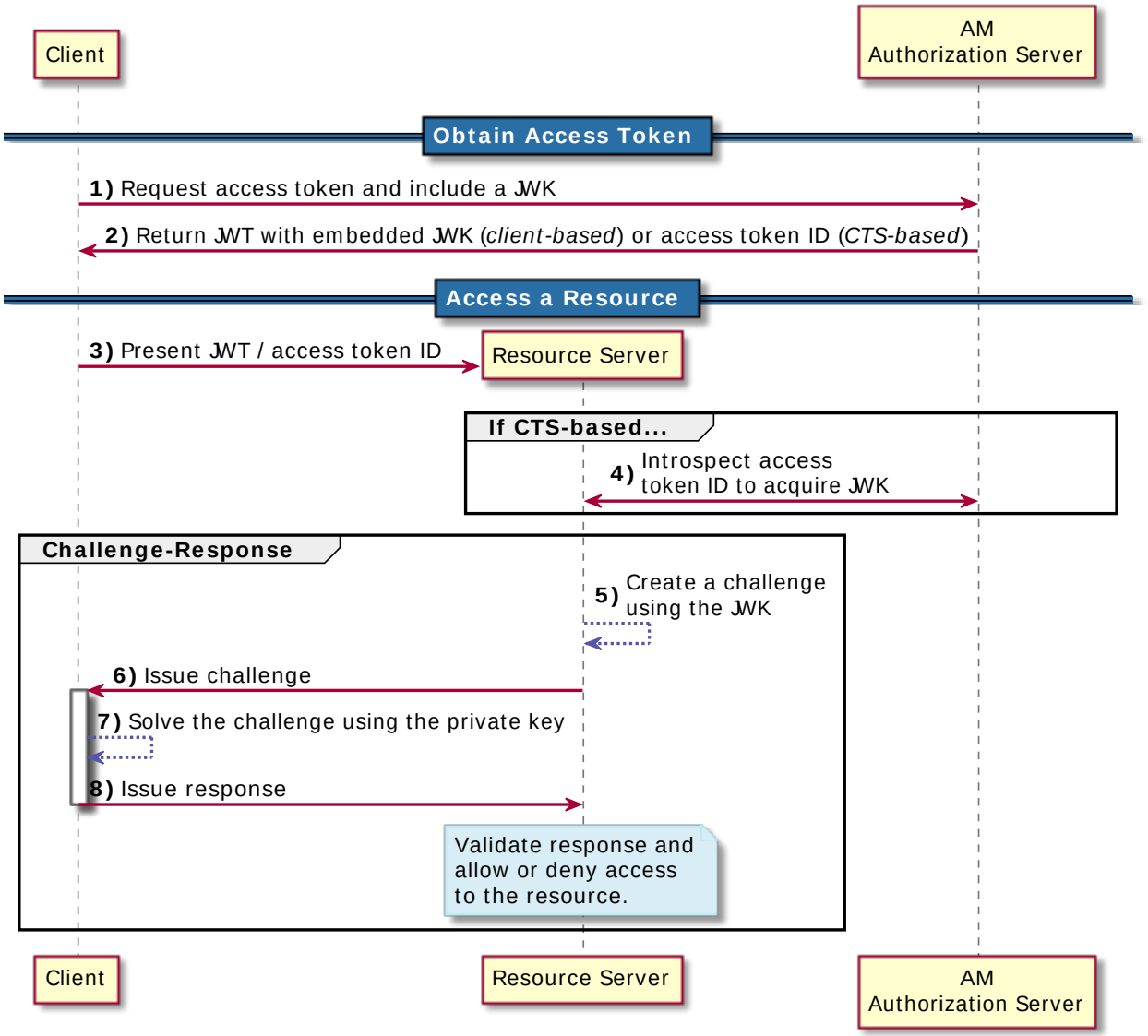
 JWK-Based Proof-of-Possession	 Certificate-Based Proof-of-Possession

JWK-Based Proof-of-Possession

To implement JWK-based proof-of-possession, the client includes a JWK when making a request to the authorization server for an access token as per Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs) spec. The JWK consists of the public key of a key pair generated by the client.

When the client presents the access token to a resource server, the resource server can cryptographically confirm proof-of-possession of the token by using the associated JWK to form a challenge-response interaction with the client.

OAuth 2.0 JWK-Based Proof-of-Possession Flow



+ *JWK-Based Proof-of-Possession Flow Explained*

The steps in the diagram are described below:

1. The client requests an access token using any of the OAuth 2.0 grant flows, and includes a JWK in the request.

   This JWK consists of the public key of a key pair generated by the client.

2. The authorization server returns the access token to the client:

   - If the authorization server is configured for CTS-based OAuth 2.0, it stores the JWK with the access token in the CTS token store and provides the client with the access token ID.
   - If the authorization server is configured for client-based OAuth 2.0, the access token is a JWT that contains the JWK embedded in it.

3. The client requests access to the protected resources from the resource server.
4. The resource server recovers the JWK associated with the access token:

   - If the resource server receives an access token ID (CTS-based OAuth 2.0), it introspects the access token ID to recover the JWK from the authorization's server CTS token store.
   - If the resource server receives an access token JWT (client-based OAuth 2.0), it already has access to the JWK, which is embedded.

5. The resource server creates a challenge using the JWK. Usually, these challenges are messages or nonces that have been encrypted with the JWK.
6. The resource server sends the challenge to the client.
7. The client solves the challenge using the private key of its key pair.
8. The client sends the response to the challenge to the resource server.
9. The resource server validates the response and allows access to the resource.

To use JWK-based proof-of-possession by associating a JWK with an OAuth 2.0 access token, perform the following steps:

To Obtain an Access Token Using JWK-Based Proof-of-Possession

1. Generate a JSON web key pair for the OAuth 2.0 client.

   AM supports both RSA and elliptic curve (EC) key types. For testing purposes, you can use an online JSON web key generator, such as https://mkjwk.org/, to generate a key pair in JWK format. Be sure to store the full key pair, including the private key, in a secure location that is accessible by your OAuth 2.0 client.

   Your OAuth 2.0 client should never reveal the private key.

2. Represent the public key of the key pair in JWK format. For example:

```
{
  "jwk":{
    "alg":"RS256",
    "e":"AQAB",
    "n":"xea7Tb7rbQ4ZrHNKrg...QFXtJ-didSTtXWCWU1Qrcj0hnDjvkuUFWoSQ_7Q",
    "kty":"RSA",
    "use":"enc",
    "kid":"myPublicJSONWebKey"
  }
}
```

Note

The `jwe` and `jku` formats are not supported, the public key must be represented in `jwk` format.

3. Base64-encode the JWK. For example:

```
ew0KICAgICJKV0siOiB7DQogICAgICAgICJhbGciOiAiUlMyNTYiLA0KICAgICAgICAiZSI6IC
JBUUFCIiwNDQogICAgICAgICJraWQiOiAibXlQdWJsaWNKU09OV2ViS2V5Ig0KICAgIH0NCn0=
```

4. The client includes the base64-encoded JWK as the value of the `cnf_key` parameter in the request to the authorization server for an access token.

   For example, in the Client Credentials grant, the client makes a POST call to the authorization server's token endpoint specifying, at least, the following parameters:

   - **grant_type**=client_credentials
   - **cnf_key**=*your_base64-encoded-JWK*

   Confidential clients can authenticate to the OAuth 2.0 endpoints in several ways. This example uses the following form parameters:

   - **client_id**=*your_client_id*
   - **client_secret**=*your_client_secret*

   For more information, see "*OAuth 2.0 Client Authentication*".

   For example:

```
$ curl \
--request POST \
--data "grant_type=client_credentials"\
--data "client_id=myClient" \
--data "client_secret=forgerock" \
--data "cnf_key=ew0KICAgICJKV0siOiB7DQogICAgICAgICJhb
            GciOiAiUlMyNTYiLA0KICAgICAgICAiZSI6IC
            JBUUFCIiwNDQogICAgICAgICJraWQiOiAibXl
            QdWJsaWNKU09OV2ViS2V5Ig0KICAgIH0NCn0=" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token"
```

For more information about how to use the different OAuth 2.0 grant flows, see "*OAuth 2.0 Grant Flows*".

The authorization server returns the access token:

- If the authorization server is configured to use *CTS-based* OAuth 2.0 tokens, the response will include an access token ID in the `access_token` property, which identifies the access token data stored on the server. For example:

```
{
    "access_token":"f08f1fcf-3ecb-4120-820d-fb71e3f51c04",
    "scope":"profile",
    "token_type":"Bearer",
    "expires_in":3599
}
```

- If the authorization server is configured to use *client-based* OAuth 2.0 tokens, the response will be a JSON web token in the `access_token`, which has the JWK embedded within. The following example has shortened the access token for display purposes:

```
{
    "access_token": "eyJ0eXAiOiJKV1QiLCHi51zbE3t...zc2NjI3NDgsInNjb3zUOCVKCX0Se0",
    "scope": "profile",
    "token_type": "Bearer",
    "expires_in": 3599
}
```

5. The client now requests access to the protected resources from the resource server.

   If CTS-based OAuth 2.0 tokens are enabled, the resource server can make a POST request to the `/oauth2/introspect` endpoint to acquire the public key. The public key from the original JWK is returned in the `cnf` element:

```
$ curl \
--request POST \
--header "Authorization: Basic bXlDbGllbnQ6Zm9yZ2Vyb2Nr" \
--data "token=f08f1fcf-3ecb-4120-820d-fb71e3f51c04" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/introspect"
{
    "active": true,
    "scope": "profile",
    "client_id": "myClient",
    "user_id": "myClient",
    "token_type": "access_token",
    "exp": 1477666348,
    "sub": "(age!myClient)",
    "subname": "myClient",
    "iss": "https://openam.example.com:8443/openam/oauth2/realms/root",
    "cnf": {
        "jwk": {
            "alg": "RS256",
            "e": "AQAB",
            "n": "xea7Tb7rbQ4ZrHNKrg...QFXtJ-didSTtXWCWU1Qrcj0hnDjvkuUFWoSQ_7Q",
            "kty": "RSA",
            "use": "enc",
            "kid": "myPublicJSONWebKey"
        },
    "auth_level": 0
    }
}
```

6. The resource server should now use the public key to cryptographically confirm proof-of-possession of the token by the presenter; for example, with a challenge-response interaction.

   Successful completion of the challenge-response means that the client must possess the private key that matches the public key presented in the original request, and access to resources can be granted.

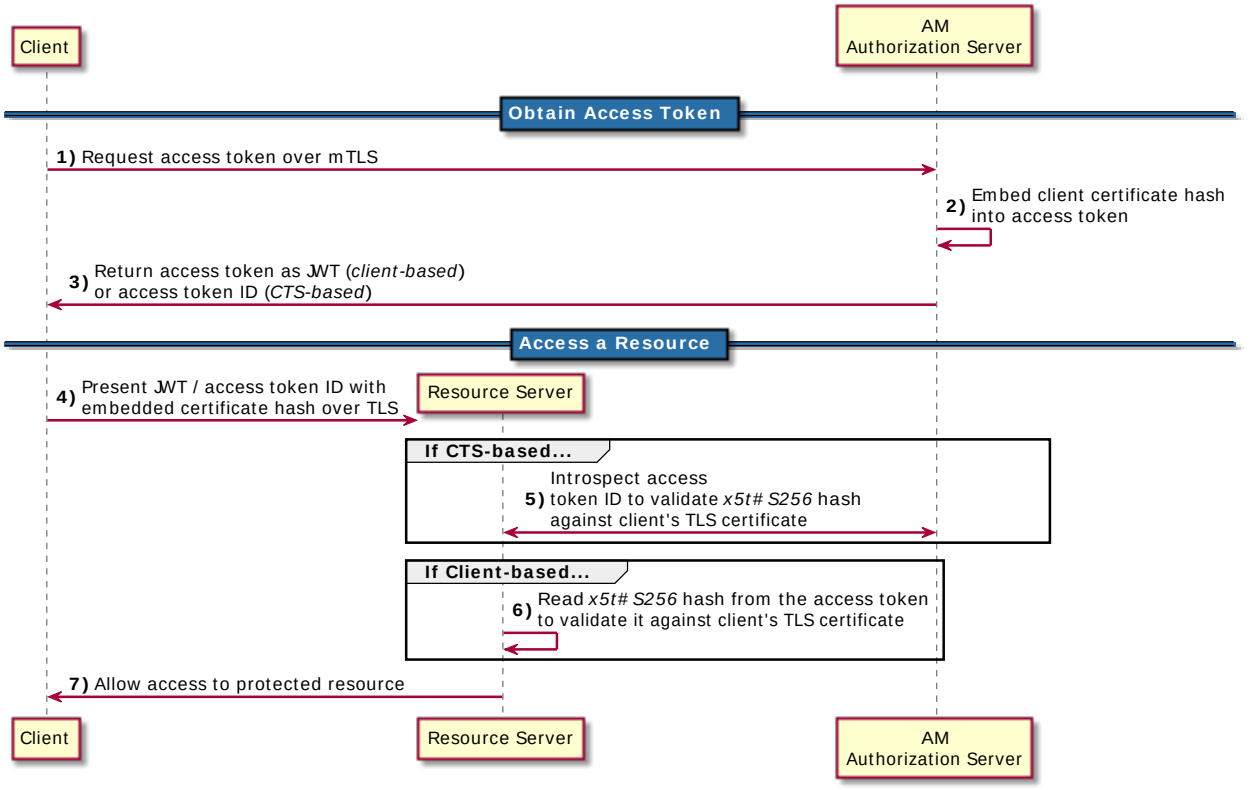
Certificate-Bound Proof-of-Possession

AM supports associating an X.509 certificate with an access token to support proof-of-possession interactions, as per version 12 of the OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens internet-draft.

This ensures that only the client in possession of the private key corresponding to the certificate can use the bearer token to access protected resources.

Since the resource server validates the hash contained in the access token as proof-of-possession against the client's certificate, clients must use the certificate used to request the bearer token when accessing the protected resources. Moreover, this implies that access tokens are invalidated when clients update their certificates.

OAuth 2.0 Certificate-Bound Proof-of-Possession Flow



+ *Certificate-Bound Proof-of-Possession Flow Explained*

The steps in the diagram are described below:

1. The client, communicating over TLS, requests an access token using an OAuth 2.0 grant flow.

   Note

   The Implicit Grant flow does not support certificate-bound proof-of-possession. For more information, see the OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens internet-draft.

2. The authorization server returns the access token to the client with the client's certificate hash embedded:

- If the authorization server is configured for CTS-based OAuth 2.0, the authorization server stores the certificate hash with the access token in the CTS token store and provides the client with the access token ID.
- If the authorization server is configured for client-based OAuth 2.0, the access token is a JWT that contains the certificate hash embedded in it.

The hash of the client's certificate is stored in the `cnf` confirmation key of the type `x5t#S256`, which contains the base64URL-encoded SHA-256 hash of the DER-encoding of the full X.509 certificate.

3. The client, communicating over mTLS, requests access to the protected resources from the resource server.
4. The resource server validates the client's certificate with the certificate hash contained in the access token:
   - If the authorization server is configured for CTS-based OAuth 2.0, the resource server calls the OAuth 2.0 `introspect` endpoint with the access token to recover the `cnf` claim that contains the certificate's hash.
   - If the authorization server is configured for client-based OAuth 2.0, the resource server recovers the `cnf` claim that contains the certificate's hash from the access token JWT.
5. The resource server allows access to the protected resources.

To configure your environment for certificate-bound tokens, see the following sections:

- "Obtaining Certificate-Bound Tokens When Mutual TLS Authentication is Configured"
- "Obtaining Certificate-Bound Tokens Without Configuring Mutual TLS Authentication"

Obtaining Certificate-Bound Tokens When Mutual TLS Authentication is Configured

Clients can authenticate to the OAuth 2.0 endpoints by presenting X.509 self-signed or CA-signed certificates as per version 12 of the OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens internet-draft.

Depending on the type of client, AM performs the following actions:

- **Confidential clients**. When clients present a certificate as the authentication method while making a call to the token endpoint, AM authenticates the client and AM binds the certificate to the access token.
- **Public clients**. When clients present a certificate while making a call to the token endpoint, AM ignores the certificate for authentication purposes and binds the certificate to the access token.

To Obtain Certificate-Bound Tokens When Authenticating with Mutual TLS

Perform the steps in the following procedure to obtain a certificate-bound access token when a client authenticates using mutual TLS:

1. Ensure your environment enforces TLS between the authorization server and the clients, and between the resource server and the clients. Self-signed and CA-signed certificates are supported.

   You must configure the container where AM runs to request and accept client certificates.

2. Configure AM as an OAuth 2.0 authorization server using the following information:

   - You must enable the Support TLS Certificate-Bound Access Tokens switch (Realms > *Realm Name* > Services > OAuth2 Provider > Advanced).

     This property specifies whether AM should bind certificates to access tokens when clients authenticate using TLS client certificates.

   - If TLS is being terminated at a reverse proxy or load balancer, you must configure the Trusted TLS Client Certificate Header property (Realms > *Realm Name* > Services > OAuth2 Provider > Advanced) to hold the name of the HTTP header that will provide AM with the client certificate.

     For more information, see "Providing Client Certificates to AM".

3. Register an OAuth 2.0 client in AM. The following configuration will be used in the examples of this procedure:

   - **Client ID**: `myClient`
   - **Scopes**: `write`
   - **Grant Types**: `Client Credentials`
   - You must enable the Use Certificate-Bound Access Tokens switch (Realms > *Realm Name* > Applications > OAuth 2.0 > *Client Name* > Signing and Encryption).

     This switch specifies whether AM should bind certificates to access tokens for this client when the client authenticates to the token endpoint using a TLS client certificate. When disabled, AM does not bind certificates to access tokens issued to the client even if the client presents a TLS client certificate.

4. Configure the client for mutual TLS authentication. For more information, see "Authenticating Clients Using Mutual TLS".

5. The client makes a call to the token endpoint to request an access token, and includes its client certificate in the call:

```
$ curl --request POST \
--cacert AMServer.cer \
--data "client_id=myClient" \
--data "grant_type=client_credentials" \
--data "scope=write" \
--data "response_type=token" \
--cert myClientCertificate.pem \
--key myClientCertificate.key.pem \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token"
```

The authorization server returns the access token:

- If CTS-based OAuth 2.0 tokens are enabled, the response will include an access token ID in the `access_token` property, which identifies the access token data stored on the server. For example:

```
{
    "access_token":"f08f1fcf-3ecb-4120-820d-fb71e3f51c04",
    "scope":"profile",
    "token_type":"Bearer",
    "expires_in":3599
}
```

- If client-based OAuth 2.0 tokens are enabled, the response will be a JWT in the `access_token`, which has the JWK embedded within. The following example has shortened the access token for display purposes:

```
{
    "access_token": "eyJ0eXAiOiJKV1QiLCHi51zbE3t...zc2NjI3NDgsInNjb3zUOCVKCX0Se0",
    "scope": "profile",
    "token_type": "Bearer",
    "expires_in": 3599
}
```

6. The client requests access to the protected resources. The resource server validates the hash contained in the access token against the certificate the client presents as part of the TLS handshake.

   The hash contained in the access token is stored in the `cnf` confirmation key of the type `x5t#S256`, which contains the base64URL-encoded SHA-256 hash of the DER-encoding of the full X.509 certificate.

   If CTS-based OAuth 2.0 tokens are enabled, the resource server can make a POST request to the introspect endpoint to acquire the certificate's hash:

```
$ curl \
--request POST \
--header "Authorization: Basic bXlDbGllbnQ6Zm9yZ2Vyb2Nr" \
--data "token=f08f1fcf-3ecb-4120-820d-fb71e3f51c04" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/introspect"
    {
   "active":true,
   "scope":"write",
   "client_id":"myClient",
   "user_id":"myClient",
   "token_type":"Bearer",
   "exp":1547079953,
   "sub":"(age!myClient)",
   "subname":"myClient"
   "iss":"https://openam.example.com:8443/openam/oauth2",
   "cnf":{
      "x5t#S256":"m8UcWBSPNtaKN19TdR8zUHvWWOSCSX9nsa5vU6fscd0"
   }
}
```

If client-based OAuth 2.0 tokens are enabled, the resource server can decode the JWT to access the `cnf` key in the JWT's payload. For example:

```
{
  "sub": "myClient",
  "cts": "OAUTH2_STATELESS_GRANT",
  ....
  "cnf": {
     "x5t#S256": "m8UcWBSPNtaKN19TdR8zUHvWWOSCSX9nsa5vU6fscd0"
  },
  "exp": 1547083590,
  "iat": 1547079990,
  "expires_in": 3600,
  "jti": "sLzkRiayAQKsrXN0Gu_vwFog3Rs"
}
```

Obtaining Certificate-Bound Tokens Without Configuring Mutual TLS Authentication

Clients can obtain a certificate-bound access token when making a call to the OAuth 2.0 endpoints as long as they provide an X.509 client certificate in one of the following ways:

- Presenting a self-signed or CA-signed certificate as part of the TLS handshake with AM.

  AM authenticates the clients using the specified credentials (for example, client ID and secret) and binds the certificate to the access token.

  Your environment must enforce TLS between the authorization server and the clients, and between the resource server and the clients.

  You must also configure the container where AM runs to request and accept client certificates.

- Providing a hash of the self-signed or CA-signed certificate in the `cnf_key` parameter as part of the call to the OAuth 2.0 endpoint.

This method uses capabilities already implemented in AM that are not part of the OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens internet-draft.

Use this option only if the client cannot authenticate its TLS connection to AM.

To Obtain Certificate-Bound Tokens Without Using Mutual TLS for Authentication

Perform the steps in the following procedure to obtain a certificate-bound access token when clients are not authenticating with mutual TLS:

1. Configure AM as an OAuth 2.0 authorization server using the following information:

   - You must enable the Support TLS Certificate-Bound Access Tokens switch (Realms > *Realm Name* > Services > OAuth2 Provider > Advanced).

     This property specifies whether AM should bind certificates to access tokens when clients authenticate using TLS client certificates.

   - If not using the `cnf_key`, and if TLS is being terminated at a reverse proxy or load balancer, you must configure the Trusted TLS Client Certificate Header property (Realms > *Realm Name* > Services > OAuth2 Provider > Advanced) to hold the name of the HTTP header that will provide AM with the client certificate.

     For more information, see "Providing Client Certificates to AM".

2. Register a client in AM. The following configuration will be used in the examples of this procedure:

   - **Client ID**: `myClient`
   - **Scopes**: `write`
   - **Grant Types**: `Client Credentials`
   - For confidential clients, configure a secret. For example:
     - **Client Secret**: `forgerock`
   - You must enable the Use Certificate-Bound Access Tokens switch (Realms > *Realm Name* > Applications > OAuth 2.0 > *Client Name* > Signing and Encryption).

     This switch specifies whether AM should bind certificates to access tokens for this client when the client authenticates to the token endpoint using a TLS client certificate. When disabled, AM does not bind certificates to access tokens issued to the client even if the client presents a TLS client certificate.

3. The client makes a call to the token endpoint to request an access token, and includes its client certificate in the call:

```
$ curl --request POST \
--cacert AMServer.cer \
--data "client_id=myClient" \
--data "client_secret=forgerock" \
--data "grant_type=client_credentials" \
--data "scope=write" \
--data "response_type=token" \
--cert myClientCertificate.pem \
--key myClientCertificate.key.pem \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token"
```

Tip

To use the `cnf_key` parameter, the client must perform the following additional steps:

- Calculate the SHA-256 hash of the DER-encoding of the full X.509 client certificate and base64URL-encode it. For example:

  ```
  m8UcWBSPNtaKN19TdR8zUHvWWOSCSX9nsa5vU6fscd0
  ```

- Store the certificate's hash in JSON format, as follows:

  ```
  {"x5t#S256":"m8UcWBSPNtaKN19TdR8zUHvWWOSCSX9nsa5vU6fscd0"}
  ```

- Base64-encode the JSON. For example:

  ```
  eyJ4NXQjUzI1NiI6Im04VWNXQlNQTnRhS04xOVRkUjh6VUh2V1dPU0NTWDluc2E1dlU2ZnNjZDAifQ==
  ```

- Make a call to the token endpoint to request an access token, including the `cnf_key` parameter with the certificate hash. Note that the client certificate is not included in any other way:

  ```
  $ curl \
  --request POST \
  --data "grant_type=client_credentials"\
  --data "client_id=myClient" \
  --data "client_secret=forgerock" \
  --data "cnf_key=eyJ4NXQjUzI1NiI6Im04
                  VWNXQlNQTnRhS04xOVRk
                  Ujh6VUh2V1dPU0NTWDlu
                  c2E1dlU2ZnNjZDAifQ==" \
  "https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token"
  ```

The authorization server returns the access token:

- If CTS-based OAuth 2.0 tokens are enabled, the response will include an access token ID in the `access_token` property, which identifies the access token data stored on the server. For example:

  ```
  {
      "access_token":"f08f1fcf-3ecb-4120-820d-fb71e3f51c04",
      "scope":"profile",
      "token_type":"Bearer",
      "expires_in":3599
  }
  ```

- If client-based OAuth 2.0 tokens are enabled, the response will be a JSON web token in the `access_token`, which has the certificate hash embedded within. The following example has shortened the access token for display purposes:

```
{
    "access_token": "eyJ0eXAiOiJKV1QiLCHi51zbE3t...zc2NjI3NDgsInNjb3zUOCVKCX0Se0",
    "scope": "profile",
    "token_type": "Bearer",
    "expires_in": 3599
}
```

4. The client requests access to the protected resources from the resource server and the resource server validates the hash contained in the access token against the certificate the client presents as part of the TLS handshake.

   The hash contained in the access token is stored in the `cnf` confirmation key of the type `x5t#S256`, which contains the base64URL-encoded SHA-256 hash of the DER-encoding of the full X.509 certificate.

   If CTS-based OAuth 2.0 tokens are enabled, the resource server can make a POST request to the introspect endpoint to acquire the certificate's hash:

```
$ curl \
--request POST \
--header "Authorization: Basic bXlDbGllbnQ6Zm9yZ2Vyb2Nr" \
--data "token=f08f1fcf-3ecb-4120-820d-fb71e3f51c04" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/introspect"
{
   "active":true,
   "scope":"write",
   "client_id":"myClient",
   "user_id":"myClient",
   "token_type":"Bearer",
   "exp":1547079953,
   "sub":"(age!myClient)",
   "subname":"myClient"
   "iss":"https://openam.example.com:8443/openam/oauth2",
   "cnf":{
      "x5t#S256":"m8UcWBSPNtaKN19TdR8zUHvWWOSCSX9nsa5vU6fscd0"
   }
}
```

   If client-based OAuth 2.0 tokens are enabled, the resource server can decode the JWT to access the `cnf` key in the JWT's payload. For example:

```
{
  "sub": "myClient",
  "cts": "OAUTH2_STATELESS_GRANT",
  ....
  "cnf": {
    "x5t#S256": "m8UcWBSPNtaKN19TdR8zUHvWWOSCSX9nsa5vU6fscd0"
  },
  "exp": 1547083590,
  "iat": 1547079990,
  "expires_in": 3600,
  "jti": "sLzkRiayAQKsrXN0Gu_vwFog3Rs"
}
```

Chapter 10

Refresh Tokens

Refresh tokens (*RFC 6749*) are a type of token that can be used to obtain a new access token that may have identical or narrower scopes than the original. AM can issue refresh tokens during every OAuth 2.0/OpenID Connect grant flow except for the Implicit and the Client Credentials grant flows.

+ *Why Are Refresh Tokens Useful?*

Access tokens are short-lived because, if leaked, they grant potentially malicious users access to the resource owner resources. However, clients may need to access the protected data for periods of time that exceed the access token lifetime or when the resource owner is not available. In some cases, it is unreasonable to ask for the resource owner's consent several times during the same operation.

Refresh tokens solve this problem by letting clients ask for a new access token without further interaction from the resource owner. While a potentially malicious user compromising an access token has access to the resource owner resources, one that holds a refresh token also needs to compromise the client ID and the client secret to be able to get an access token, since the client needs to authenticate to the token endpoint to obtain an access token using the refresh token.

Refresh tokens are long-lived by default, and AM lets you configure the lifetime of the tokens in the OAuth 2.0 Provider settings, or in each client. By default, the configuration of the OAuth 2.0 Provider is used. For more information, see Advanced Properties and "OAuth2 Provider" in the *Reference* .

Refresh tokens can also be revoked. For more information, see "/oauth2/token/revoke").

Tasks:

- "To Configure AM to Issue Refresh Tokens"
- "To Refresh an Access Token"

To Configure AM to Issue Refresh Tokens

AM can issue refresh tokens during the following actions:

- When issuing an access token to the client after a successful OAuth 2.0 grant flow.
- When the client successfully uses a refresh token to obtain a new access token. Note that, when a new refresh token is issued, the old refresh token is deactivated.

1. To enable AM to issue refresh tokens at the same time the access token is issued, navigate to Realms > *Realm Name* > Services > OAuth2 Provider > Core, and enable Issue Refresh Tokens.

   Note that you configure refresh tokens at realm level. Consider carefully the types of clients registered to the realm before configuring AM to issue refresh tokens.

2. (Optional) To enable AM to also issue refresh tokens when refreshing access tokens, navigate to Realms > *Realm Name* > Services > OAuth2 Provider > Core, and enable Issue Refresh Tokens on Refreshing Access Tokens.

3. Save your changes.

4. To configure a client to use the `Refresh Token` grant flow perform the following steps:

   a. Navigate to Realms > *Realm Name* > Applications > OAuth 2.0 > *Client Name* > Advanced.

   b. On the Grant Types field, add the `Refresh Token` grant type.

   c. Save your changes.

To Refresh an Access Token

This procedure assumes the following configuration:

- AM is configured as an OAuth 2.0 authorization server in the with the following configuration:
  - **Issue Refresh Tokens** is enabled.
  - **Issue Refresh Tokens on Refreshing Access Tokens** is enabled.
  - The `Refresh Token` grant type is configured in the Grant Types field.

  For more information, see "*Authorization Server Configuration*".
- A confidential client called `myClient` is registered in AM with the following configuration:
  - **Client secret**: `forgerock`
  - **Scopes**: `write read`
  - **Grant Types**: `Authorization Code Refresh Token`

Perform the steps in the procedure to refresh an access token:

1. The client obtains an access token and a refresh token using the Authorization Code Grant flow. For more information, see "Authorization Code Grant".

   The example assumes the refresh token is `qz1qx-9AYOkRp3AWcCZULvPitpM`.

2. The client makes a POST call to the authorization's server token endpoint, specifying, at least, the following parameters:

- **grant_type**=refresh_token
- **refresh_token**=*your_refresh_token*

For more information about the parameters supported by the `/oauth2/access_token` endpoint, see "/oauth2/access_token".

Confidential clients can authenticate to the OAuth 2.0 endpoints in several ways. This example uses the following form parameters:

- **client_id**=*your_client_id*
- **client_secret**=*your_client_secret*

For more information, see "*OAuth 2.0 Client Authentication*".

If the OAuth 2.0 provider is configured for a subrealm rather than the Top Level Realm, you must specify it in the endpoint. For example, if the OAuth 2.0 provider is configured for the `/alpha` realm, then use `/oauth2/realms/root/realms/alpha/access_token`.

For example:

```
$ curl --request POST \
--data "grant_type=refresh_token" \
--data "refresh_token=qz1qx-9AYOkRp3AWcCZULvPitpM" \
--data "client_id=myClient" \
--data "client_secret=forgerock" \
--data "scope=read" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token"
{
   "access_token":"y-C_A1RKJIg-BUlKhp--kv5Iywk",
   "refresh_token":"qdqVnFJK8FjiQAjYMaBuUY6z_HU",
   "scope":"read",
   "token_type":"Bearer",
   "expires_in":3599
}
```

Note that the `scope` parameter is not required. By default, AM will issue an access token with the same scopes of the original. This example is restricting the new access token to the `read` scope.

Also note that AM has issued a new refresh token; the original refresh token is now inactive.

Chapter 11

Macaroons as Access and Refresh Tokens

Macaroons are a type of bearer token that can be used when issuing OAuth 2.0 access and refresh tokens. They can be used in place of regular access or refresh tokens, as they allow the sharing of a single token with multiple clients and resource servers without compromising security.

The idea behind it is that, rather than issuing multiple access tokens with different scopes for a set of clients, AM issues a macaroon access token with a broad scope to a client. The client then creates as many macaroons as needed from the single macaroon access token, restricting their scopes as required by using *caveats*. This is very useful, for example, in a microservice architecture where a single client can delegate tasks to other services, with a limited set of capabilities or bound by certain restrictions.

Caveats are restrictions placed on a macaroon that must be satisfied before using the token. Meaning, for example, that if the expiry time is past, the token is invalid.

Caveats that can be satisfied locally are referred to as *first-party* caveats, and caveats satisfied by a service external from AM are referred to as *third-party* caveats. Support for third-party caveats and discharge macaroons in AM is *evolving*.

+ *About Third-Party Caveats and Discharge Macaroons*

Third-party caveats are those that require the client to use a service other than AM to get proof that the condition specified by the caveat is satisfied. They are useful in situations where you have services external to AM that can make additional authorization checks relevant to the access token.

For example, consider a case where you have a service external to AM, akin to an IDP in a SAML v2.0 architecture, that you can query to know if the user related to the access token belongs to a particular user group.

The proof that the condition is satisfied is returned by the third party in a *discharge macaroon*, and the client must present both the access token macaroon and the discharge macaroon to get access to the resource.

The discharge macaroon can also have first-party caveats attached to it, such as expiry time. This allows for flows where the access token macaroon is long-lived and the discharge macaroon is not, which forces the client to acquire a new discharge macaroon to access the resource.

Caution

Any first-party caveats attached to the discharge macaroons will be treated as if they were caveats on the access token itself. For example, if the discharge macaroon limits the expiry time to five minutes, the

introspection response will list the expiry time of the access token as five minutes even if the access token was valid for longer.

Another possible use case is related to transactional authorization. Consider a case where a payment is tied to a unique transaction; you could create a macaroon access token containing a third-party caveat that requires the client to obtain a one-time discharge macaroon from an external transactional service.

A third-party caveat has the following parts:

- A hint describing where the client can find the third-party service, which is usually a URL.
- A unique secret key to sign discharge macaroons, known as the *discharge key*.
- An identifier for the third-party service to know which condition needs to be checked, and how to recover the discharge key.

  There is no standard format for the identifier part.

+ *Which First-Party Caveats Does AM Support?*

There is no standard format for caveats in Macaroons, so AM has adopted a JSON-based syntax that mirrors the existing JWT-based token restrictions:

`scope`

Restricts the scope of the token. The returned scope will be the intersection of the original token scope and any `scope` caveats.

`exp`

Restricts the expiry time of the token. The effective expiry time is the minimum of the original expiry time and any expiry caveats added to the token. If you have appended more than one `exp` caveat, the most restrictive one applies.

`cnf`

Binds the access token to a client certificate. This means that a client can be issued with a regular access token and then can later bind it to their client certificate. You can only bind one client certificate to the macaroon. Any attempt to bind a new certificate with subsequent caveats is ignored.

`aud`

Restricts the audience of the token. The effective audience is the intersection of any audience restriction and any `aud` caveats.

AM returns any other caveats in a `caveats` object on the JSON introspection response.

Tip

OpenID Connect clients must ensure that the following information is present in the JSON:

- The `openid` scope. For example, `"scopes": ["profile", "openid"]`.
- The `id_token` response type. For example, `"response_types": ["code", "id_token code"]`.

Appending Caveats to Macaroons

You append caveats to a macaroon through a macaroon library of your choosing. AM provides the `/json/token/macaroon` endpoint to add first-party caveats to a macaroon. You can also use this endpoint to inspect the caveats already appended.

You can also add caveats to the macaroon before AM issues it using access token modification scripts. AM can append third-party caveats by using access token modification scripts only.

Using OAuth 2.0 Endpoints with Macaroons

AM endpoints that support access tokens also support macaroons without further configuration. Endpoints will reject macaroons whose caveats are not satisfied.

When dealing with macaroons containing third-party caveats, use the `X-Discharge-Macaroon` header to pass a discharge macaroon.

Macaroons and CTS-Based and Client-Based Tokens

Macaroons are layered on top of the existing CTS-Based OAuth 2.0 tokens and Client-Based OAuth 2.0 tokens. When you enable macaroons, AM will issue one of the following:

- CTS-Based Macaroon Tokens: the access token is stored in the CTS, and macaroons are the tokens issued to clients, where the identifier of the Macaroon is the pointer to the access token in the CTS.
- Client-Based Macaroon Tokens: the access token is a signed and/or encrypted JWT, which is then wrapped in a Macaroon. Note that the resulting size of the token may impact your deployment. If client storage is limited, such as when using browser cookies, the token may be too large to store. Token size may also impact network performance.

Enabling Macaroons

Follow these steps to enable macaroons in the OAuth 2.0 Provider Service:

1. In the AM console, go to Realms > *Realm Name* > Services > OAuth2 Provider.
2. On the Core tab, enable Use Macaroon Access and Refresh Tokens.
3. On the Advanced tab, select the Macaroon Token Format.

   Note

   It is recommended that you use the default `V2`. Configuring Macaroons to use the older token format `V1` is much less efficient. It should only be used when compatibility with older Macaroon libaries is required.

4. Set the OAuth2 Token Signing Algorithm to `HS256`, or higher.
5. Save your changes.
6. Ensure that the `am.services.oauth2.jwt.authenticity.signing` secret ID is mapped either in the realm, or globally. AM uses this secret ID mapping to sign and verify macaroon access and refresh tokens.

   For information about secret stores, see "Configuring Secret Stores" in the *Security Guide*.

Chapter 12

OAuth 2.0 Grant Flows

This chapter describes the OAuth 2.0 flows that AM supports, and also provides the information required to implement them. All the examples assume the realm is configured for CTS-based tokens, however, the examples also apply to client-based tokens.

You should decide which flow is best for your environment based on the application that will be the OAuth 2.0 client. The following table provides an overview of the flows AM supports and when they should be used:

Deciding Which Flow to Use Depending on the OAuth 2.0 Client

Client Type	Which Grant to use?	Description
The client is a web application running on a server. For example, a `.war` application.	**Authorization Code**	(*RFC 6749*) The authorization server uses the user-agent, for example, the resource owner's browser, to transport a code that is later exchanged for an access token.
The client is a native application or a single-page application (SPA). For example, a desktop, a mobile application, or a JavaScript application.	**Authorization Code with PKCE**	Authorization Code with PKCE (*RFC 6749*, *RFC 7636*) The authorization server uses the user-agent, for example, the resource owner's browser, to transport a code that is later exchanged for an access token. Since the client does not communicate securely with the authorization server, the code may be intercepted by malicious users. The implementation of the Proof Key for Code Exchange (PKCE) standard mitigates against those attacks.
The client is a SPA. For example, a JavaScript application.	**Implicit**	(*RFC 6749*) The authorization server gives the access token to the user-agent so it can forward the token to the client. Therefore, the access token might be exposed to the user and other applications. For security purposes, you should use the Authorization Code grant with PKCE when possible.
The client is trusted with the resource owner credentials. For example, the resource owner's operating system.	**Resource Owner Password Credentials**	(*RFC 6749*) The resource owner provides their credentials to the client, which uses them to obtain an access token from the authorization server.

Client Type	Which Grant to use?	Description
		This flow should only be used if other flows are not available.
The client is the resource owner, or the client does not act on behalf of the resource owner.	**Client Credentials**	(*RFC 6749*) Similar to the Resource Owner Password Credentials grant type, but the resource owner is not part of the flow and the client accesses information relevant to itself.
The client is an input-constrained device. For example, a TV set.	**Device Flow**	(*OAuth 2.0 Device Flow for Browserless and Input Constrained Devices*) The resource owner authorizes the client to access protected resources on their behalf by using a different user-agent and entering a code displayed on the client device.
The client has a SAML v2.0 trust relationship with the resource owner. For example, an application in an environment where a SAML v2.0 ecosystem coexists with an OAuth 2.0 one.	**SAML v2.0 Profile**	(*RFC 7522*) The client uses the resource owner's SAML v2.0 assertion to obtain an access token from the authorization server without interacting with the resource owner again.
The client has a trust relationship with the resource owner that is specified as a JWT. For example, an application in an environment where a non-SAML v2.0 identity ecosystem coexists with an OAuth 2.0 one.	**JWT Bearer Profile**	(*RFC 7523*) The client uses a signed JWT to obtain an access token from the authorization server without interacting with the resource owner.

ForgeRock provides a Postman collection to try out the flows. See "ForgeRock Grant Flows Collection".

Tip

AM supports associating a confirmation key or a certificate with an access token to support proof-of-possession interactions.

For more information, see "Certificate-Bound Proof-of-Possession" and "JWK-Based Proof-of-Possession".

ForgeRock Grant Flows Collection

ForgeRock provides an OAuth 2.0 and OpenID Connect Postman collection to try out the flows that AM support. The source for the REST calls, including the prerequisites needed to run the collection, is provided as a downloadable JSON file collection.

1. Download and install Postman.
2. Download the ForgeRock OAuth 2.0 and OpenID Connect Collection.

3. Import the collection in Postman:

   a. Go to File > Import ... > Upload Files.

   b. Select the collection you downloaded, and click Open. Then, click Import.

4. Configure the collection's variables to suit your environment:

   a. In Postman, on the Collections tab, select the ForgeRock OAuth 2.0 and OpenID Connect Collection. Click on the **...** button, and then on Edit.

      The Edit Collection page appears.

   b. Click on the Variables tab, and change at least the value of the following variables:

      - `URL_base`
      - `admin_password`

   c. Click Update to save your changes.

      You are ready to start running the collection.

   The collection is divided into the following folders:

   - `Prerequisites`, containing REST calls to configure AM as an authorization server, and to create the clients and users required to run the collection.
   - `OAuth 2.0 Flows`, containing the flows explained in "*OAuth 2.0 Grant Flows*".
   - `OpenID Connect Flows`, containing the flows explained in "*OpenID Connect Grant Flows*" in the *OpenID Connect 1.0 Guide*.

     The Backchannel (CIBA) grant is not included, since it requires push notifications and an additional device to work.
   - `Refresh Token Flow`, containing calls explained in "*Refresh Tokens*" and "/oauth2/token/revoke".
   - `Token Exchange Flows`, containing the token exchange flows explained in "Token Exchange Flows".

Authorization Code Grant

Endpoints

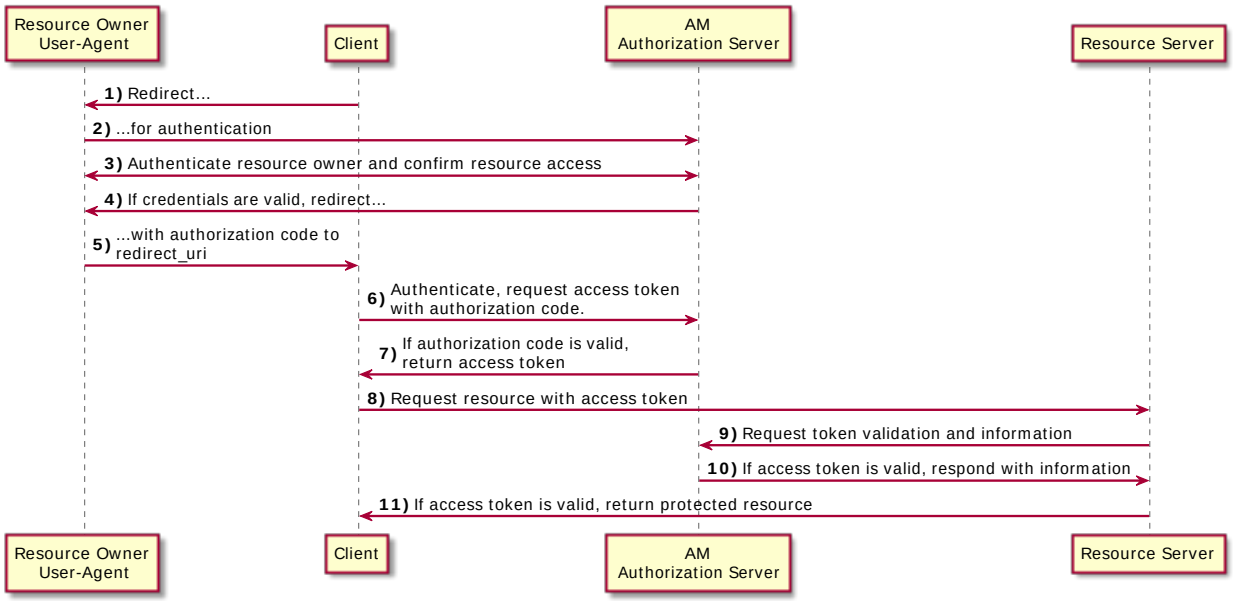
- /oauth2/authorize
- /oauth2/access_token

The Authorization Code grant is a two-step interactive process used when the client, for example, a Java application running on a server, requires access to protected resources.

The Authorization Code grant is the most secure of all the OAuth 2.0 grants for the following reasons:

- It is a two-step process. The user must authenticate and authorize the client to see the resources and the authorization server must validate the code again before issuing the access token.
- The authorization server delivers the access token directly to the client, usually over HTTPS. The client secret is never exposed publicly, which protects confidential clients.

OAuth 2.0 Authorization Code Grant Flow



+ *Authorization Code Grant Flow Explained*

The steps in the diagram are described below:

1. The client, usually a web-based service, receives a request to access a protected resource. To access the resources, the client requires authorization from the resource owner.
2. The client redirects the resource owner's user-agent to the authorization server.
3. The authorization server authenticates the resource owner, confirms resource access, and gathers consent if not previously saved.

4. The authorization server redirects the resource owner's user agent to the client.
5. During the redirection process, the authorization server appends an authorization code.
6. The client receives the authorization code and authenticates to the authorization server to exchange the code for an access token.

   Note that this example assumes a confidential client. Public clients are not required to authenticate.
7. If the authorization code is valid, the authorization server returns an access token (and a refresh token, if configured) to the client.
8. The client requests access to the protected resources from the resource server.
9. The resource server contacts the authorization server to validate the access token.
10. The authorization server validates the token and responds to the resource server.
11. If the token is valid, the resource server allows the client to access the protected resources.

Perform the steps in the following procedures to obtain an authorization code and exchange it for an access token:

- "To Obtain an Authorization Code Using a Browser in the Authorization Code Grant Flow"
- "To Obtain an Authorization Code Without Using a Browser in the Authorization Code Grant Flow"
- "To Exchange an Authorization Code for an Access Token"

To Obtain an Authorization Code Using a Browser in the Authorization Code Grant Flow

This procedure assumes the following configuration:

- AM is configured as an OAuth 2.0 authorization server. Ensure that:
  - The `code` plugin is configured in the Response Type Plugins field.
  - The `Authorization Code` grant type is configured in the Grant Types field.

  For more information, see "*Authorization Server Configuration*".
- A confidential client called `myClient` is registered in AM with the following configuration:
  - **Client secret**: `forgerock`
  - **Scopes**: `write`

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

For more information, see "*Client Registration*".

Perform the steps in this procedure to obtain an authorization code using a browser:

1. The client redirects the resource owner's user-agent to the authorization server's authorization endpoint specifying, at least, the following form parameters:

   - **client_id**=*your_client_id*
   - **response_type**=code
   - **redirect_uri**=*your_redirect_uri*

   For information about the parameters supported by the `/oauth2/authorize` endpoint, see "/oauth2/authorize".

   If the OAuth 2.0 provider is configured for a subrealm rather than the Top Level Realm, you must specify it in the endpoint. For example, if the OAuth 2.0 provider is configured for the `/alpha` realm, then use `/oauth2/realms/root/realms/alpha/authorize`.

   For example:

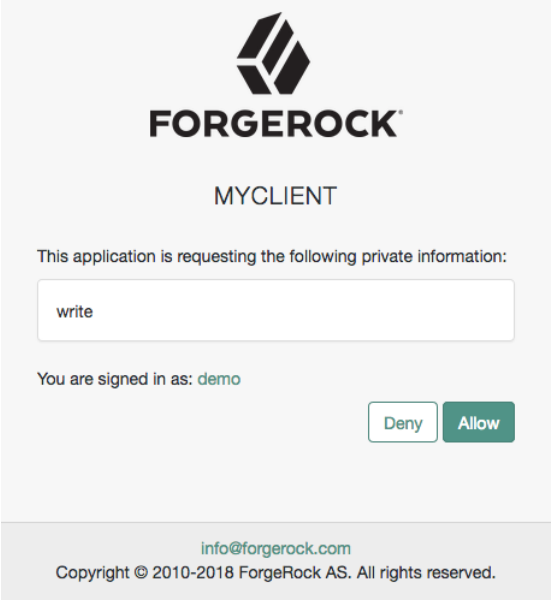
   ```
   https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/authorize \
   ?client_id=myClient \
   &response_type=code \
   &scope=write \
   &state=abc123 \
   &redirect_uri=https://www.example.com:443/callback
   ```

   Note that the URL is split and spaces have been added for readability purposes and that the `scope` and `state` parameters have been included. Scopes are not required, since they can be configured by default in the authorization server and the client, and have been added only as an example. The `state` parameter is added to protect against CSRF attacks.

2. The resource owner authenticates to the authorization server, for example, using the credentials of the `demo` user. In this case, they log in using the default chain or tree configured for the realm.

   After logging in, the authorization server presents the AM consent screen:

OAuth 2.0 Consent Screen



3. The resource owner selects the `Allow` button to grant consent for the `write` scope.

   The authorization server redirects the resource owner to the URL specified in the `redirect_uri` parameter.

4. Inspect the URL in the browser. It contains a `code` parameter with the authorization code the authorization server has issued. For example:

   ```
   http://www.example.com/?code=g5B3qZ8rWzKIU2xodV_kkSIk0F4&scope=write&iss=https%3A%2F
   %2Fopenam.example.com%3A8443%2Fopenam%2Foauth2&state=abc123&client_id=myClient
   ```

5. The client performs the steps in "To Exchange an Authorization Code for an Access Token" to exchange the authorization code for an access token.

To Obtain an Authorization Code Without Using a Browser in the Authorization Code Grant Flow

This procedure assumes the following configuration:

- AM is configured as an OAuth 2.0 authorization server. Ensure that:
  - The `code` plugin is configured in the Response Type Plugins field.
  - The `Authorization Code` grant type is configured in the Grant Types field.

For more information, see "*Authorization Server Configuration*".

- A confidential client called `myClient` is registered in AM with the following configuration:
  - **Client secret**: `forgerock`
  - **Scopes**: `write`
  - **Response Types**: `code`
  - **Grant Types**: `Authorization Code`

For more information, see "*Client Registration*".

Perform the steps in this procedure to obtain an authorization code without using a browser:

1. The resource owner logs in to the authorization server, for example, using the credentials of the `demo` user. For example:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "X-OpenAM-Username: demo" \
--header "X-OpenAM-Password: Ch4ng31t" \
--header "Accept-API-Version: resource=2.0, protocol=1.0" \
'https://openam.example.com:8443/openam/json/realms/root/realms/alpha/authenticate'
{
    "tokenId":"AQIC5wM...TU3OQ*",
    "successUrl":"/openam/console",
    "realm":"/alpha"
}
```

2. The client makes a POST call to the authorization server's authorization endpoint, specifying the SSO token of the `demo` in a cookie and, at least, the following parameters:
   - **client_id**=*your_client_id*
   - **response_type**=code
   - **redirect_uri**=*your_redirect_uri*
   - **decision**=allow
   - **csrf**=*demo_user_SSO_token*

   For information about the parameters supported by the `/oauth2/authorize` endpoint, see "/oauth2/authorize".

   If the OAuth 2.0 provider is configured for a subrealm rather than the Top Level Realm, you must specify it in the endpoint. For example, if the OAuth 2.0 provider is configured for the `/alpha` realm, then use `/oauth2/realms/root/realms/alpha/authorize`.

For example:

```
$ curl --dump-header - \
--request POST \
--Cookie "iPlanetDirectoryPro=AQIC5wM...TU3OQ*" \
--data "scope=write" \
--data "response_type=code" \
--data "client_id=myClient" \
--data "csrf=AQIC5wM...TU3OQ*" \
--data "redirect_uri=https://www.example.com:443/callback" \
--data "state=abc123" \
--data "decision=allow" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/authorize"
```

Note that the `scope` and the `state` parameters have been included. Scopes are not required, since they can be configured by default in the authorization server and the client, and have been added only as an example. The `state` parameter is added to protect against CSRF attacks.

If the authorization server is able to authenticate the user and the client, it returns an HTTP 302 response with the authorization code appended to the redirection URL:

```
HTTP/1.1 302 Found
Server: Apache-Coyote/1.1
X-Frame-Options: SAMEORIGIN
Pragma: no-cache
Cache-Control: no-store
Date: Mon, 30 Jul 2018 11:42:37 GMT
Accept-Ranges: bytes
Location: https://www.example.com:443/callback?code=g5B3qZ8rWzKIU2xodV_kkSIk0F4&scope=write&iss=https
%3A%2F%2Fopenam.example.com%3A8443%2Fopenam%2Foauth2&state=abc123&client_id=myClient
Vary: Accept-Charset, Accept-Encoding, Accept-Language, Accept
Content-Length: 0
```

3. Perform the steps in "To Exchange an Authorization Code for an Access Token" to exchange the authorization code for an access token.

To Exchange an Authorization Code for an Access Token

Perform the steps in the following procedure to exchange an authorization code for an access token:

1. Ensure the client has obtained an authorization code by performing the steps in either "To Obtain an Authorization Code Using a Browser in the Authorization Code Grant Flow" or "To Obtain an Authorization Code Without Using a Browser in the Authorization Code Grant Flow".

2. The client creates a POST request to the token endpoint in the authorization server specifying, at least, the following parameters:

   - **grant_type**=authorization_code
   - **code**=*your_authorization_code*
   - **redirect_uri**=*your_redirect_uri*

For information about the parameters supported by the `/oauth2/access_token` endpoint, see "/oauth2/access_token".

Confidential clients can authenticate to the OAuth 2.0 endpoints in several ways. This example uses the following form parameters:

- **client_id**=*your_client_id*
- **client_secret**=*your_client_secret*

For more information, see "*OAuth 2.0 Client Authentication*".

If the OAuth 2.0 provider is configured for a subrealm rather than the Top Level Realm, you must specify it in the endpoint. For example, if the OAuth 2.0 provider is configured for the `/alpha` realm, then use `/oauth2/realms/root/realms/alpha/access_token`.

For example:

```
$ curl --request POST \
--data "grant_type=authorization_code" \
--data "code=g5B3qZ8rWzKIU2xodV_kkSIk0F4" \
--data "client_id=myClient" \
--data "client_secret=forgerock" \
--data "redirect_uri=https://www.example.com:443/callback" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token"
```

The `client_id` and the `redirection_uri` parameters specified in this call must match those used as part of the authorization code request, or the authorization server will not validate the code.

The authorization server returns an access token in the `access_token` property. For example:

```
{
  "access_token": "sbQZuveFumUDV5R1vVBl6QAGNB8",
  "scope": "write",
  "token_type": "Bearer",
  "expires_in": 3599
}
```

Tip

The authorization server can also issue refresh tokens at the same time the access tokens are issued. For more information, see "*Refresh Tokens*".

Authorization Code Grant with PKCE

Endpoints

- /oauth2/authorize

- /oauth2/access_token

The Authorization Code grant, when combined with the PKCE standard (*RFC 7636*), is used when the client, usually a mobile or a JavaScript application, requires access to protected resources.

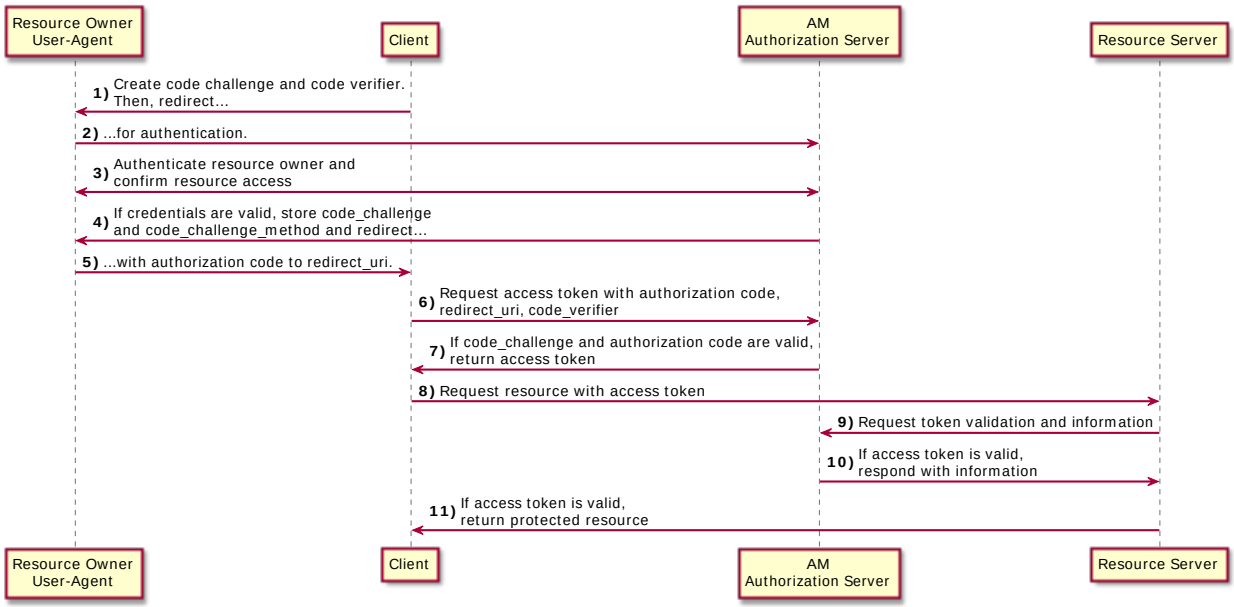
The flow is similar to the regular Authorization Code grant type, but the client must generate a code that will be part of the communication between the client and the authorization server. This code mitigates against interception attacks performed by malicious users.

Since communication between the client and the authorization server is not secure, clients are usually *public* so their secrets do not get compromised. Also, browser-based clients making OAuth 2.0 requests to different domains must implement Cross-Origin Resource Sharing (CORS) calls to access OAuth 2.0 resources in different domains.

The PKCE flow adds three parameters on top of those used for the Authorization code grant:

- **code_verifier** (form parameter). Contains a random string that correlates the authorization request to the token request.
- **code_challenge** (query parameter). Contains a string derived from the code verifier that is sent in the authorization request and that needs to be verified later with the code verifier.
- **code_challenge_method** (query parameter). Contains the method used to derive the code challenge.

OAuth 2.0 Authorization Code Grant with PKCE Flow



+ *Authorization Code Grant with PKCE Flow Explained*

The steps in the diagram are described below:

1. The client receives a request to access a protected resource. To access the resources, the client requires authorization from the resource owner. When using the PKCE standard, the client must generate a unique code and a way to verify it, and append the code to the request for the authorization code.
2. The client redirects the resource owner's user-agent to the authorization server.
3. The authorization server authenticates the resource owner, confirms resource access, and gathers consent if not previously saved.
4. If the resource owner's credentials are valid, the authorization server stores the code challenge and redirects the resource owner's user agent to the redirection URI.
5. During the redirection process, the authorization server appends an authorization code to the request to the client.
6. The client receives the authorization code and calls the authorization server's token endpoint to exchange the authorization code for an access token appending the verification code to the request.
7. The authorization server verifies the code stored in memory using the validation code. It also verifies the authorization code. If both codes are valid, the authorization server returns an access token (and a refresh token, if configured) to the client.
8. The client requests access to the protected resources from the resource server.
9. The resource server contacts the authorization server to validate the access token.
10. The authorization server validates the token and responds to the resource server.
11. If the token is valid, the resource server allows the client to access the protected resources.

Perform the steps in the following procedures to obtain an authorization code and exchange it for an access token:

- "To Generate a Code Verifier and a Code Challenge"
- "To Obtain an Authorization Code Using a Browser in the Authorization Code Grant with PKCE Flow"
- "To Obtain an Authorization Code Without Using a Browser in the Authorization Code Grant with PKCE Flow"
- "To Exchange an Authorization Code for an Access Token in the Authorization Code Grant Flow with PKCE Flow"

To Generate a Code Verifier and a Code Challenge

The client application must be able to generate a code verifier and a code challenge. For details, see the PKCE standard (*RFC 7636*). The information contained in this procedure is for example purposes only:

1. The client generates the code challenge and the code verifier. Creating the challenge using a SHA-256 algorithm is mandatory if the client supports it, as per the RFC 7636 standard.

   The following is an example of a code verifier and code challenge written in JavaScript:

   ```
   function base64URLEncode(words) {
      return CryptoJS.enc.Base64.stringify(words)
      .replace(/\+/g, '-')
      .replace(/\//g, '_')
      .replace(/=/g, '');
   }
   var verifier = base64URLEncode(CryptoJS.lib.WordArray.random(50));
   var challenge = base64URLEncode(CryptoJS.SHA256(verifier));
   ```

   This example generates values such as `ZpJiIM_G0SE9WlxzS69Cq0mQh8uyFaeEbILlW8tHs62SmEE6n7Nke0XJGx_F4OduTI4` for the code verifier and `j3wKnK2Fa_mc2tgdqa6GtUfCYjdWSA5S23JKTTtPF8Y` for the code challenge. These values will be used in subsequent procedures.

   The client is now ready to request an authorization code.

2. The client performs the steps in one of the following procedures to request an authorization code:

   - "To Obtain an Authorization Code Using a Browser in the Authorization Code Grant with PKCE Flow"
   - "To Obtain an Authorization Code Without Using a Browser in the Authorization Code Grant with PKCE Flow"

To Obtain an Authorization Code Using a Browser in the Authorization Code Grant with PKCE Flow

This procedure assumes the following configuration:

- AM is configured as an OAuth 2.0 authorization server. Ensure that:
  - The `code` plugin is configured in the Response Type Plugins field.
  - The `Authorization Code` grant type is configured in the Grant Types field.

  The Code Verifier Parameter Required drop-down (Realms > *Realm Name* > Services > OAuth2 Provider > Advanced) specifies whether AM require clients to include a code verifier in their calls.

  However, if a client makes a call to AM with the `code_challenge` parameter, AM will honor the code exchange regardless of the configuration of the Code Verifier Parameter Required drop-down.

- A public client called `myClient` is registered in AM with the following configuration:
  - **Scopes**: `write`
  - **Response Types**: `code`
  - **Grant Types**: `Authorization Code`

For more information, see "*Client Registration*".

Perform the steps in this procedure to obtain an authorization code using a browser:

1. The client redirects the resource owner's user-agent to the authorization server's authorization endpoint specifying, at least, the following query parameters:
   - **client_id**=*your_client_id*
   - **response_type**=code
   - **redirect_uri**=*your_redirect_uri*
   - **code_challenge**=*your_code_challenge*
   - **code_challenge_method**=S256

   For information about the parameters supported by the `/oauth2/authorize` endpoint, see "/oauth2/authorize".

   If the OAuth 2.0 provider is configured for a subrealm rather than the Top Level Realm, you must specify it in the endpoint. For example, if the OAuth 2.0 provider is configured for the `/alpha` realm, then use `/oauth2/realms/root/realms/alpha/authorize`.

   For example:

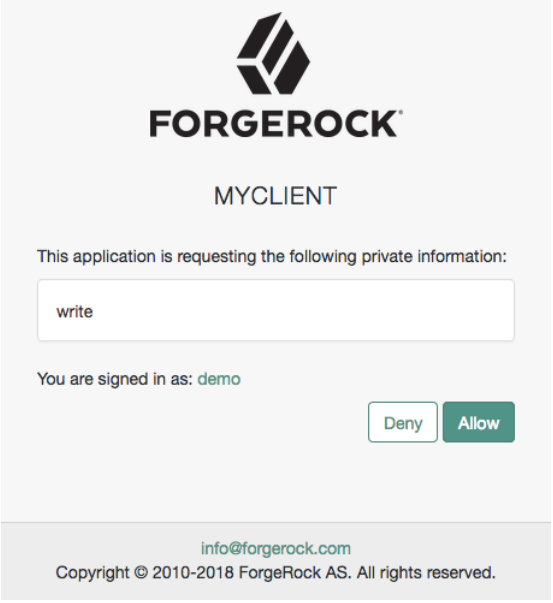
   ```
   https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/authorize \
   ?client_id=myClient \
   &response_type=code \
   &scope=write \
   &redirect_uri=https://www.example.com:443/callback \
   &code_challenge=j3wKnK2Fa_mc2tgdqa6GtUfCYjdWSA5S23JKTTtPF8Y \
   &code_challenge_method=S256 \
   &state=abc123
   ```

   Note that the URL is split and spaces have been added for readability purposes and that the `scope` and `state` parameters have been included. Scopes are not required, since they can be configured by default in the authorization server and the client, and have been added only as an example. The `state` parameter is added to protect against CSRF attacks.

2. The resource owner authenticates to the authorization server, for example, using the credentials of the `demo` user. In this case, they log in using the default chain or tree configured for the realm.

   After logging in, the authorization server presents the AM consent screen:

OAuth 2.0 Consent Screen



3. The resource owner selects the `Allow` button to grant consent for the `write` scope.

   The authorization server redirects the resource owner to the URL specified in the `redirect_uri` parameter.

4. Inspect the URL in the browser. It contains a `code` parameter with the authorization code the authorization server has issued. For example:

   ```
   http://www.example.com/?code=ZNSDo8LrsI2w-6NOCYKQgvDPqtg&scope=write&iss=https%3A%2F
   %2Fopenam.example.com%3A8443%2Fopenam%2Foauth2&state=abc123&client_id=myClient
   ```

5. The client performs the steps in "To Exchange an Authorization Code for an Access Token" to exchange the authorization code for an access token.

To Obtain an Authorization Code Without Using a Browser in the Authorization Code Grant with PKCE Flow

This procedure assumes the following configuration:

- AM is configured as an OAuth 2.0 authorization server. Ensure that:
  - The `code` plugin is configured in the Response Type Plugins field.
  - The `Authorization Code` grant type is configured in the Grant Types field.

The Code Verifier Parameter Required drop-down (Realms > *Realm Name* > Services > OAuth2 Provider > Advanced) specifies whether AM require clients to include a code verifier in their calls.

However, if a client makes a call to AM with the `code_challenge` parameter, AM will honor the code exchange regardless of the configuration of the Code Verifier Parameter Required drop-down.

- A public client called `myClient` is registered in AM with the following configuration:
  - **Scopes**: `write`
  - **Response Types**: `code`
  - **Grant Types**: `Authorization Code`

For more information, see "*Client Registration*".

Perform the steps in this procedure to obtain an authorization code:

1. The resource owner logs in to the authorization server, for example, using the credentials of the `demo` user. For example:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "X-OpenAM-Username: demo" \
--header "X-OpenAM-Password: Ch4ng31t" \
--header "Accept-API-Version: resource=2.0, protocol=1.0" \
'https://openam.example.com:8443/openam/json/realms/root/realms/alpha/authenticate'
{
    "tokenId":"AQIC5wM...TU3OQ*",
    "successUrl":"/openam/console",
    "realm":"/alpha"
}
```

2. The client makes a POST call to the authorization server's authorization endpoint specifying in a cookie SSO token of the `demo` and, at least, the following parameters:
   - **client_id**=*your_client_id*
   - **response_type**=code
   - **redirect_uri**=*your_redirect_uri*
   - **decision**=allow
   - **csrf**=*demo_user_SSO_token*
   - **code_challenge**=*your_code_challenge*
   - **code_challenge_method**=S256

For information about the parameters supported by the `/oauth2/authorize` endpoint, see "/oauth2/authorize".

If the OAuth 2.0 provider is configured for a subrealm rather than the Top Level Realm, you must specify it in the endpoint. For example, if the OAuth 2.0 provider is configured for the `/alpha` realm, then use `/oauth2/realms/root/realms/alpha/authorize`.

For example:

```
$ curl --dump-header - \
--request POST \
--Cookie "iPlanetDirectoryPro=AQIC5wM...TU3OQ*" \
--data "redirect_uri=https://www.example.com:443/callback" \
--data "scope=write" \
--data "response_type=code" \
--data "client_id=myClient" \
--data "csrf=AQIC5wM...TU3OQ*" \
--data "state=abc123" \
--data "decision=allow" \
--data "code_challenge=j3wKnK2Fa_mc2tgdqa6GtUfCYjdWSA5S23JKTTtPF8Y" \
--data "code_challenge_method=S256" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/authorize"
```

Note that the `scope` and the `state` parameters have been included. Scopes are not required, since they can be configured by default in the authorization server and the client, and have been added only as an example. The `state` parameter is added to protect against CSRF attacks.

If the authorization server is able to authenticate the user and the client, it returns an HTTP 302 response with the authorization code appended to the redirection URL:

```
HTTP/1.1 302 Found
Server: Apache-Coyote/1.1
X-Frame-Options: SAMEORIGIN
Pragma: no-cache
Cache-Control: no-store
Date: Mon, 30 Jul 2018 11:42:37 GMT
Accept-Ranges: bytes
Location: http://www.example.com?code=g5B3qZ8rWzKIU2xodV_kkSIk0F4&scope=write&iss=https%3A%2F
%2Fopenam.example.com%3A8443%2Fopenam%2Foauth2&state=abc123&client_id=myClient
Vary: Accept-Charset, Accept-Encoding, Accept-Language, Accept
Content-Length: 0
```

3. Perform the steps in "To Exchange an Authorization Code for an Access Token in the Authorization Code Grant Flow with PKCE Flow" to exchange the authorization code for an access token.

To Exchange an Authorization Code for an Access Token in the Authorization Code Grant Flow with PKCE Flow

Perform the steps in the following procedure to exchange an authorization code for an access token:

1. Ensure the client has obtained an authorization code by performing the steps in either "To Obtain an Authorization Code Using a Browser in the Authorization Code Grant with PKCE Flow" or "To Obtain an Authorization Code Without Using a Browser in the Authorization Code Grant with PKCE Flow".

2. The client creates a POST request to the token endpoint in the authorization server specifying, at least, the following parameters:

   - **grant_type**=authorization_code
   - **code**=*your_authorization_code*
   - **client_id**=*your_client_id*
   - **redirect_uri**=*your_redirect_uri*
   - **code_verifier**=*your_code_verifier*

   For information about the parameters supported by the `/oauth2/access_token` endpoint, see "/oauth2/access_token". For information about private client authentication methods, see "*OAuth 2.0 Client Authentication*".

   For example:

   ```
   $ curl --request POST \
   --data "grant_type=authorization_code" \
   --data "code=g5B3qZ8rWzKIU2xodV_kkSIk0F4" \
   --data "client_id=myClient" \
   --data "redirect_uri=https://www.example.com:443/callback" \
   --data "code_verifier=ZpJiIM_G0SE9WlxzS69Cq0mQh8uyFaeEbILlW8tHs62SmEE6n7Nke0XJGx_F4OduTI4" \
   "https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token"
   ```

   The `client_id` and the `redirection_uri` parameters specified in this call must match those used as part of the authorization code request, or the authorization server will not validate the code.

   The authorization server returns an access token in the `access_token` property. For example:

   ```
   {
     "access_token": "sbQZuveFumUDV5R1vVBl6QAGNB8",
     "scope": "write",
     "token_type": "Bearer",
     "expires_in": 3599
   }
   ```

Tip

The authorization server can also issue refresh tokens at the same time the access tokens are issued. For more information, see "*Refresh Tokens*".

Implicit Grant

Endpoints

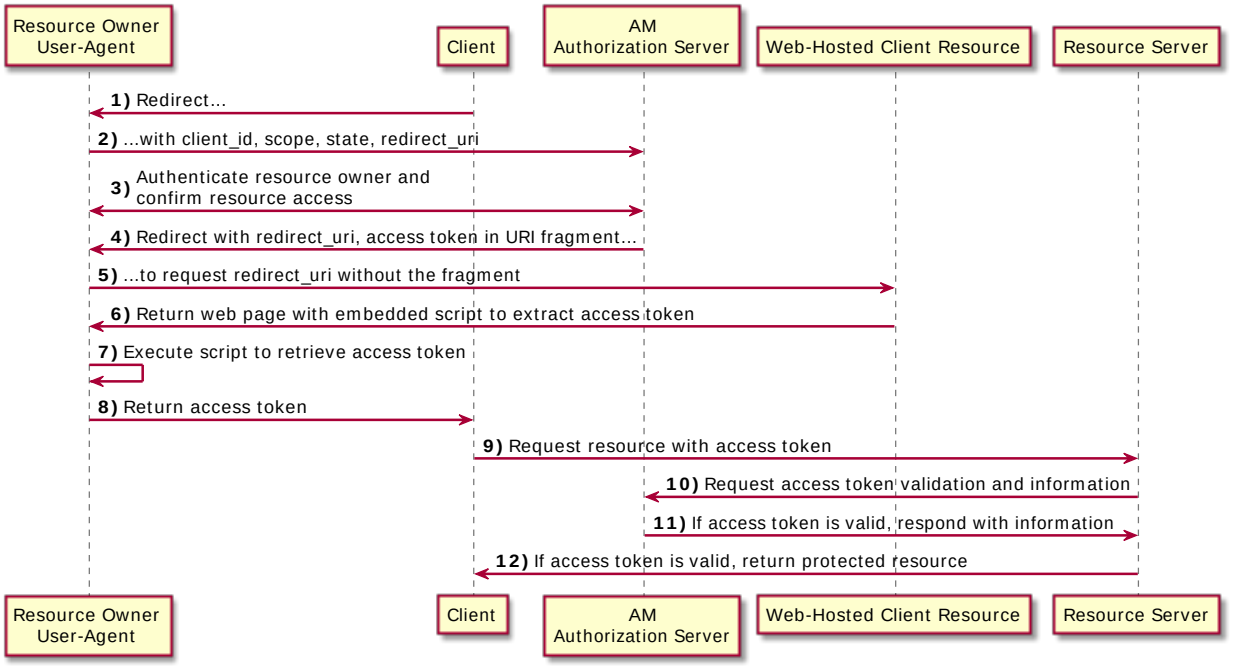
- /oauth2/authorize

The Implicit grant is designed for public clients that run inside the resource owner's user-agent, for example, JavaScript applications.

Since applications running in the user-agent are considered less trusted than applications running in servers, the authorization server will never issue refresh tokens in this flow. Also, you must consider the security impact of cross-site scripting (XSS) attacks that could leak the access token to other systems, and implement Cross-Origin Resource Sharing (CORS) to make OAuth 2.0 requests to different domains.

Due to the security implications of this flow, it is recommended to use the Authorization Code grant with PKCE flow whenever possible.

OAuth 2.0 Implicit Grant Flow



+ *Implicit Grant Flow Explained*

The steps in the diagram are described below:

1. The client, usually a single-page application (SPA), receives a request to access a protected resource. To access the resources, the client requires authorization from the resource owner.
2. The client redirects the resource owner's user-agent or opens a new frame to the AM authorization service.
3. The authorization server authenticates the resource owner, confirms resource access, and gathers consent if not previously saved.
4. If the resource owner's credentials are valid, the authorization server returns the access token to the user-agent as part of the redirection URI.
5. Now, the client must extract the access token from the URI. In this example, the user-agent follows the redirection to the web-hosted server that contains the protected resources without the access token....

6. ...And the web-hosted server returns a web page with an embedded script to extract the access token from the URI.

   In another possible scenario, the redirection URI is a dummy URI in the client, and the client already has the logic in itself to extract the access token.

7. The user-agent executes the script and retrieves the access token.
8. The user-agent returns the access token to the client.
9. The client requests access to the protected resources presenting the access token to the resource server.
10. The resource server contacts the authorization server to validate the access token.
11. The authorization server validates the token and responds to the resource server.
12. If the token is valid, the resource server allows the client to access the protected resources.

Perform the steps in the following procedures to obtain an access token:

- "To Obtain an Access Token Using a Browser in the Implicit Grant"
- "To Obtain an Access Token Without Using a Browser in the Implicit Grant"

To Obtain an Access Token Using a Browser in the Implicit Grant

This procedure assumes the following configuration:

- AM is configured as an OAuth 2.0 authorization server. Ensure that:
  - The `token` plugin is configured in the Response Type Plugins field.
  - The `Implicit Grant` grant type is configured in the Grant Types field.

  For more information, see "*Authorization Server Configuration*".
- A public client called `myClient` is registered in AM with the following configuration:
  - **Scopes**: `write`
  - **Response Types**: `token`
  - **Grant Types**: `Implicit`

For more information, see "*Client Registration*".

Perform the steps in this procedure to obtain an access token using the Implicit grant:

1. The client makes a GET call to the authorization server's authorization endpoint specifying, at least, the following parameters:

   - **client_id**=*your_client_id*
   - **response_type**=token
   - **redirect_uri**=*your_redirect_uri*

   For information about the parameters supported by the `/oauth2/authorize` endpoint, see "/oauth2/authorize".

   If the OAuth 2.0 provider is configured for a subrealm rather than the Top Level Realm, you must specify it in the endpoint. For example, if the OAuth 2.0 provider is configured for the `/alpha` realm, then use `/oauth2/realms/root/realms/alpha/authorize`.

   For example:

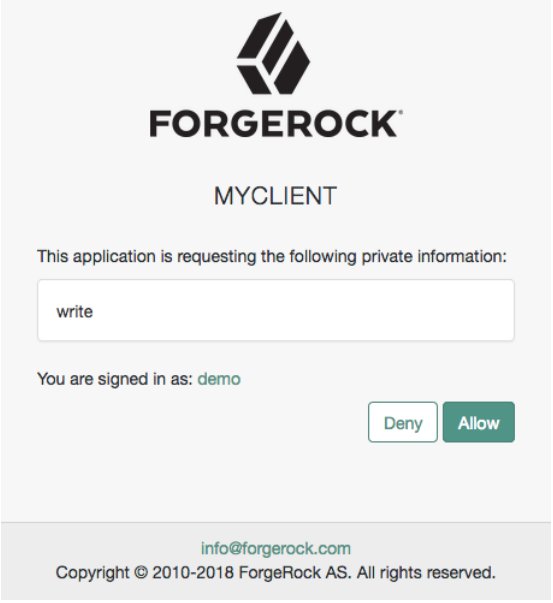
   ```
   https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/authorize \
   ?client_id=myClient \
   &response_type=token \
   &scope=write \
   &redirect_uri=https://www.example.com:443/callback \
   &state=abc123
   ```

   Note that the URL is split for readability purposes and that the `scope` and `state` parameters have been included. Scopes are not required, since they can be configured by default in the authorization server and the client, and have been added only as an example. The `state` parameter is added to protect against CSRF attacks. Also, the redirection URI was not specified, and the URI defined in the client profile is used by default.

2. The resource owner logs in to the authorization server, for example, using the credentials of the `demo` user. In this case, they log in using the default chain or tree configured for the realm.

   After logging in, the authorization server presents the AM user interface consent screen:

OAuth 2.0 Consent Screen



3. The resource owner selects the `Allow` button to grant consent for the `write` scope.

   The authorization server redirects the resource owner to the URL specified in the `redirect_uri` parameter.

4. Inspect the URL in the browser. It contains an `access_token` parameter with the access token the authorization server has issued. For example:

   ```
   https://www.example.com:443/callback#access_token=1i5IfaebiLnpyxFM4mcTSZSegb4&scope=write&redirect_uri
   %3Dhttps%3A%2F%2Fwww.example.com%3A8443%2Fcallback&iss=https%3A%2F%2Fopenam.example.com
   %3A8443%2Fopenam%2Foauth2&state=abc123&client_id=myClient
   ```

To Obtain an Access Token Without Using a Browser in the Implicit Grant

This procedure assumes the following configuration:

- AM is configured as an OAuth 2.0 authorization server. Ensure that:
  - The `token` plugin is configured in the Response Type Plugins field.
  - The `Implicit Grant` grant type is configured in the Grant Types field.

  For more information, see "*Authorization Server Configuration*".

- A public client called `myClient` is registered in AM with the following configuration:
  - **Scopes**: `write`
  - **Response Types**: `token`
  - **Grant Types**: `Implicit`

For more information, see "*Client Registration*".

Perform the steps in this procedure to obtain an access token using the Implicit grant:

1. The resource owner authenticates to the authorization server, for example, using the credentials of the `demo` user. For example:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "X-OpenAM-Username: demo" \
--header "X-OpenAM-Password: Ch4ng31t" \
--header "Accept-API-Version: resource=2.0, protocol=1.0" \
'https://openam.example.com:8443/openam/json/realms/root/realms/alpha/authenticate'
{
    "tokenId":"AQIC5wM...TU3OQ*",
    "successUrl":"/openam/console",
    "realm":"/alpha"
}
```

2. The client makes a POST call to the authorization server's authorization endpoint, specifying the SSO token of the `demo` in a cookie and, at least, the following parameters:
   - **client_id**=*your_client_id*
   - **response_type**=token
   - **decision**=allow
   - **csrf**=*demo_user_SSO_token*
   - **redirect_uri**=*your_redirect_uri*

   For information about the parameters supported by the `/oauth2/authorize` endpoint, see "/oauth2/authorize".

   If the OAuth 2.0 provider is configured for a subrealm rather than the Top Level Realm, you must specify it in the endpoint. For example, if the OAuth 2.0 provider is configured for the `/alpha` realm, then use `/oauth2/realms/root/realms/customers/alpha`.

   For example:

```
curl --dump-header - \
--Cookie "iPlanetDirectoryPro=AQIC5wM...TU3OQ*" \
--request POST \
--data "client_id=myClient" \
--data "response_type=token" \
--data "scope=write" \
--data "state=123abc" \
--data "decision=allow" \
--data "csrf=AQIC5wM...TU3OQ*" \
--data "redirect_uri=https://www.example.com:443/callback" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/authorize"
```

Note that the `scope` and `state` parameters have been included. Scopes are not required, since they can be configured by default in the authorization server and the client, and have been added only as an example. The `state` parameter is added to protect against CSRF attacks.

If the authorization server is able to authenticate the user, it returns an HTTP 302 response with the access token appended to the redirection URI:

```
1.1 302 Found
Server: Apache-Coyote/1.1
X-Frame-Options: SAMEORIGIN
Pragma: no-cache
Cache-Control: no-store
Date: Wed, 22 Aug 2018 11:19:54 GMT
Accept-Ranges: bytes
Location: https://www.example.com:443/
callback#access_token=PAsODWCvnb5W8uuBT12H62Rvmro&scope=write&redirect_uri%3Dhttps%3A%2F
%2Fwww.example.com%3A8443%2Fcallback&iss=https%3A%2F%2Fopenam.example.com%3A8443%2Fopenam
%2Foauth2&state=123abc&token_type=Bearer&expires_in=3599&client_id=myClient
Vary: Accept-Charset, Accept-Encoding, Accept-Language, Accept
Content-Length: 0
```

In this case, the redirection URI was not specified in the command, and the URI defined in the client profile is used by default.

Resource Owner Password Credentials Grant

Endpoints

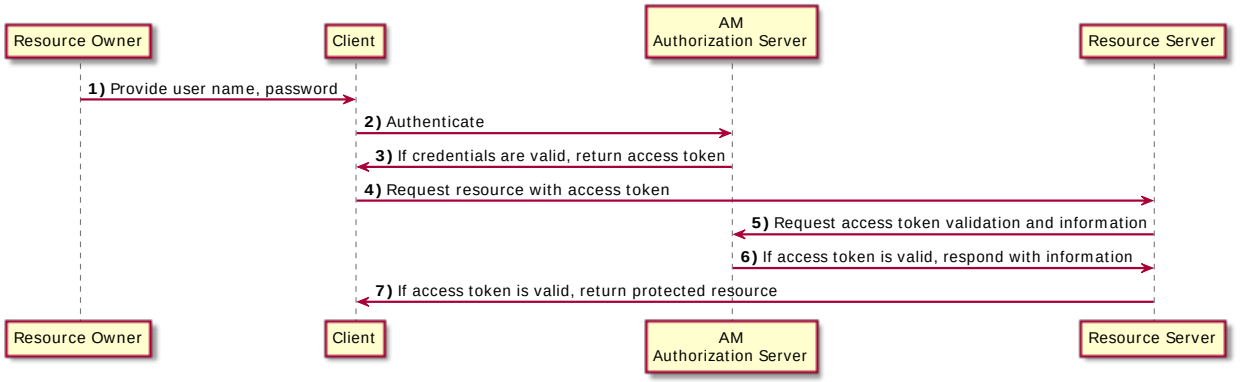
- /oauth2/access_token

The Resource Owner Password Credentials (ROPC) grant flow lets the client use the resource owner's user name and password to get an access token.

Since the resource owner shares their credentials with the client, this flow is deemed the most insecure of the OAuth 2.0 flows. The resource owner's credentials can potentially be leaked or abused by the client application, and the resource owner has no control over the authorization process.

You should implement the ROPC grant flow only if the resource owner has a trust relationship with the client (such as the device operating system, or a highly privileged application).

OAuth 2.0 Resource Owner Password Credentials Grant Flow



+ *ROPC Grant Flow Explained*

The steps in the diagram are described below:

1. The resource owner provides the client with their username and password.
2. The client sends the resource owner's and its own credentials to the authorization server, which authenticates the credentials and authorizes the resource owner's request.
3. If the credentials are valid, the authorization server returns an access token to the client.
4. The client requests access to the protected resources presenting the access token to the resource server.
5. The resource server contacts the authorization server to validate the access token.
6. The authorization server validates the token and responds to the resource server.
7. If the token is valid, the resource server allows the client to access the protected resources.

Perform the following procedure to obtain an access token:

To Obtain an Access Token Using the ROPC Grant Flow

This procedure assumes the following configuration:

- An authentication service - a chain or tree - that is able to authenticate a username and password combination, without requiring any UI-based interaction from the resource owner, is available.

  For example, the `ldapService` chain (the default), or the `Example` tree.

Specify the chain or tree by using one or more of the methods below. AM checks for the configured value in the following order, using the first value found:

1. For a specific access token REST request.

   Set the `auth_chain` parameter.

2. Individually for a realm, overriding the realm-level setting below.

   Navigate to Realms > *Realm Name* > Services > OAuth2 Provider > Advanced, and set the Password Grant Authentication Service property.

3. Individually for a realm.

   Navigate to Realms > *Realm Name* > Authentication > Settings > Core, and set the Organization Authentication Configuration property.

4. Globally, for all realms.

   Navigate to Configure > Authentication > Core Attributes > Core, and set the Organization Authentication Configuration property.

For more information, see Configure Sensible Default Authentication Services in the *Security Guide*.

- AM is configured as an OAuth 2.0 authorization server. Ensure that:
  - The `Resource Owner Password Credentials` grant type is configured in the Grant Types field.

  For more information, see "*Authorization Server Configuration*".
- A confidential client called `myClient` is registered in AM with the following configuration:
  - **Client secret**: `forgerock`
  - **Scopes**: `write`
  - **Grant Types**: `Resource Owner Password Credentials`

For more information, see "*Client Registration*".

Perform the following steps to obtain an access token using the ROPC grant flow:

1. The resource owner provides their credentials to the client. This is done outside the scope of this procedure.
2. The client creates a POST request to the authorization server's token endpoint specifying, at least, the following parameters:
   - **username**= *your_resource_owner_username*

- **password**= *your_resource_owner_password*
- **grant_type**=password

For information about the parameters supported by the `/oauth2/access_token` endpoint, see "/oauth2/access_token".

Confidential clients can authenticate to the OAuth 2.0 endpoints in several ways. This example uses the following form parameters:

- **client_id**=*your_client_id*
- **client_secret**=*your_client_secret*

For more information, see "*OAuth 2.0 Client Authentication*".

If the OAuth 2.0 provider is configured for a subrealm rather than the Top Level Realm, you must specify it in the endpoint. For example, if the OAuth 2.0 provider is configured for the `/alpha` realm, then use `/oauth2/realms/root/realms/alpha/access_token`.

For example:

```
$ curl --request POST \
--data "grant_type=password" \
--data "username=demo" \
--data "password=Ch4ng31t" \
--data "scope=write" \
--data "client_id=myClient" \
--data "client_secret=forgerock" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token"
```

Note that the `scope` parameter has been included. Scopes are not required, since they can be configured by default in the authorization server and the client, and have been added only as an example.

The authorization server returns an access token in the `access_token` property. For example:

```
{
  "access_token": "sbQZuveFumUDV5R1vVBl6QAGNB8",
  "scope": "write",
  "token_type": "Bearer",
  "expires_in": 3599
}
```

Tip

The authorization server can also issue refresh tokens at the same time the access tokens are issued. For more information, see "*Refresh Tokens*".

Client Credentials Grant

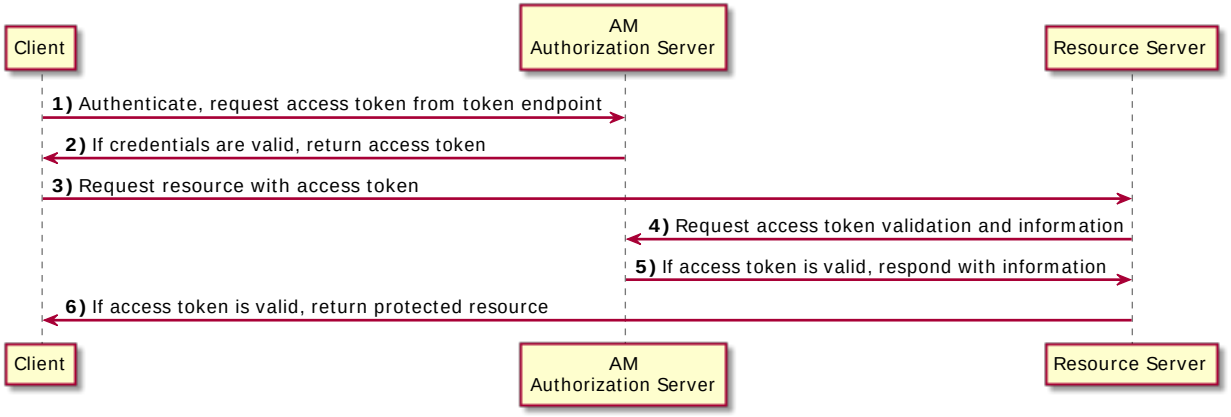
Endpoints

- /oauth2/access_token

The Client Credentials grant is used when the client is also the resource owner and it is accessing its own data instead of acting in behalf of a user. For example, an application that needs access to a protected resource to retrieve its own data to perform a task, or update its configuration, would use the Client Credentials grant to acquire an access token.

The Client Credentials Grant flow supports confidential clients only.

OAuth 2.0 Client Credentials Grant Flow



+ *Client Credentials Flow Explained*

The steps in the diagram are described below:

1. The client sends its credentials to the authorization server to get authenticated, and requests an access token.
2. If the client credentials are valid, the authorization server returns an access token to the client.

3. The client requests access to the protected resources from the resource server.
4. The resource server contacts the authorization server to validate the access token.
5. The authorization server validates the token and responds to the resource server.
6. If the token is valid, the resource server allows the client to access the protected resources.

Perform the steps in the following procedure to obtain an access token:

To Obtain an Access Token Using the Client Credentials Grant

This procedure assumes the following configuration:

- AM is configured as an OAuth 2.0 authorization server. Ensure that:
  - The `Client Credentials` grant type is configured in the Grant Types field.

  For more information, see "*Authorization Server Configuration*".
- A confidential client called `myClient` is registered in AM with the following configuration:
  - **Client secret**: `forgerock`
  - **Scopes**: `write`
  - **Grant Types**: `Client Credentials`

For more information, see "*Client Registration*".

Perform the steps in this procedure to obtain an access token using the Client Credentials grant:

- The client makes a POST call to the authorization server's token endpoint specifying, at least, the following parameters:
  - **grant_type**=client_credentials

  For information about the parameters supported by the `/oauth2/access_token` endpoint, see "/oauth2/access_token".

  Confidential clients can authenticate to the OAuth 2.0 endpoints in several ways. This example uses the following form parameters:
  - **client_id**=*your_client_id*
  - **client_secret**=*your_client_secret*

  For more information, see "*OAuth 2.0 Client Authentication*".

If the OAuth 2.0 provider is configured for a subrealm rather than the Top Level Realm, you must specify it in the endpoint. For example, if the OAuth 2.0 provider is configured for the `/alpha` realm, use `/oauth2/realms/root/realms/alpha/access_token`.

For example:

```
$ curl --request POST \
--data "grant_type=client_credentials" \
--data "client_id=myClient" \
--data "client_secret=forgerock" \
--data "scope=write" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token"
```

Note that the `scope` parameter has been included. Scopes are not required, since they can be configured by default in the authorization server and the client, and have been added only as an example.

The authorization server returns an access token in the `access_token` property. For example:

```
{
  "access_token": "sbQZuveFumUDV5R1vVBl6QAGNB8",
  "scope": "write",
  "token_type": "Bearer",
  "expires_in": 3599
}
```

Device Flow

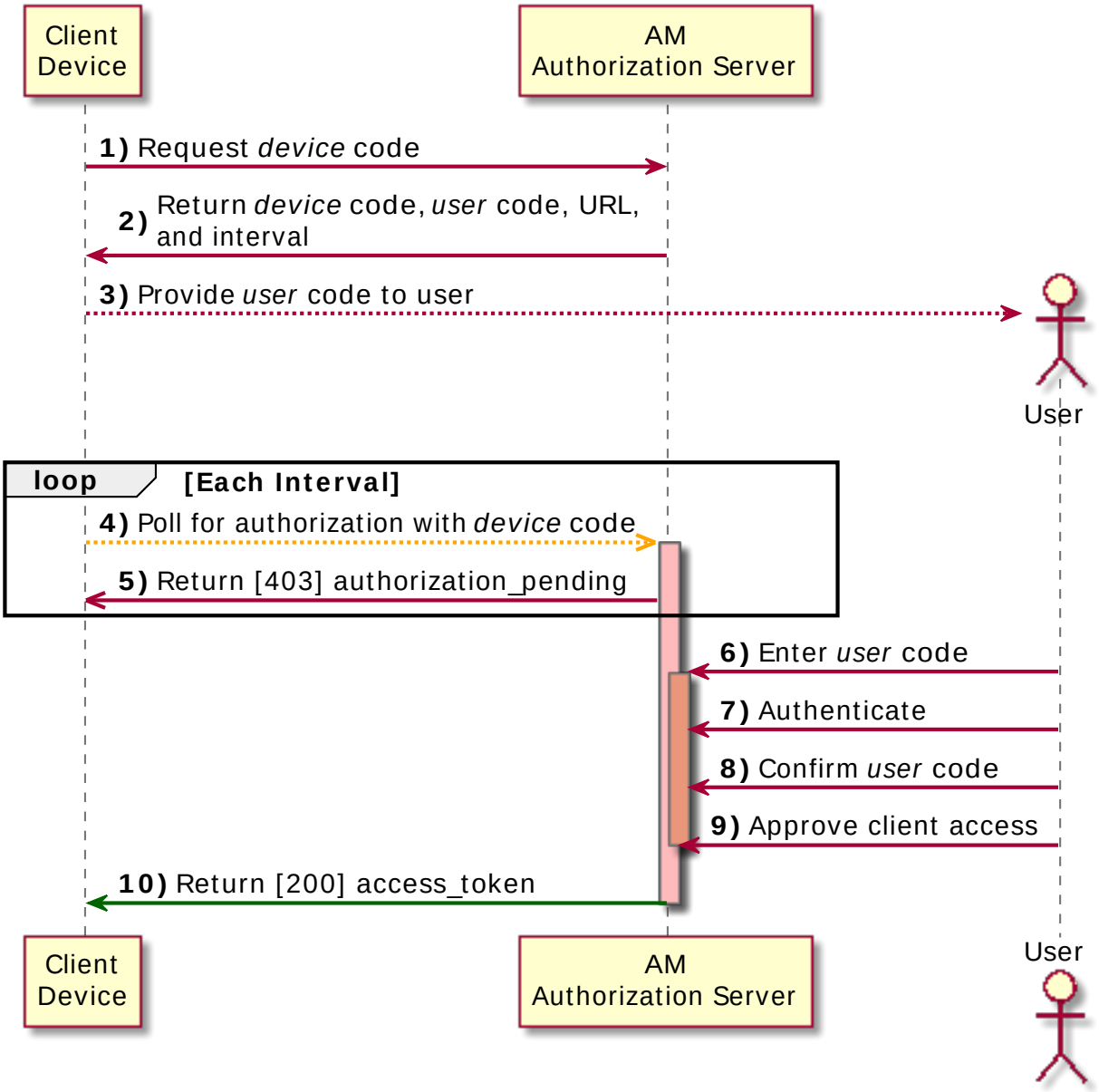
Endpoints

- /oauth2/device/code
- /oauth2/device/user
- /oauth2/access_token

The Device Flow is designed for client devices that have limited user interfaces, such as a set-top box, streaming radio, or a server process running on a headless operating system.

Rather than logging in by using the client device itself, you can authorize the client to access protected resources on your behalf by logging in with a different user agent, such as an Internet browser or smartphone, and entering a code displayed on the client device.

OAuth 2.0 Device Flow



+ *Device Flow Explained*

The steps in the diagram are described below:

1. The client device requests a device code from AM.
2. AM returns a device code, a user code, a URL for entering the user code, and an interval, in seconds.
3. The client device provides instructions to the user to enter the user code. The client may choose an appropriate method to convey the instructions, for example, text instructions on screen, or a QR code.
4. The client device begins to continuously poll AM to see if authorization has been completed.
5. If the user has not yet completed the authorization, AM returns an HTTP 403 status code, with an `authorization_pending` message.
6. The user follows the instructions from the client device to enter the user code by using a separate device.
7. If the user code is valid, AM redirects the resource owner for authentication.
8. Upon authentication, the user is prompted to confirm the user code. The page is pre-populated with the one entered before.
9. The user can authorize the client device. The AM consent page also displays the requested scopes, and their values.

   Note

   AM does not display the confirmation nor the consent pages if the user has a valid session when they entered the code, and the client is allowed to skip consent.

   This is also true if you perform the call using REST and pass the `decision=allow` parameter.

10. Upon authorization, AM responds to the client device's polling with an HTTP 200 status, and an access token, giving the client device access to the requested resources.

The following procedures show how to use the OAuth 2.0 device flow endpoints:

- "To Obtain a User Code For the Device".
- "To Grant Consent with a User Code Using a Browser in the Device Flow".

- "To Grant Consent with a User Code Without Using a Browser in the Device Flow".
- "To Poll for Authorization in the OAuth 2.0 Device Flow".

To Obtain a User Code For the Device

Devices can display a user code and instructions for a user, which can be used on a separate client to provide consent, allowing the device to access resources.

As user codes may be displayed on lower resolution devices, the list of possible characters used has been optimized to reduce ambiguity. User codes consist of a random selection of eight of the following characters:

```
234567ABCDEFGHIJKLMNOPQRSTVWXYZabcdefghijkmnopqrstvwxyz
```

This procedure assumes the following configuration:

- AM is configured as an OAuth 2.0 authorization server. Ensure that:
  - The `Device Code` grant type is configured in the Grant Types field.

  For more information, see "*Authorization Server Configuration*".
- A public client called `myClient` is registered in AM with the following configuration:
  - **Scopes**: `write`
  - **Grant Types**: `Device Code`

For more information, see "*Client Registration*".

Perform the following steps to request a user code in the OAuth 2.0 device flow:

1. The client creates a POST request to the `/oauth2/device/code` endpoint specifying, at least, the following parameters:
   - **response_type**=`device_code`
   - **client_id**=*your_client_ID*

   For information about the parameters supported by the `/oauth2/device/code` endpoint, see "/oauth2/device/code". For information about private client authentication methods, see "*OAuth 2.0 Client Authentication*".

   For example:

```
$ curl \
--request POST \
--data "response_type=device_code" \
--data "client_id=myClient" \
--data "scope=write" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/device/code"
{
    "interval": 5,
    "device_code": "7a95a0a4-6f13-42e3-ac3e-d3d159c94c55...",
    "verification_uri": "https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/
device/user",
    "verification_url": "https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/
device/user",
    "user_code": "VAL12e0v",
    "expires_in": 300
}
```

On success, AM returns a verification URI (the `verification_url` output is included to support earlier versions of the draft), and a user code to enter at that URL. AM also returns an interval, in seconds, that the client device must wait for in between requests for an access token.

Tip

You can configure the returned values by navigating to Realms > *Realm Name* > Services > OAuth2 Provider > Device Flow.

2. The client device should now provide instructions to the user to enter the user code and grant access to the OAuth 2.0 device. The client may choose an appropriate method to convey the instructions, for example, text instructions on screen, or a QR code. Perform the steps in one of the following procedures:

   - To grant access to the client using a browser, see "To Grant Consent with a User Code Using a Browser in the Device Flow".
   - To grant access to the client without using a browser, see "To Grant Consent with a User Code Without Using a Browser in the Device Flow".

3. The client device should also begin polling the authorization server for the access token using the interval and device code information obtained in the previous step. For more information, see "To Poll for Authorization in the OAuth 2.0 Device Flow".

To Grant Consent with a User Code Without Using a Browser in the Device Flow

OAuth 2.0 Device Flow requires that the user grants consent to allow the client device to access the resources. The authorization server would then provide the client with an access token.

To grant consent with a user code without using a browser, perform the following steps:

1. The resource owner logs in to the authorization server, for example, using the credentials of the `demo` user. For example:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "X-OpenAM-Username: demo" \
--header "X-OpenAM-Password: Ch4ng31t" \
--header "Accept-API-Version: resource=2.0, protocol=1.0" \
'https://openam.example.com:8443/openam/json/realms/root/realms/alpha/authenticate'
{
    "tokenId":"AQIC5wM...TU3OQ*",
    "successUrl":"/openam/console",
    "realm":"/alpha"
}
```

2. The client makes a POST call to the authorization server's authorization device user endpoint specifying in a cookie SSO token of the `demo` and, at least, the following parameters:

   - **user_code**=*resource_owner_user_code*
   - **decision**=`allow`
   - **csrf**=*demo_user_SSO_token*

   For information about the parameters supported by the `/oauth2/device/user` endpoint, see "/oauth2/device/user".

   The `iPlanetDirectoryPro` cookie is required and should contain the SSO token of the user granting access to the client. For example:

   ```
   $ curl \
   --request POST \
   --header "Cookie: iPlanetDirectoryPro=AQIC5wM...TU3OQ*" \
   --data "user_code=VAL12e0v" \
   --data "decision=allow" \
   --data "csrf=AQIC5wM...TU3OQ*" \
   "https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/device/user"
   ```

   The `scope` and the `client_id` parameters have not been included because the user code already contains that information.

   AM returns HTML containing a JavaScript fragment named `pageData`, with details of the result.

   Successfully allowing or denying access returns:

   ```
   pageData = {
       locale: "en_US",
       baseUrl : "https://openam.example.com:8443/openam/XUI/",
       realm : "/alpha",
       done: true
   }
   ```

   `done: true` means that the flow can now continue.

   If the supplied user code has already been used, or is incorrect, AM returns the following:

```
pageData = {
    locale: "en_US",
    errorCode: "not_found",
    realm : "/alpha",
    baseUrl : "https://openam.example.com:8443/openam/XUI/"
    oauth2Data: {
          csrf: "ErFIk8pMraJ1rvKbloTgpp6b7GZ57kyk9HaIiKMVK3g=",
          userCode: "VAL12e0v",
    }
}
```

Important

As per Section 4.1.1 of the OAuth 2.0 authorization framework, it is required that the authorization server legitimately obtains an authorization decision from the resource owner.

Any client using the endpoints to register consent is responsible for ensuring this requirement, AM cannot assert that consent was given in these cases.

To Grant Consent with a User Code Using a Browser in the Device Flow

OAuth 2.0 Device Flow requires that the user grants consent to allow the client device to access the resources. The authorization server would then provide the client with an access token.

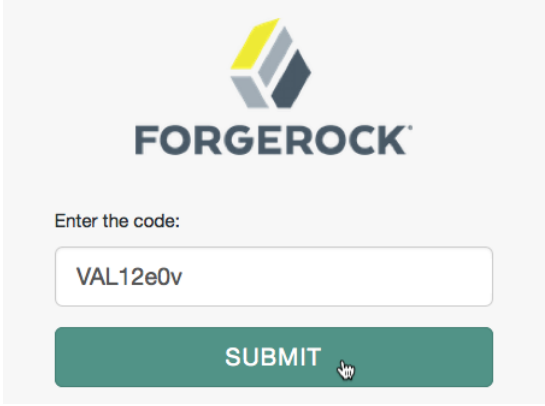
To grant consent with a user code using a browser, perform the following steps:

1. The resource owner navigates to the verification URL acquired with the user code, for example, `https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/device/user`.

2. The resource owner logs in to the authorization server using, for example, the `demo` user credentials.

3. The resource owner enters their user code:

OAuth 2.0 User Code



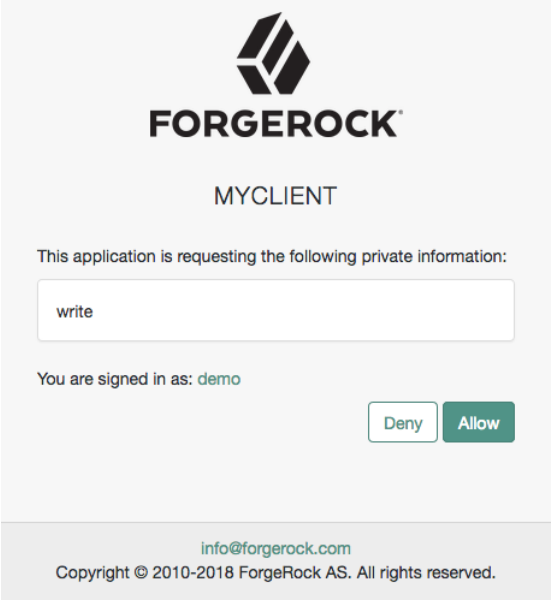
Note

If the user is not logged in to AM when they provide the code, AM redirects them to the login page.

After authenticating successfully, the user is prompted to enter the code again. The user code is pre-populated with the code they entered before.

4. The resource owner authorizes the device flow client by allowing the requested scopes:

OAuth 2.0 Consent Page



Note

If the client is allowed to skip consent, the user will not see this screen.

5. AM adds the OAuth 2.0 client to the user's profile page in the *Authorized Apps* section and displays that the user is done with the flow:

OAuth 2.0 Done Page

The device now can request an access token from AM.

To Poll for Authorization in the OAuth 2.0 Device Flow

The client device must poll the authorization server for an access token, since it cannot know whether the resource owner has already given consent or not.

Perform the following steps to poll for an access token:

- On the client device, create a POST request to poll the `/oauth2/access_token` endpoint to request an access token specifying, at least, the following parameters:

  - **client_id**=*your_client_id*
  - **grant_type**=`urn:ietf:params:oauth:grant-type:device_code`
  - **device_code**=*your_device_code*

  For information about the parameters supported by the `/oauth2/access_token` endpoint, see "/oauth2/access_token".

  The client device must wait for the number of seconds previously provided as the value of `interval` between polling AM for an access token. For example:

```
$ curl \
--request POST \
--data "client_id=myClient" \
--data "grant_type=urn:ietf:params:oauth:grant-type:device_code" \
--data "device_code=7a95a0a4-6f13-42e3-ac3e-d3d159c94c55..." \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token"
```

If the user has authorized the client device, an HTTP 200 status code is returned, with an access token that can be used to request resources:

```
{
    "expires_in": 3599,
    "token_type": "Bearer",
    "access_token": "c1e9c8a4-6a6c-45b2-919c-335f2cec5a40"
}
```

If the user has not yet authorized the client device, an HTTP 403 status code is returned, with the following error message:

```
{
    "error": "authorization_pending",
    "error_description": "The user has not yet completed authorization"
}
```

If the client device is polling faster than the specified interval, an HTTP 400 status code is returned, with the following error message:

```
{
    "error": "slow_down",
    "error_description": "The polling interval has not elapsed since the last request"
}
```

Tip

The authorization server can also issue refresh tokens at the same time the access tokens are issued. For more information, see "*Refresh Tokens*".

SAML v2.0 Profile for Authorization Grant

Endpoints

- /oauth2/access_token

The SAML v2.0 Profile for Authorization Grant is designed for environments that want to leverage the REST-based services provided by AM's OAuth 2.0 support, while keeping their existing SAML v2.0 federation implementation.

Note

The *RFC 7522* describes the means to use SAML v2.0 bearer assertions to request access tokens and to authenticate OAuth 2.0 clients.

At present, AM implements the profile to request access tokens.

Consider the following requirements before implementing this flow:

- The client (the application the resource owner uses to start the flow) must inform the resource owner that, by authenticating to the SAML v2.0 identity provider, the resource owner grants the client access to the protected resources. AM does not present the resource owner with consent pages.

  This client must be able to consume the access token and handle errors as required.

- The OAuth 2.0 authorization service and SAML v2.0 service provider must be configured in the same AM instance.

- The service provider must require that assertions are signed.

- The SAML v2.0 identity provider must issue signed assertions.

  The assertion must contain the SAML v2.0 entity names, as follows:

  - The issuer must be set to the identity provider's name. For example, `https://idp.example.com:8443/idp`.

  - The audience must be set to the service provider's name. For example, `https://openam.example.com:8443/openam`.

- The identity provider and the service provider must belong to the same circle of trust.

- AM must be able to determine the resource owner from the name ID contained in the assertion. Failure to determine the resource owner results in an error similar to:

  ```
  {"error_description":"AM identity should not be null","error":"server_error"}
  ```

  AM may fail to determine the resource owner if the assertion contains an opaque name ID during transient federation. Because the opaque reference is never stored during a transient flow, the OAuth 2.0 provider cannot determine the resource owner it relates to.

  To work around this, configure an identity in the Transient User field of the SAML v2.0 service provider. This will map all transient ID references to that identity.

- The OAuth 2.0 client is registered, at least, with the following configuration:

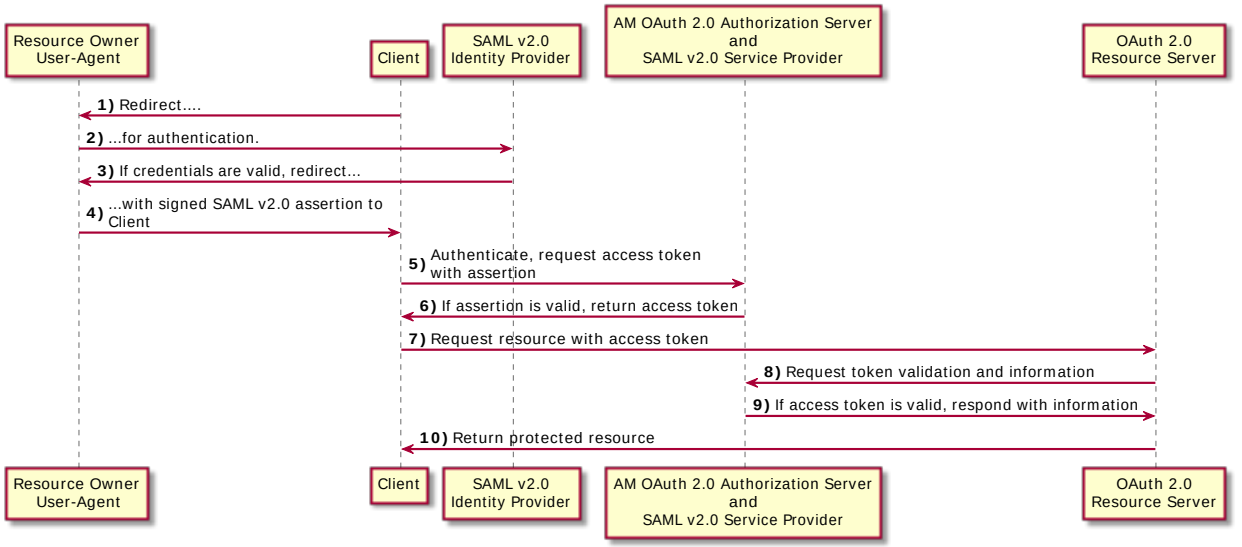
  - **Grant Types**: `SAML2`

- The OAuth 2.0 provider is configured. Ensure that:

  - The `SAML2` grant type is configured in the Grant Types field.

The following diagram demonstrates the SAML v2.0 Profile for Authorization Grants:

SAML v2.0 Profile for Authorization Grant Flow



The steps in the diagram are described below:

1. The client requests the SAML v2.0 identity provider the SAML v2.0 assertion related to the resource owner. Usually, this means the client redirects the resource owner to the identity provider for authentication.
2. The SAML v2.0 identity provider returns the signed assertion to the client.
3. The client includes the assertion and a special grant type in the call to the OAuth 2.0 token endpoint in the following parameters:
   - **grant_type**=urn:ietf:params:oauth:grant-type:saml2-bearer
   - **assertion**=*my_assertion*

     Note that the assertion must be first base64-encoded, and then URL encoded.

   For example:

```
$ curl \
--request POST \
--data "client_id=myClient" \
--data "client_secret=forgerock" \
--data-urlencode "assertion=PHNhbWxwOl...ZT4" \
--data "grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Asaml2-bearer" \
--data "redirect_uri=https://www.example.com:443/callback" \
--data "scope=write" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token"
```

4. The AM authorization server validates the assertion. If the assertion is valid, the authorization server returns an access token to the client.
5. The client request access to the protected resources from the resource server.
6. The resource server contacts the authorization server to validate the access token.
7. The authorization server validates the token and responds to the resource server.
8. If the token is valid, the resource server allows the client to access the protected resources.

JWT Profile for OAuth 2.0 Authorization Grant

Endpoints

- /oauth2/access_token

The JWT Profile for OAuth 2.0 Authorization Grant is designed for environments that want to leverage the REST-based services provided by AM's OAuth 2.0 framework while keeping their existing authentication services, as long as the trust relationship can be expressed with a JWT bearer token.

Since the trust relationship is already established, this flow does not require the end user's interaction.

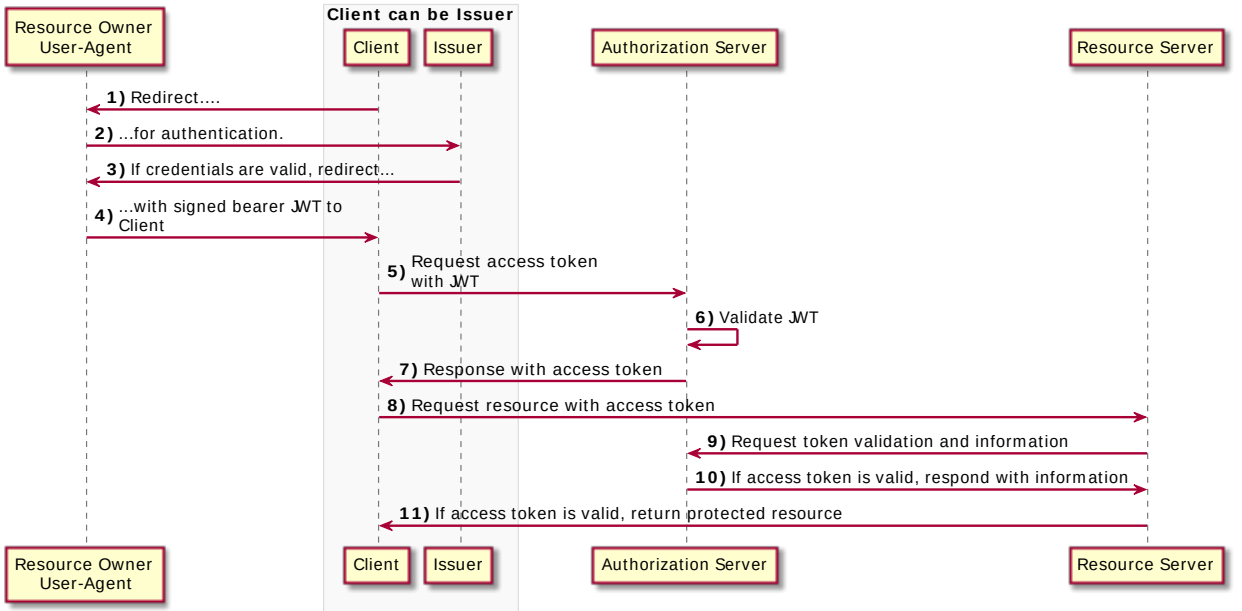
Note

The *RFC 7523* defines the use of JWT bearer tokens for both requesting access tokens as well as for client authentication.

Read this section for information about requesting access tokens. To use JWTs for client authentication, see "Authenticating Clients Using JWT Profiles".

As the authorization server, AM must validate the bearer JWT to issue the access token to the client. To ensure that malicious clients cannot self-sign their own JWTs to acquire tokens, AM requires the token issuer to be pre-registered in AM as an special type of agent.

JWT Bearer Profile for Authorization Grant



1. The client requests a JWT from the issuer. The client itself can be the issuer, in which case it will create a JWT for itself before starting the OAuth 2.0 flow.

   Regardless of who signs the JWT, the issuer must be pre-registered in AM as a trusted JWT issuer. For more information, see "To Configure a Trusted JWT Issuer Agent".

2. The issuer returns a signed JWT to the client; JWTs with message authentication codes (MACs) applied to them are not supported.

   The JWT must contain, at least, the following claims in the payload:

   - **aud**. Specifies a string or an array of strings that is the intended audience of the JWT. Must be set to, or contain, the authorization server's token endpoint.
   - **exp**. Specifies the expiration time of the JWT in Unix time.

     Providing a JWT with an expiry time greater than 30 minutes causes AM to return a `JWT expiration time is unreasonable` error message.
   - **iss**. Specifies the unique identifier of the JWT issuer. This could be the client or a third party.

     The identifier must match the issuer field configured in the trusted JWT issuer agent.

- **sub**. Specifies the principal who is the subject of the JWT. It must be a string that identifies the resource owner.

  Tip

  You can configure the trusted JWT issuer agent to check a different claim for the principal. For example, the `preferred_userame` from an ID token.

  In this case, the JWT would contain both the `sub` and the `preferred_username` claims.

  For more information, see "To Configure a Trusted JWT Issuer Agent".

The following is an example of the payload of a basic JWT:

```
{
  "aud": [
    "https://openam.example.com:8443/openam/oauth2/realms/root/access_token
  ],
  "iss": "https://www.example.com/issuer",
  "exp": 1555530663,
  "sub": "demo"
}
```

For an example of a JWT containing different claims as supported by the trusted JWT issuer agent, see "To Configure a Trusted JWT Issuer Agent".

For more information about JWTs, see the RFC 7523 standard.

3. The client includes the JWT and a client assertion type in the call to the OAuth 2.0 endpoint in the following parameters:

   - `grant_type`=`urn:ietf:params:oauth:grant-type:jwt-bearer`
   - `assertion`=*my_JWT*

   For example:

```
$ curl \
--request POST \
--data "client_id=myClient" \
--data "client_secret=forgerock" \
--data "grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Ajwt-bearer" \
--data "assertion=eyAiYWxnIjogIlJTMjU2IiB9.eyAic3ViIjogImp3..."
--data "redirect_uri=http://www.example.com" \
--data "scope=write" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token"
```

For information about the parameters supported by the `/oauth2/access_token` endpoint, see "/oauth2/access_token".

For more information about client authentication methods, see "*OAuth 2.0 Client Authentication*".

4. AM validates the JWT following the guidance specified in section 3 of the RFC7523 and also performs the following additional checks:

   a. Decodes the payload and compares the value of the `iss` claim with the value of the JWT Issuer field in the list of trusted JWT issuer agents.

   b. Validates the JWT signature either with the keys exposed on the trusted issue agent's JWK URI, or with the keys configured in the JWK Set field of the agent.

   If AM cannot validate the JWT it will return an error, such as `JWT signature is invalid`.

5. The authorization server issues an access token to the client.
6. The client requests access to the protected resources from the resource server.
7. The resource server contacts the authorization server to validate the access token.
8. The authorization server validates the token and responds to the resource server.
9. If the token is valid, the resource server allows the client to access the protected resources.

The following procedure demonstrates how to configure a trusted JWT issuer agent:

To Configure a Trusted JWT Issuer Agent

Perform the steps in this procedure to configure a trusted JWT issuer agent:

1. In the AM console, go to Realms > *Realm Name* > Applications > OAuth 2.0 > Trusted JWT Issuer.
2. Add a new trusted JWT issuer agent.
3. Complete the following fields to create the agent:

   a. In the Agent ID field, give the trusted JWT issuer agent a name. For example, `myJWTAgent`.

   b. In the JWT Issuer field, provide the URI of the JWT issuer. This URI must match the value of the issuer (the `iss` claim) in the JWTs.

   c. Select Create.

      You are presented with a screen with additional information regarding the agent.

4. Review the trusted JWT issuer agent information. You must, configure *either* the JWKs URI or the JWK Set fields, as follows:

   - **JWKs URI**: specifies a URI in the JWT issuer that exposes the verification keys AM will use to validate the JWT signature. For example, `http://www.example.com/issuer/jwk_uri`.

     If you configure this field, ensure the following properties are configured with sensible values for your environment:

- JWKs URI content cache timeout in ms
- JWKs URI content cache miss cache time

- **JWK set**: Specifies a JWK set containing the verification keys to validate the JWT signature. The following is an example of an elliptic curve JWK set:

```
{
  "keys": [{
     "kty": "EC",
     "crv": "P-256",
     "x": "i-rdOmi5lC3pn3y5sTgYiLLFVFY7XxDLinWneHEaAXA",
     "y": "mxmqqauiq44INgyyPP2vATt3IkDL_6W5CAcfAMSZl8k",
     "kid": "signing_key",
     "x5c": [
         "MIIBSjCB76ADAgEC.....955PByPrflZkQOC/g==" ]
  }]
}
```

For more information about the contents of the JWK set, see the *JSON Web Key (JWK)* specification.

You can store more than one key in the JWK set. However, it is easier to implement key rotation exposing the validation keys on the URI instead.

5. (Optional) Configure the following values to suit your environment:

- **Consented Scopes Claim**. The name of a JWT claim that indicates which scopes the resource owner consented to. The claim in the JWT can contain either a JSON array or a space-separated whitelist of scopes that the resource owner has consented to.

  For example, if you configure the `scp` claim name in this field and the JWT contains the claim `"scp":"read"`, but you request both the `read` and `write` scopes, AM will only grant the `read` scope.

  Leave this field blank to allow any scope.

  The following are example JWTs containing a claim that specifies scopes:

```
{
   "aud": [
       "https://openam.example.com:8443/openam/oauth2/realms/root/access_token
   ],
   "iss": "https://www.example.com/issuer",
   "exp": 1555530663,
   "sub": "demo",
   "scope": ["read", "write"]
}
```

  In this case, the `scope` claim is a JSON array of scopes.

```
{
   "aud": [
       "https://openam.example.com:8443/openam/oauth2/realms/root/access_token
   ],
   "iss": "https://www.example.com/issuer",
   "exp": 1555530663,
   "sub": "demo",
   "scp": "read write"
}
```

In this case, the `scp` claim is a space-separated list of scopes.

- **Resource Owner Identity Claim**. Claim in the JWT that identifies the resource owner in AM. By default, the `sub` claim.

  Note that even if you configure the trusted JWT issuer agent to verify a different claim, such as the `preferred_username` claim, the `sub` claim must still exist in the JWT.

- **Allowed Subjects**. List of subjects this JWT issuer is allowed to provide consent for.

  For example, if you configure the `demo` user in this field but the JWT subject value is `demo2`, AM will not grant the access token.

  Leave it blank to provide consent to any user.

6. Save your changes.

   The trusted JWT issuer agent is ready for use.

Chapter 13

OAuth 2.0 Token Exchange

Working as an OAuth 2.0/OpenID Connect authorization server, and following OAuth 2.0 Token Exchange specification, AM can exchange:

- Access tokens for access tokens
- Access tokens for ID tokens
- ID tokens for ID tokens
- ID tokens for access tokens

If the OAuth2 Provider service is configured to issue refresh tokens, it will also issue them after a token exchange when appropriate. However, AM does not allow the exchange of tokens for refresh tokens as part of the token exchange flows. Exchanging tokens for SAML assertions is also not supported.

Note

Clients can only exchange tokens at the OAuth 2.0 provider that issued them. For example, if a token was issued by `https://openam.example.com:8443/openam/oauth2/`, it cannot be exchanged at `https://my.example.com:8443/openam/oauth2/`.

The same restriction applies across realms in the same AM deployment, although this is in line with other AM features to ensure that realms are separate entities.

+ *OAuth 2.0 Token Exchange or the Security Token Service (STS)?*

Even though the goal of both implementations is transforming tokens, they are completely different in implementation and capabilities:

The STS service lets AM establish cross-domain trust federation relationships. To do this, AM provides a REST STS framework loosely based on the SOAP WS-Trust specification that you need to build, deploy, and maintain yourself.

The REST STS service supports username/password, SSO tokens, X.509 certificates, and ID tokens as input tokens, and SAML v2.0 assertions and ID tokens as output tokens.

Due to its transformation capabilities, the STS service is more suitable to helping federate legacy platforms.

The OAuth 2.0 Token Exchange specification makes use of AM as an OAuth 2.0/OpenID Connect authorization server to transform OAuth 2.0-related tokens.

Use the OAuth 2.0 Token Exchange in OAuth 2.0/OpenID Connect platforms.

How Does Token Exchange Work?

Exchanging tokens requires, at least, a *subject token*, which is the original token to be exchanged. The subject token can be obtained using any of the OAuth 2.0/OpenID Connect flows that AM supports.

An *actor token* may also be included in certain token exchange requests. It represents an identity that can act on behalf of another identity.

The new token resulting from token exchange is referred as exchanged token in this documentation.

The cornerstone of token exchange is the `may_act` claim. The claim, as explained in RFC 8693, is a JSON object that specifies the party or parties allowed to act on behalf of the subject of the token. Only the client or the client/subject combination specified in the `may_act` claim can exchange that token for another.

For more information and implementation details, see About the may_act Claim.

Token exchange operations are logged under the `AM-TOKEN-EXCHANGE` audit event, and new tokens can be tracked using the value of their `auditTrackingId` claim.

AM performs token exchange as if it were issuing a normal token of that type. The only differences are:

- It copies, from the subject token, the claims and values that must stay the same in the new token. For example, claims related to the subject or to the issuer of the token.

  Claims that are not relevant because the new token is of a different type are not copied over. In the same way, claims in the new token that cannot be inferred from the subject token because they are of a different type, are not added.

  Scopes are not copied from the subject token. AM derives them from the regular scope implementation that is used in the OAuth 2.0/OpenID Connect flows (see "*About Scopes*").

- It adds the `may_act` and the `act` claims when relevant.

For examples of new exchanged tokens, see "Token Exchange Flows".

Exchanged tokens do not expire at the same time as the subject tokens used to create them. Rather, they expire after the amount of time specified in the Access Token Lifetime (seconds) or the OpenID Connect JWT Token Lifetime (seconds) fields of the OAuth2 Provider service, has passed.

Restricted and Expanded Tokens

Exchanged tokens do not need to have the exact same scopes/claims that the subject token does. Tokens can have their scopes/claims restricted or expanded:

Exchanged tokens are *restricted* when their scopes/claims are a subsection of those available to the subject token. The idea behind this is that, rather than gathering consent for different sets of scopes/claims, clients gather consent for a broad range initially, and then exchange the broad-scoped token for as many restricted tokens as required.

Exchanged tokens are *expanded* when they are granted scopes/claims that are not included in the subject token. Expanding tokens works best when exchanging between ID and access tokens, due to the differences between scopes and claims.

There is no interaction with the user during token exchange. Therefore, it is impossible for AM to request consent about the expanded scopes/claims. It is the responsibility of the client application to ensure that either the user has consented beforehand, or that the expanded scope/claim is unrelated to the user's resources.

For examples of restricted and expanded tokens, see "Token Exchange Use Cases".

Token Exchange Use Cases

Clients may want to exchange tokens for *impersonation* and *delegation* purposes:

Impersonation

To impersonate means to pretend you are another person when performing a job or duty. OAuth 2.0/OpenID Connect impersonation is a token exchange use case whereby a client performs an action on behalf of the user on an environment where there is no need to keep a separation between the user and the client.

For example, a user uploading a picture to social media using an app, where it is not important for the downstream social media servers to differentiate the user from the client to resize and store the picture.

As a high-level explanation, the client in possession of an access or ID token on behalf of the user (the subject token) obtains a new access token or ID token that is identical to the original in terms of its privileges (unless the exchanged token's scopes/claims have been expanded or restricted).

Impersonated tokens cannot be told apart from regular tokens, since they do not have any specific claim or value that indicates they are the result of an impersonation flow. This is different for delegated tokens, as you will see later in this section.

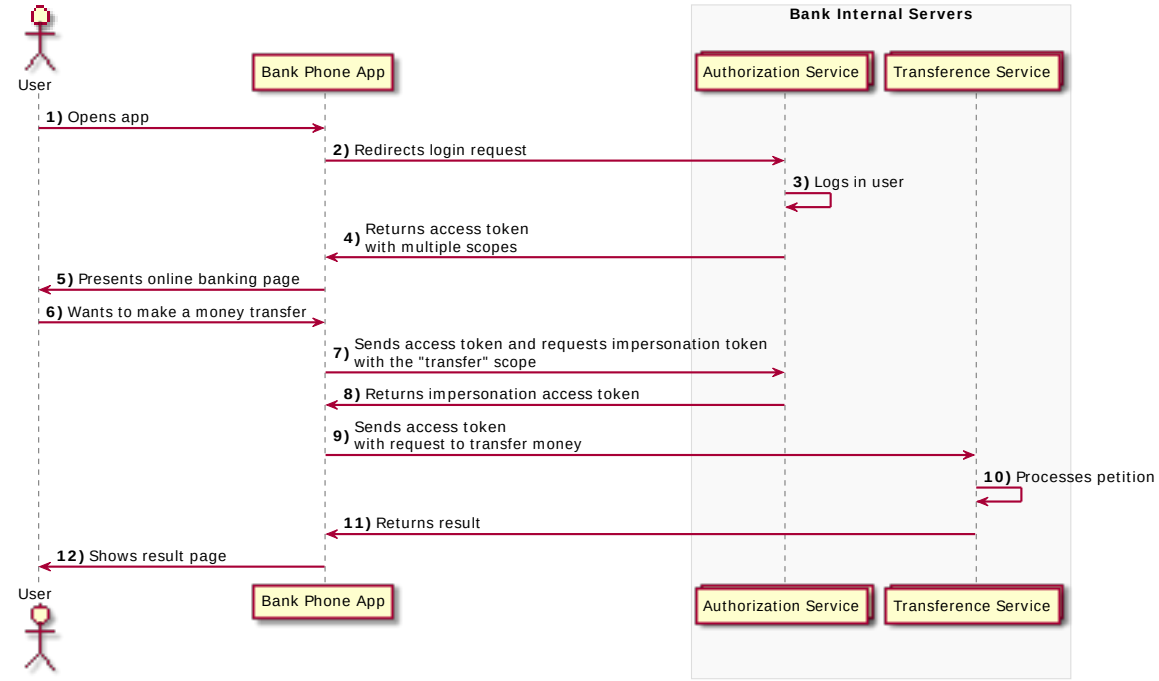
Due to the risk of identity theft, allow token impersonation across trusted systems only.

+ *Impersonation Example Use Cases*

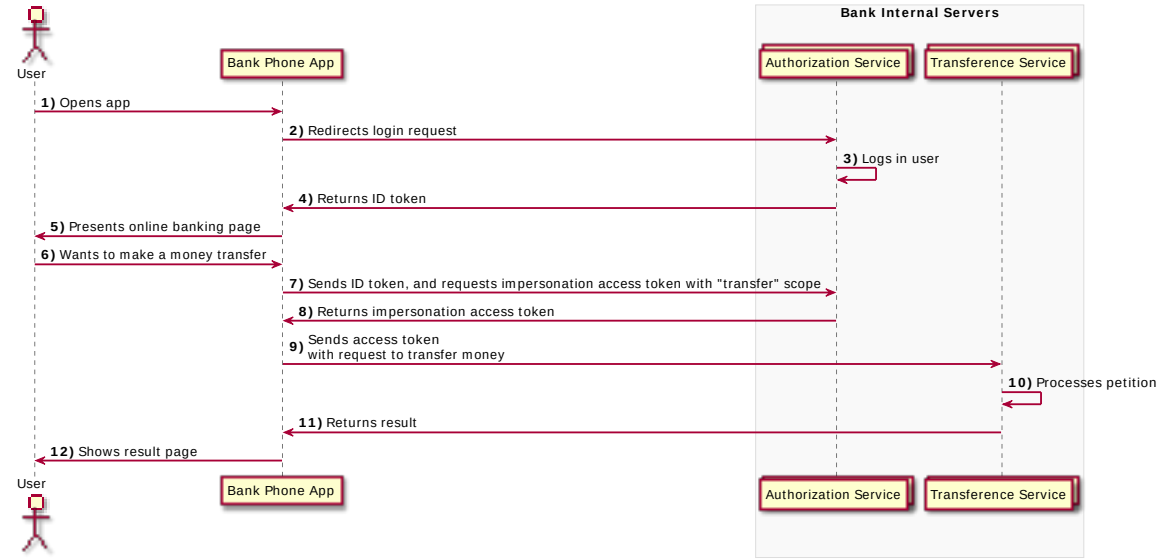
Consider a user who wants to do a money transfer using their bank application. The user logs in to the application, and trusts the application to act on their behalf to reach out to the internal banking systems to perform the transaction.

When the user logs in, they receive the subject token, which is an access token with the `read_accounts`, `create_accounts`,`transfer`, and `change_data` scopes. These scopes represent every banking service they can use in the app.

If the user only wants to make a money transfer, there is no need for the app to send all scopes to the transaction service. To reduce the security risks of impersonation, the app transforms the broad-scope access token into another access token that is restricted to the `transfer` scope:



Now, consider what happens if the example used ID tokens instead of access tokens. After logging in, the user receives the subject token, which is an ID token as a proof of their login and identity. When the user wants to make a transaction, the client exchanges the user's ID token with an access token with the `transfer` scope, and sends it to the transfer service:



In this case, the impersonation token is an expanded token since it has the `transfer` scope, which the ID token did not have.

Delegation

To delegate means giving a job or duty to someone else to perform it on your behalf. OAuth 2.0/ OpenID Connect delegation is a use case of token exchange whereby a client performs an action on behalf of the user on an environment where keeping the user and client as different entities is important.

As a high-level explanation, the client is in possession of two tokens: one on behalf of the user (the subject token), and another for itself or someone else (the actor token). Then, using both tokens, the client obtains a new token that is identical to the original in terms of its privileges (unless the exchanged token's scopes/claims have been expanded or restricted).

However, delegated tokens contain the `act` claim, which lets the resource server know that the client using the token is not the user. The implementation of the resource server could be adapted, therefore, to perform different actions when presented with a delegated token.

About the act Claim

The `act` claim, as explained in RFC 8693, is a JSON object that expresses that delegation has occurred. It also identifies the party that acts on behalf of the subject of the token. For example:

```
"act": {
  "sub": "(usr!demo)"
}
```

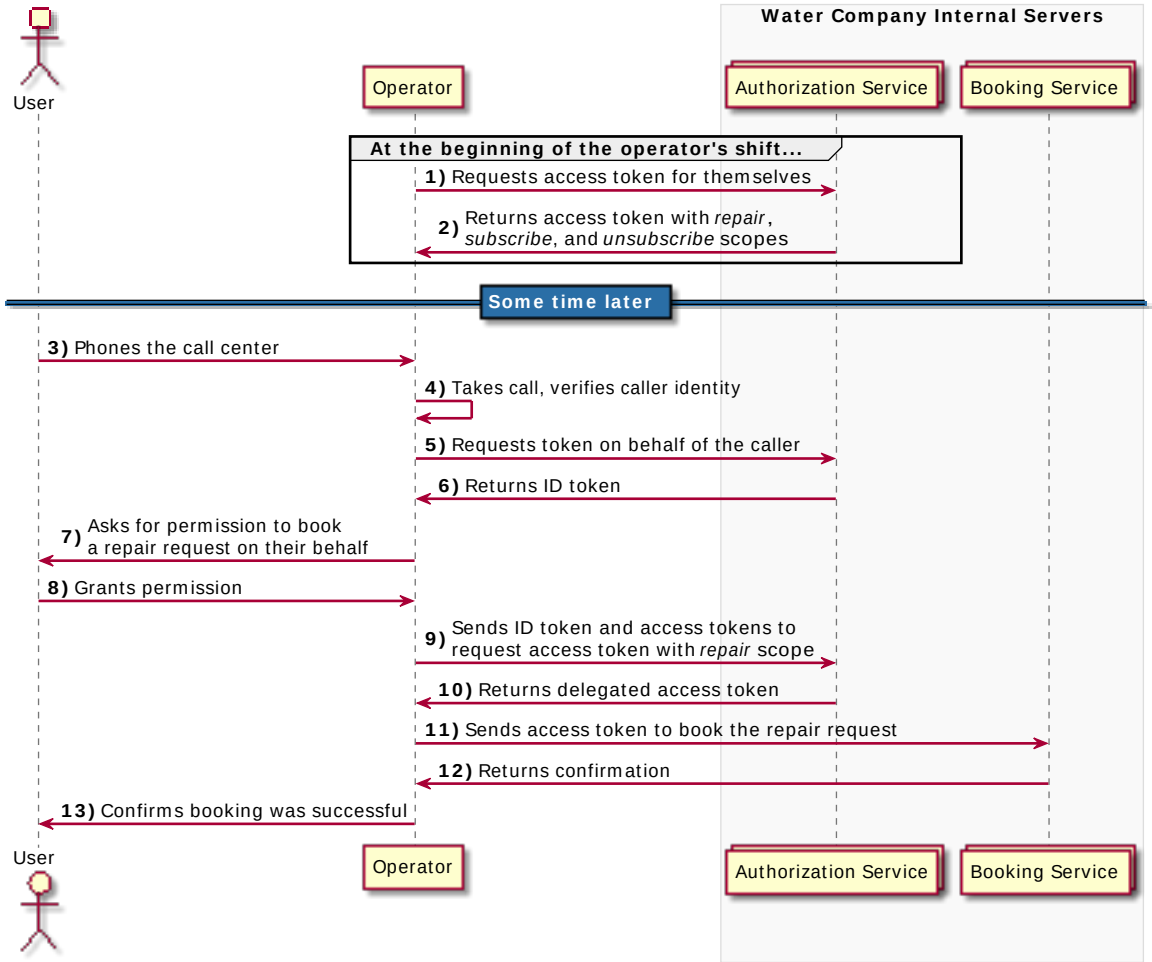
The `sub` object of the `act` claim is always the subject of the actor token used during delegation.

This approach is more secure if the token must travel through third-party systems, because even if the token was stolen, the implementation of the resource server should not give it full privileges over the user's resources.

+ *Delegation Example Use Cases*

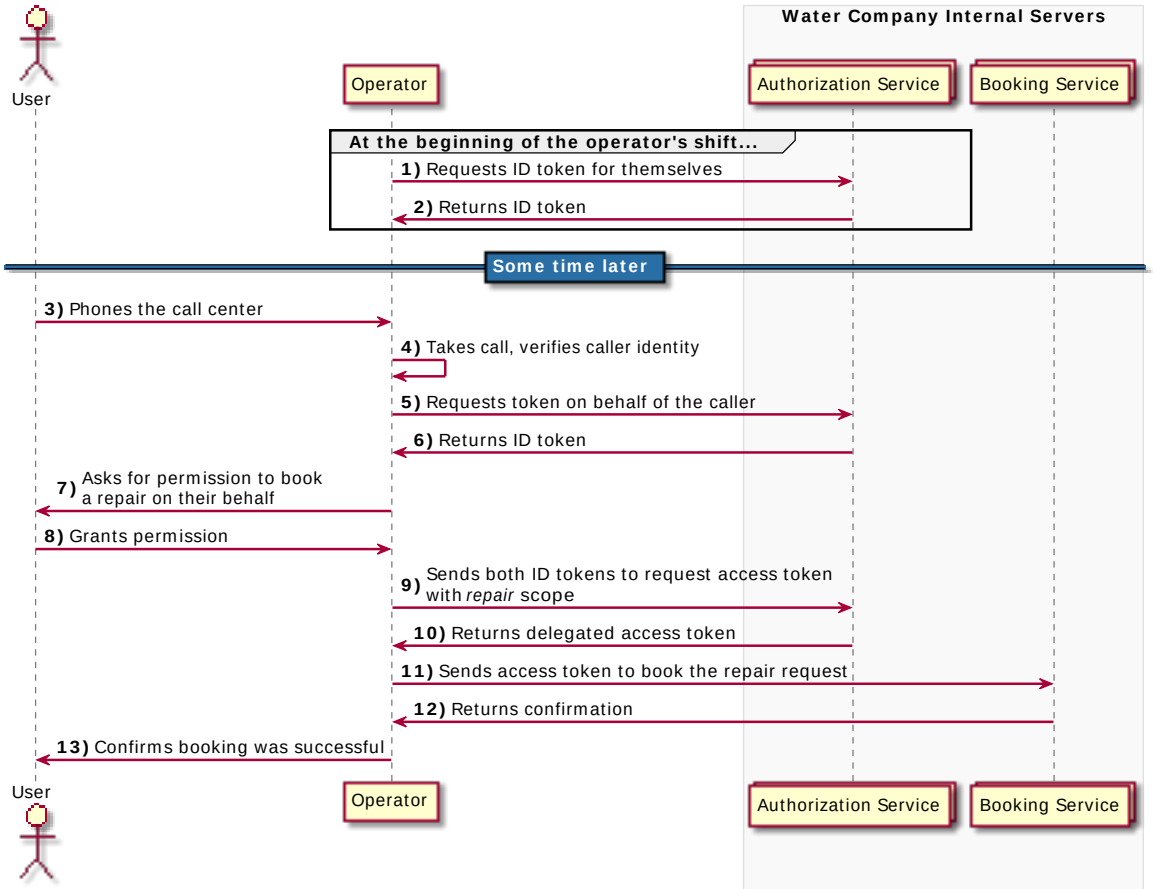
Consider a user who is phoning a call center because of a problem with their water supply. The operator that picks up the phone verifies the identity of the user and creates an ID token (the subject token) on behalf of the user. The operator also creates an access token (the actor token) for themselves.

Then, the operator exchanges both tokens for a delegated access token so that when they book a repair for the user, both the operator and the user are reflected on the repair request:



The delegated token is a restricted token, since it can only be used for booking a repair on behalf of the user, but not for ending the user's contract, for example.

Consider now a scenario where both the subject and the actor tokens are ID tokens. To authorize a repair, the operator could request a delegated access token with the scope of `repair`:



In this case, the delegated token is an expanded token since it has the `repair` scope, which neither of the ID tokens had.

Review the tasks in the following table to configure token exchange in your environment:

Task	Resources
Configure the OAuth2 Provider service and the relevant client profiles for token exchange.	"Configuring AM for Token Exchange"

Task	Resources
Learn how to exchange tokens for impersonation and delegation.	"Token Exchange Flows"

Configuring AM for Token Exchange

To configure AM for token exchange, perform the following tasks:

Task	Procedures
Create at least one script of the type `OAuth2 May Act`, and then configure it in the OAuth2 Provider service. This script adds the `may_act` claim to the issued tokens.	"To Configure a New May Act Script"
Configure the OAuth2 Provider to allow clients to exchange tokens.	"To Configure the OAuth2 Provider Service for Token Exchange"
Configure the relevant client profiles for token exchange.	"To Configure Clients for Token Exchange"

To Configure a New May Act Script

AM provides a script of the type `OAuth2 May Act` to add the `may_act` claims to tokens. Select the script in the OAuth2 Provider service.

No additional configuration is required to add the `act` claim to token exchange delegated tokens. AM automatically adds the claim as needed when issuing them.

Perform the following steps to create a new script and to configure it in the provider service:

1. Go to Realms > *Realm Name* > Scripts.
2. Create a new script of the type `OAuth2 May Act`. Name it, for example, `Test May Act Script`.

   Note that the new script has a comment with the different variables supported by the script. It also has example code to add a `may_act` claim containing a client ID and a subject name.

   You scripts can perform additional tasks as required, but must also add the `may_act` claim to the tokens. For more information about the variables available, see "Token Exchange Scripting API".

   Once configured, AM adds the `may_act` claim to the access tokens, ID tokens, and refresh tokens that meet the the criteria established in the script.

   There is no built-in approach to configure token exchange for particular clients; if you need this functionality, either configure these clients in a specific realm/OAuth2 provider, or customize the script on a per-client basis.

   + *About the may_act Claim*

The `may_act` claim must contain the `client_id` object. In delegation cases, it must also contain the `sub` object. The objects may be formatted as a single entry, or as a collection. For example:

```
"may_act": {
   "client_id": [ "myAppClient", "myOtherAppClient" ],
   "sub": "(usr!demo)"
}
```

+ *About the Subject (sub) Object*

The subject claim is in the format *`(type!subject)`*, where:

- `subject` is the identifier of the user/identity, or the name of the OAuth 2.0/OpenID Connect client that is the subject of the token.
- `type` can be one of the following:
  - `age`. Specifies that the *subject* is an OAuth 2.0/OpenID Connect-related user-agent or client. For example, an OAuth 2.0 client, a Remote Consent Service agent, and a Web and Java Agent internal client.
  - `usr`. Specifies that the *subject* is a user/identity.

For example, `(usr!demo)`, or `(age!myOAuth2Client)`.

The `may_act` claim acts as a condition for the authorization server to issue exchanged tokens, as follows:

- The client making an impersonation or delegation token exchange request must be authorized in the `may_act` claim. Requests from other clients are rejected.
- In delegation cases, the subject of the actor token must also be authorized in the `may_act` claim of the subject token. Requests coming from other subject IDs are rejected.

The following Groovy code demonstrates using the may_act claim for delegation, and possibly impersonation:

```
import org.forgerock.json.JsonValue

token.setMayAct(
  JsonValue.json(JsonValue.object(
    JsonValue.field("client_id", "serviceConfidentialClient"),
    JsonValue.field("sub", "(usr!ForgerockDemo2)"))));
```

The following sections will use these values in the token exchange example calls.

To add multiple values to one of the fields, declare the second object of the field as an array. For example, `JsonValue.field("sub", JsonValue.array("(usr!demo)", "(usr!demo2)"))`.

There is no wildcard-like implementation to allow any client or subject to exchange a token.

You cannot specify in the script the type of token (access or ID token) that AM will add the claim to. You configure that in the OAuth2 Provider service in a later step of this procedure.

3. (Optional) Click on Validate to ensure the script syntax is correct.
4. Save your changes.
5. Go to Realms > *Realm Name* > Services > OAuth2 Provider > Core.
6. Specify the script as follows:

   a. In the OAuth2 Access Token May Act Script drop-down list, select `Test May Act Script`.

   b. In the OIDC ID Token May Act Script drop-down list, select `Test May Act Script`.

   You can use different scripts for OAuth 2.0 and OpenID Connect tokens but, for this example, use the same script. You can also enable token exchange for OAuth 2.0 flows only, or for OpenID Connect flows only.

   To disable token exchange for either OAuth 2.0 or OpenID Connect flows, select `--- Select a script ---` in the appropriate drop-down list.

   AM does not add the `may_act` claim to tokens when a May Act script is not selected.
7. Save your changes.

To Configure the OAuth2 Provider Service for Token Exchange

1. Go to Realms > *Realm Name* > Services > OAuth2 Provider > Advanced.
2. In the Grant Types field, add the `Token Exchange` type if it is not already configured.
3. Review the Token Exchange Plugins and the Token Validator Plugins fields.

   Validator plugins ensure that the subject and/or actor tokens meet the criteria to be exchanged for other tokens. Exchange plugins exchange subject and/or actor tokens into new tokens.

   You may remove any plugins you do not want to allow. For example, you may remove the ID token to ID token exchange plugin. If you need to add plugins back, click on the field and the UI will show you the plugins you can add.

   If you decided to write your own implementations of the plugins, select them here, too.
4. Save your changes, if any.

The OAuth 2.0 provider is now ready to exchange tokens. For more information and examples, see "Token Exchange Flows".

To Configure Clients for Token Exchange

This procedure shows how to configure clients for token exchange using the AM console.

Token exchange is also supported for "*Dynamic Client Registration*" in the *OpenID Connect 1.0 Guide*. Ensure that you provide the configuration explained in this procedure in the JSON representation of the client.

1. Go to Realms > *Realm Name* > Applications > OAuth 2.0 > Clients > *Client Name* > Advanced.
2. (Optional) In the Token Exchange Auth Level field, enter the authentication level in the *Authentication and Single Sign-On Guide* that AM should grant to access tokens generated as a result of token exchange when the subject token had no `auth_level` claim. For example, when the subject token is an ID token.
3. In the Grant Types field, add the `Token Exchange` type.
4. Save your changes.

   Now you are ready to test the "Token Exchange Flows".

Token Exchange Flows

Endpoints

- /oauth2/access_token

Use the token exchange flows to exchange access and ID tokens for impersonated or delegated access and ID tokens, as explained in the OAuth 2.0 Token Exchange specification. For implementation details and use case examples, see "*OAuth 2.0 Token Exchange*".

Example Prerequisites

Tip

Download the ForgeRock OAuth 2.0 and OpenID Connect Postman Collection to configure AM for the examples, and to run the token exchange flows.

The example procedure in this section assumes the following configuration:

- AM is configured as an OAuth 2.0/OpenID Connect provider in a realm called `mySubRealm`, and it is also configured for token exchange.

+ *More Information*

  - "*Authorization Server Configuration*"
  - "To Configure a New May Act Script"
  - "To Configure the OAuth2 Provider Service for Token Exchange"

- Two clients are registered in AM. One is the client that will create the subject token, and the other is the client that will request the token exchange.

  + *Example Client Configuration*

    - `customerConfidentialClient`, the client that will make the requests for the subject tokens, has the following configuration:
      - Client secret: `forgerock`
      - Client type: `Confidential`
      - Redirect URIs: `https://httpbin.org/anything`
      - Scopes:`read` `write` `openid`
      - Token Endpoint Authentication Method:`client_secret_post`

        This is not required; confidential clients can authenticate in several ways, but `client_secret_post` makes the examples easier to read.

      - Response Type:`code`

        This is not required; depending on the flow you use to obtain the tokens, you will need a different response type configuration. This example assumes you will use the Authorization Code grant flow.

        Configure the OAuth 2.0 provider accordingly to the response type.

    - `serviceConfidentialClient`, the client that will make the token exchange requests, has the following configuration:
      - Client secret: `forgerock`
      - Client type: `Confidential`
      - Redirect URIs: `https://httpbin.org/anything`
      - Response Type:`code`

This is not required; depending on the flow you use to obtain the tokens, you will need a different response type configuration. This example assumes you will use the Authorization Code grant flow.

Configure the OAuth 2.0 provider accordingly to the response type.

- Scopes: `read` `write` `transfer`

  The procedure will use the `transfer` scope to extend the scope of a token, which is why the client of the subject token does not have it configured.

- Token Endpoint Authentication Method:`client_secret_post`

  This is not required; confidential clients can authenticate in several ways, but `client_secret_post` makes the examples easier to read.

- Token Exchange Auth Level:`10`

+ *More Information*

- "*Client Registration*"
- "*OAuth 2.0 Client Authentication*"
- "To Configure Clients for Token Exchange"
- "*OAuth 2.0 Grant Flows*"
- "*OpenID Connect Grant Flows*" in the *OpenID Connect 1.0 Guide*

- The `ForgerockDemo` and `ForgerockDemo2` identities are registered in AM.

The following procedure demonstrates how to exchange tokens for both impersonation and delegation cases:

To Exchange Tokens

1. Ensure you have configured AM as per the "Example Prerequisites".

2. Obtain an access token and an ID token for the `ForgerockDemo`/`customerConfidentialClient` user and client combination. Use, for example, the "Authorization Code Grant" in the *OpenID Connect 1.0 Guide*. Do not request the `transfer` scope; you will use it later in the procedure to expand the scopes of the exchanged token.

   Introspect the access token and retrieve in the *OpenID Connect 1.0 Guide* the ID token information to check that they have the `may_act` claim.

If they do not, review the "Configuring AM for Token Exchange" section.

+ *Example Access Token*


```
{
    "active": true,
    "scope": "read write",
    "realm": "/mySubRealm",
    "client_id": "customerConfidentialClient",
    "user_id": "ForgerockDemo",
    "token_type": "Bearer",
    "exp": 1610552102,
    "sub": "(usr!ForgerockDemo)",
    "subname": "ForgerockDemo",
    "iss": "https://openam.example.com:8443/openam/oauth2/mySubRealm",
    "auth_level": 0,
    "authGrantId": "Zcw9MEANrbPjz4tuDLBfzmYrZP0",
    "may_act": {
        "client_id": "serviceConfidentialClient",
        "sub": "(usr!ForgerockDemo2)"
    },
    "auditTrackingId": "961c12ad-0a56-471e-9017-036f3cb873ce-571823"
}
```


Note the value of the `client_id` and `sub` claims.

+ *Example ID Token*


```
{
    "at_hash": "XAhNBCa7Utuc5dujUUA5mQ",
    "sub": "(usr!ForgerockDemo)",
    "auditTrackingId": "961c12ad-0a56-471e-9017-036f3cb873ce-576398",
    "iss": "https://openam.example.com:8443/openam/oauth2/mySubRealm",
    "tokenName": "id_token",
    "nonce": "123abc",
    "sid": "I0GdWDfy1qhahDl1PpEA0v5LDspul+qW70biBhetUCk=",
    "aud": "customerConfidentialClient",
    "c_hash": "0EEiPQxhwezCGa2bPgKYDQ",
    "acr": "0",
    "org.forgerock.openidconnect.ops": "Sj07ATq01pVwzF7Kqop0ZuCCxFg",
    "s_hash": "bKE9UspwyIPg8LsQHkJaiQ",
    "azp": "customerConfidentialClient",
    "auth_time": 1610549052,
    "realm": "/mySubRealm",
    "may_act": {
        "client_id": "serviceConfidentialClient",
        "sub": "(usr!ForgerockDemo2)"
    },
    "exp": 1610552698,
    "tokenType": "JWTToken",
    "iat": 1610549098
}
```

> Note the value of the `aud` and `sub` claims.

Impersonation and delegation examples will use one of these tokens as the subject token.

3. Obtain an access token and an ID token for the `ForgerockDemo2`/ `customerConfidentialClient` user and client combination.

   Delegation examples will use one of these tokens as the actor token.

   Use the examples and the information from the previous step, if needed.

4. To perform a token exchange, the client makes an HTTP POST call to the authorization server's token endpoint. AM will issue a delegation token if the request includes an actor token, and an impersonation token otherwise.

   The request must contain, at least, the following parameters:

   - **grant_type**=`urn:ietf:params:oauth:grant-type:token-exchange`
   - **subject_token**=*token_to_be_exchanged*

     For this example, this is the access token obtained for the `ForgerockDemo`/`customerConfidentialClient` user and client combination.

   - **subject_token_type**=*type_of_subject_token*

     The value of the `subject_token_type` parameter is one of the following, depending of the token you are exchanging:

     + *Supported Subject Token Types*

       - `urn:ietf:params:oauth:token-type:access_token`
       - `urn:ietf:params:oauth:token-type:id_token`

   - **requested_token_type**=*type_of_request_token*

     The value of the `requested_token_type` parameter is one of the following, depending of the token you require:

     + *Supported Requested Token Types*

       - `urn:ietf:params:oauth:token-type:access_token`

- `urn:ietf:params:oauth:token-type:id_token`

For delegation use cases, include the actor token as follows:

- **actor_token**=*token_that_acts_on_behalf_of_the_subject*

  For this example, this is the access token obtained for the `ForgerockDemo2`/`customerConfidentialClient` user and client combination.

- **actor_token_type**=*type_of_actor_token*

  The value of the `actor_token_type` parameter is one of the following, depending of the token you are exchanging:

  + *Supported Actor Token Types*

  - `urn:ietf:params:oauth:token-type:access_token`
  - `urn:ietf:params:oauth:token-type:id_token`

Confidential clients can authenticate to the `/oauth2/access_token` endpoint in several ways. This example uses the following parameters:

- **client_id**=*client_requesting_impersonation*
- **client_secret**=*client_secret*

For more information, see "*OAuth 2.0 Client Authentication*" and "/oauth2/access_token".

The client that makes the request must be the one authorized in the `may_act` claim of the subject token. In delegation cases, the subject of the actor token must also match the subject authorized in `may_act` claim of the subject token.

The following is an example of a request for an *impersonation* token that exchanges an access token for another access token:

```
$ curl --request POST \
--data "client_id=serviceConfidentialClient" \
--data "client_secret=forgerock" \
--data "grant_type=urn:ietf:params:oauth:grant-type:token-exchange" \
--data "scope=read write transfer" \
--data "subject_token=HTVnsWhZhmyM13b-fMsbUqowBqQ" \
--data "subject_token_type=urn:ietf:params:oauth:token-type:access_token" \
--data "requested_token_type=urn:ietf:params:oauth:token-type:access_token" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/mySubRealm/access_token"
```

The following is an example of a request for a *delegation* token that exchanges two access tokens for an ID token:

```
$ curl --request POST \
--data "client_id=serviceConfidentialClient" \
--data "client_secret=forgerock" \
--data "grant_type=urn:ietf:params:oauth:grant-type:token-exchange" \
--data "scope=read write transfer" \
--data "subject_token=HTVnsWhZhmyM13b-fMsbUqowBqQ" \
--data "subject_token_type=urn:ietf:params:oauth:token-type:access_token" \
--data "actor_token=f08f1fcf-3ecb-4120-820d-fb71e3f51c04" \
--data "actor_token_type=urn:ietf:params:oauth:token-type:access_token" \
--data "requested_token_type=urn:ietf:params:oauth:token-type:id_token" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/mySubRealm/access_token"
```

Note that the `scopes` parameter has been added in both cases. Like in the regular OAuth 2.0/ OpenID Connect flows, this parameter is not required, since it can be derived using the same mechanisms AM uses for regular OAuth 2.0/OpenID Connect flows (see "*About Scopes*").

For example purposes, the call requested all the scopes configured in the subject token plus the `transfer` scope, which is only available to this client. This is an example of expanding scopes.

The `openid` scope is not needed to request ID tokens.

Caution

There is no interaction with the user during token exchange, and therefore, it is impossible for AM to request consent about the expanded scopes/claims. It is the responsibility of the client application to ensure that either the user has consented beforehand, or that the expanded scope/claim is unrelated to the user's resources.

Clients should not request more scopes/claims than those required to perform the task requested by the user.

To restrict the scopes/claims in the exchanged token, request fewer scopes/claims than those available to the subject token. For example, `--data "scopes=write"`.

Tip

You can also use exchanged tokens as subject tokens.

+ *Example Output for Different Token Exchanges*

- Access token to access token // ID token to access token :

```
{
    "access_token": "sq2MACbRu6eEGGvoO4_rAEOnNOQ",
    "refresh_token": "wFAPJKb57e-4EGBduApXqFOvyDw",
    "issued_token_type": "urn:ietf:params:oauth:token-type:access_token",
    "scope": "read write",
    "token_type": "Bearer",
    "expires_in": 3599
}
```

Refresh tokens are only issued if AM is configured to issue them.

- Access token to ID token // ID token to ID token:

```
{
    "access_token": "eyJ0eXAiOiJKV1QiLCJraWQiOiJ3VTNpZkl...",
    "refresh_token": null,
    "issued_token_type": "urn:ietf:params:oauth:token-type:id_token",
    "scope": "profile email",
    "token_type": "Bearer",
    "expires_in": 3599
}
```

ID tokens are issued in the `access_token` object. Refresh tokens are never issued with ID tokens.

+ *Error Responses*

- Invalid token exchange

```
{
    "error_description": "Invalid token exchange.",
    "error": "invalid_request"
}
```

HTTP 400. The subject token, the actor token, or both, are invalid, of the wrong type, or cannot be exchanged due to the constrains imposed in the `may_act` claim.

- Subject token type is required

```
{
    "error_description": "Subject token type is required.",
    "error": "invalid_request"
}
```

HTTP 400. The subject token type has not been included in the request.

5. Introspect the token you obtained, and compare it with the subject token you exchanged to understand the differences.

+ *Example of Exchanged Access Token*

Access token-specific claims are not populated unless the subject token contained that information. The only exception is the authentication level, as explained below:

```
{
    "active": true,
    "scope": "read transfer write", ❶
    "realm": "/mySubRealm",
    "client_id": "serviceConfidentialClient", ❷
    "user_id": "ForgerockDemo",
    "token_type": "Bearer",
    "exp": 1610551390,
    "sub": "(usr!ForgerockDemo)", ❸
    "subname": "ForgerockDemo",
    "iss": "https://openam.example.com:8443/openam/oauth2/mySubRealm",
    "auth_level": 10, ❹
    "authGrantId": "VokhnJCdMcvK-opywivldpKJyuk",
    "act": { ❺
        "sub": "(usr!ForgerockDemo2)"
    },
    "may_act": { ❻
        "client_id": "serviceConfidentialClient",
        "sub": "(usr!ForgerockDemo2)"
    },
    "auditTrackingId": "961c12ad-0a56-471e-9017-036f3cb873ce-565818"
}
```

❶ The exchanged token has the scopes requested in the token exchange call. In this case, the exchanged token has the same scopes as the subject token, and the additional `transfer` token, which expands the subject token's authorization.

❷ The client ID has changed to that of the client that requested the exchanged token.

❸ The subject of the exchanged token is the same as that of the subject token. The same is true for any other claims related to the subject, such as `user_id` and `subname`.

+ *About the Subject and the Subname Claims*

The subject claim is in the format *`(type!subject)`*, where:

- `subject` is the identifier of the user/identity, or the name of the OAuth 2.0/OpenID Connect client that is the subject of the token.
- `type` can be one of the following:
  - `age`. Specifies that the *subject* is an OAuth 2.0/OpenID Connect-related user-agent or client. For example, an OAuth 2.0 client, a Remote Consent Service agent, and a Web and Java Agent internal client.
  - `usr`. Specifies that the *subject* is a user/identity.

For example, `(usr!demo)`, or `(age!myOAuth2Client)`.

The value of the `subname` claim matches the value of the *subject* portion of the `sub` claim.

❹ When exchanging an access token for an access token, the authentication level is copied across. When exchanging an ID token for an access token, the authentication level

is the value of the Token Exchange Auth Level field in the client profiles of the the `serviceConfidentialClient` client.

❺ The `act` claim is only present in delegated tokens, and contains the subject of the actor token.

When using delegated tokens as subject tokens, you may see the `act` claim nesting. For example:

```
"act": {
  "sub": "(usr!ForgerockDemo2)",
  "act": {
    "sub": "(usr!ForgerockDemo3)"
  }
}
```

❻ The `may_act` claim will only be present if the token meet the criteria established in the May Act script.

+ *Example of Exchanged ID Token*

ID token-specific claims, such as `acr` and `nonce`, are not populated unless the subject token contained that information.

```
{
   "sub": "(usr!ForgerockDemo)", ❶
   "auditTrackingId": "961c12ad-0a56-471e-9017-036f3cb873ce-607580",
   "iss": "https://openam.example.com:8443/openam/oauth2/mySubRealm",
   "tokenName": "id_token",
   "nonce": "123abc",
   "sid": "I0GdWDfy1qhahDl1PpEA0v5LDspul+qW70biBhetUCk=",
   "aud": "serviceConfidentialClient", ❷
   "acr": "0",
   "org.forgerock.openidconnect.ops": "eVzAOcG-UAyweUaGeEhggWDDWvU",
   "azp": "serviceConfidentialClient",
   "auth_time": 1610552780,
   "realm": "/mySubRealm",
   "may_act": { ❸
       "client_id": "serviceConfidentialClient",
       "sub": "(usr!ForgerockDemo2)"
   },
   "act": { ❹
       "sub": "(usr!ForgerockDemo2)"
   },
   "exp": 1610556942,
   "tokenType": "JWTToken",
   "iat": 1610553342
   }
```

❶ The subject of the exchanged token is the same as that of the subject token.

+ *About the Subject and the Subname Claims*

The subject claim is in the format *`(type!subject)`*, where:

- `subject` is the identifier of the user/identity, or the name of the OAuth 2.0/OpenID Connect client that is the subject of the token.
- `type` can be one of the following:
  - `age`. Specifies that the *subject* is an OAuth 2.0/OpenID Connect-related user-agent or client. For example, an OAuth 2.0 client, a Remote Consent Service agent, and a Web and Java Agent internal client.
  - `usr`. Specifies that the *subject* is a user/identity.

  For example, `(usr!demo)`, or `(age!myOAuth2Client)`.

  The `subname` claim is not included in the `OIDC Claims Script`. Therefore, AM does not add it to ID tokens by default.

❷ The audience is the client ID that requested the exchanged token. Note that the authorized party (`azp`) is also this client ID.

❸ The `may_act` claim will only be present if the token meet the criteria established in the May Act script.

❹ The `act` claim is only present in delegated tokens, and contains the subject of the actor token.

When using delegated tokens as subject tokens, you may see the `act` claim nesting. For example:

```
"act": {
  "sub": "(usr!ForgerockDemo2)",
  "act": {
    "sub": "(usr!ForgerockDemo3)"
  }
}
```

Token Exchange Scripting API

The following properties are available when creating a `OAuth2 May Act Script`:

`clientProperties`

A map of properties configured in the relevant client profile. Only present if the client was correctly identified.

The keys in the map are as follows:

`clientId`

The URI of the client.

`allowedGrantTypes`

The list of the allowed grant types (`org.forgerock.oauth2.core.GrantType`) for the client.

`allowedResponseTypes`

The list of the allowed response types for the client.

`allowedScopes`

The list of the allowed scopes for the client.

`customProperties`

A map of any custom properties added to the client.

Lists or maps are included as sub-maps. For example, a custom property of `customMap[Key1]=Value1` is returned as `customMap` > `Key1` > `Value1`.

To add custom properties to a client, go to OAuth 2.0 > Clients > *Client ID* > Advanced, and then update the Custom Properties field.

`identity`

Contains a representation of the identity of the resource owner.

For more details, see the `com.sun.identity.idm.AMIdentity` class in the ForgeRock Access Management Javadoc.

`logger`

Write information to the AM debug logs.

Created log files have a prefix of `scripts.OAUTH2_ACCESS_TOKEN_MODIFICATION`.

For more information, see "Debug Logging" in the *Getting Started with Scripting*.

`requestProperties`

A map of the properties present in the request. Always present.

The keys in the map are as follows:

`requestUri`

The URI of the request.

`realm`

The realm to which the request was made.

`requestParams`

The request parameters, and/or posted data. Each value in this map is a list of one, or more, properties.

> **Important**
>
> To mitigate the risk of reflection-type attacks, use OWASP best practices when handling these properties. For example, see Unsafe use of Reflection.

`scopes`

Contains a set of the requested scopes. For example:

```
[
  "read",
  "transfer",
  "download"
]
```

`scriptName`

The display name of the script. Always present.

`session`

Contains a representation of the user's session object if the request contained a session cookie.

For more details, see the `com.iplanet.sso.SSOToken` class in the ForgeRock Access Management Javadoc.

`token`

Contains a representation of the token to be updated. As a mutable object, any changes made are reflected in the resulting token.

Use the following method when performing token exchange:

Token Exchange Methods

Method	Information
`token.setMayAct(JsonValue value)`	Adds the `may_act` claim to a token. See "To Configure a New May Act Script".

For more details, see the `ExchangeableToken` interface in the ForgeRock Access Management Javadoc.

Chapter 14

OAuth 2.0 Endpoints

When acting as an OAuth 2.0 authorization server, AM exposes the following endpoints:

OAuth 2.0 Endpoints

Endpoint	Description
`/oauth2/authorize`	Obtain consent and an authorization grant (RFC 6749 authorization endpoint)
`/oauth2/bc-authorize`	Initiate backchannel authorization (Backchannel flow endpoint)
`/oauth2/access_token`	Obtain an access token (RFC 6749 token endpoint)
`/oauth2/device/code`	Obtain a device code (Device flow endpoint)
`/oauth2/device/user`	Obtain consent and authorization grant (Device flow endpoint)
`/oauth2/token/revoke`	Revoke both access and refresh tokens (RFC 7009 endpoint)
`/oauth2/introspect`	Retrieve metadata about a token, such as approved scopes and the context in which the token was issued (RFC 7662 endpoint)
`/json/token/macaroon`	Retrieve metadata about a macaroon, and add caveats.

Tip

As an OAuth 2.0/OpenID Connect/UMA provider, AM also exposes the following:

- OAuth 2.0 endpoints to perform administrative tasks, such as creating clients. For more information, see "*OAuth 2.0 Administration and Supporting REST Endpoints*"
- OpenID Connect-specific endpoints. For more information, see "*OpenID Connect 1.0 Endpoints*" in the *OpenID Connect 1.0 Guide*.
- UMA-specific endpoints. For more information, see "*UMA Endpoints*" in the *User-Managed Access (UMA) 2.0 Guide*.

/oauth2/authorize

The `/oauth2/authorize` endpoint is the OAuth 2.0 authorization endpoint as defined in RFC 6749. Use this endpoint to gather consent and authorization from the resource owner when using the following flows:

- Authorization Code Grant (OAuth 2.0) | OpenID Connect)
- Authorization Code Grant with PKCE (OAuth 2.0) | OpenID Connect)
- Implicit Grant (OAuth 2.0) | OpenID Connect)
- UMA Grant (UMA)

You must compose the path to the authorize endpoint addressing the specific realm where the access code will be issued. For example, `https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/authorize`.

The authorization endpoint supports the following parameters:

`client_id`

Specifies the client ID unique to the application making the request.

Required: Yes.

`response_type`

Specifies the type of response expected from the authorization server. Set this parameter to one of the following values:

- `code`, to request an authorization code during the Authorization Code grant flow.
- `token`, to request an access token during the Implicit grant flow.
- `id_token`, to request an ID token during the Implicit grant flow
- `code token`, to request an authorization code and an access token during the Hybrid grant flow.
- `code id_token`, to request an authorization code and an ID token during the Hybrid grant flow.
- `code token id_token`, to request an authorization code, an access token, and an ID token during the Hybrid grant flow.
- `token id_token`, to request an access token and an ID token during the Implicit grant flow.
- `none`, to request AM *not* to issue any token or code in the request. Use this response type in conjunction with the `id_token_hint` parameter only.

Required: Yes.

`csrf`

When interacting with the OAuth 2.0 consent page, this parameter helps prevent against Cross-Site Request Forgery (CSRF) attacks.

The parameter duplicates the contents of the `iPlanetDirectoryPro` cookie, which contains the SSO token of the resource owner giving consent.

When using the AM consent pages, this parameter is set in the consent page once the resource owner has authenticated, and it is sent to AM along with the consent.

When replacing AM consent pages with your own consent pages or when trying the flows without a browser, you must set this parameter manually. For an example of a curl command, see the Authorization Code Grant.

Required: Yes, unless you use the Remote Consent Service to gather consent.

`code_challenge`

Specifies a string derived from the code verifier that is sent in the authorization request during the Authorization Code with PKCE grant flow.

Required: Yes, when requesting an authorization code during the Authorization Code with PKCE grant flow.

`code_challenge_method`

Contains the method used to derive the code challenge. Possible values are `plain` and `S256`. When unset, it defaults to `plain`.

Required: Yes, when requesting an authorization code during the Authorization Code with PKCE grant flow and the code challenge was created using an SHA256 algorithm.

`decision`

Specifies whether the resource owner consents to the requested access. Set to `allow` to grant consent. Any other value denies consent.

Required: Yes, unless consent is already saved for the scope.

`redirect_uri`

The URI to return the resource owner to after authorization is complete. If not set, the redirection URI defaults to that configured in the client profile registered with AM.

Required: No.

`response_mode`

Set to `form_post` to return a self-submitting form that contains the code instead of redirecting to the redirect URL with the code as a string parameter. For more information, see the OAuth 2.0 Form Post Response Mode spec.

Required: No.

`scope`

Specify the scopes linked to the permissions requested by the client from the resource owner. If not specified, the default scopes specified in the client or the authorization server are requested.

Required: No.

`save_consent`

Updates the resource owner's profile to avoid having to prompt the resource owner to grant authorization when the client issues subsequent authorization requests.

Set this parameter to `on` to save the consent.

To save the consent, you must have configured the Saved Consent Attribute Name property with a profile attribute in which to store the resource owner's consent decision.

For more information on setting this property in the OAuth2 Provider service, see "OAuth2 Provider" in the *Reference*.

Required: No.

`service/module`

Use either as described in "Authentication Parameters" in the *Authentication and Single Sign-On Guide*, where `module` specifies the authentication module instance to use, or `service` specifies the authentication tree or chain to use when authenticating the resource owner.

If not specified, the resource owner authenticates using the default chain or tree configured for the realm.

Required: No.

`state`

Value to maintain state between the request and the callback. During authentication, the client sends this parameter to the authorization server. The authorization server must send it back unchanged in the response.

The application should use this value to ensure the response belongs to the user that initiated the requests, which mitigates CSRF attacks.

The value of `state` is typically a base64-encoded string that contains user state and that is unique to a user and their request.

Required: No, but it is strongly recommended.

`acr_values`

Authentication Context class Reference values used to communicate acceptable LoAs that users must satisfy when authenticating to the OpenID provider.

For more information, see "*Adding Authentication Requirements to ID Tokens*" in the *OpenID Connect 1.0 Guide*.

Required: No. OIDC flows only.

`claims`

Specifies a JSON object containing specific attributes about users to be returned in the ID Token.

Required: No. OIDC flows only.

`id_token_hint`

ID token previously issued by AM that is passed as a hint about the end user's session with the client. Using this parameter requires the `response_type` and `prompt` parameters to be set to `none`.

For more information about using the `id_token_hint` parameter, see "Session Management Draft 10" in the *OpenID Connect 1.0 Guide*.

Required: No. OIDC flows only.

`login_hint`

String value that can be set to the ID the user uses to log in. For example, `Bob` or `bob@example.com`, depending on how the authentication node or module is configured to search for users.

When provided as part of the OIDC Authentication Request, the `login_hint` is set as the value of a cookie named `oidcLoginHint`, which is an HttpOnly cookie (only sent over HTTPS).

For more information, see "*GSMA Mobile Connect*" in the *OpenID Connect 1.0 Guide*.

Required: No. OIDC flows only.

`nonce`

String value that associates the client session with the ID token that also mitigates against replay attacks. For more information, see the OpenID Connect 1.0 Guide.

Required: No. OIDC flows only.

`prompt`

A space-separated, case sensitive list of ASCII values that specifies whether AM should prompt the end user for authentication and consent. Possible values are:

- `none`. AM does not display authentication or consent pages. Use with the `id_token_hint` and the `response_type=none` parameters only.
- `login`. AM prompts the end user to authenticate to the default service of their realm, or to the service provided in the `service` parameter.

  If the user reauthenticates to a tree, AM destroys the original session and provides them with a new one that reflects the new authentication journey.

  If the user reauthenticates to a chain, AM updates the original session to reflect the new authentication journey.

Note

It is strongly recommended that users are required to authenticate using trees, not chains, when `prompt=login` leads to reauthentication *at the same level*. This recreates the session and mitigates the threat of session fixation attacks.

- `consent`. AM prompts the end user to grant consent, even if a consent response was previously saved.

Required: No. OIDC flows only.

`ui_locales`

Specifies a space-separated list of the end user preferred languages for the user interface, ordered by preference. For example, `en fr-CA fr`.

Required: No. OIDC flows only.

`request`

As per the OpenID Connect specification, this parameter specifies a base64url-encoded JWT whose claims are the query parameters required for the OpenID Connect flow. This JWT is called the request object.

You may send query string parameters and a request object in the same request to AM. This is useful to keep sensitive information protected in the request object, and to ensure that parameters whose value changes frequently, such as `nonce` and `state`, remain visible and mutable across calls.

The value of the claims included in the request object supersede the value passed as query string parameters, but some claims/parameters must be configured and sent in a certain manner for the request to be valid. You must:

- Include the value of `response_type` and `client_id` as query string parameters, regardless of whether they are included in the request object or not.

  If they are included in the request object, their values must match those passed as query string parameters.

- Include the `openid` scope as a query string parameter, regardless of whether it is included in the request object or not.

  The value of the `scope` claim may differ from that passed as a query parameter. This is useful to protect application-related scopes inside the request object, yet allows AM to process the request as part of an OpenID Connect flow.

You must follow these rules as well if you're passing the request object as a reference using the `request_uri` parameter.

The following is an example of a request object. Note that it includes the `iss` and `aud` JWT claims in addition to the OpenID Connect claims:

```
{
  "iss": "myClient",
  "client_id": "myClient",
  "aud": "https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha",
  "redirect_uri": "https://www.example.com:8443",
  "scope" : "openid profile",
  "claims":
    {
      id_token":
        {
          "acr":
            {
               "essential": true,
               "values": ["example_tree1","example_tree2"]
            }
        }
    }
}
```

The JWT can be signed and/or encrypted, in which case you should always include the `iss` and `aud` parameters in the JWT as shown in the example. However, AM ignores keys specified in JWT headers, such as `jku` and `jwe`.

If you are compressing the JWT, note that by default, AM rejects JWTs that expand to a size larger than 32 KiB (32768 bytes). For more information, see "Controlling the Maximum Size of Compressed JWTs" in the *Security Guide*.

To retrieve a list of public keys clients can use to encrypt this JWT, make a request to the realm's JWK URI endpoint in the *OpenID Connect 1.0 Guide*.

The following is an example call sending the request object as value:

```
https://openam.example.com:8443/openam/oauth2/realms/root/authorize? \
&request=eyJhbGciOiJSUzI1NiIsImtpZCI6ImsyYmRjIn0.ew0KICJpc3MiOiAiczZCaGRSa3.... \
&client_id=myClient \
&scope=openid profile\
&response_type=code%20id_token \
&nonce=abc123 \
&state=123abc
```

Note that the URL is split for readability purposes.

Required: No. OIDC flows only.

`request_uri`

Specifies an array of URIs from which AM can retrieve a base64-encoded JWTs whose claims are the request parameters required for the OpenID Connect flow. For information about the JWT, see the `request` parameter.

The URI must be pre-registered in the Requests uris field (Realms > *Realm Name* > Applications > OAuth 2.0 > Clients > *Client Name* > Advanced).

Each of the request URIs must not exceed 512 ASCII characters and must use either HTTP or HTTPS. For example, `https://www.example.com:8443/JTWs/myJWT`.

AM caches the retrieved JWT to avoid fetching it too often. To force AM to flush the cache, add a unique fragment to the `request_uri` parameter. For example, `?request_uri=https://www.example.com:8443/JTWs/myJWT#foo`.

Required: No. OIDC flows only.

/oauth2/bc-authorize

The `/oauth2/bc-authorize` endpoint is the backchannel authorization endpoint as used by OpenID Connect Client Initiated Backchannel Authentication Flow - Core 1.0 draft-02. Use this endpoint to initiate backchannel authorization with the resource owner when using the following flow:

- Backchannel Request Grant (OpenID Connect)

You must compose the path to the backchannel authorization endpoint addressing the specific realm where the authorization request ID will be issued. For example, `https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/bc-authorize`.

The endpoint supports the following parameters:

`client_id`

Specifies the client ID unique to the application making the request.

Required: Yes.

`client_secret`

Specifies the password of the private client making the request. Do not use in conjunction with the `cnf_key` parameter.

Required: A form of password or credentials is required for confidential clients. However, the use of the `client_secret` parameter depends on the client authentication method used. For more information, see "*OAuth 2.0 Client Authentication*".

`client_assertion`

Specifies the signed JWT that the client uses as a credential when using the JWT bearer client authentication method.

For more information, see "*OAuth 2.0 Client Authentication*".

Required: Yes, when using the JWT bearer client authentication method.

`client_assertion_type`

Specifies the type of assertion when the client is authenticating to the authorization server using JWT bearer client authentication. Do not use with other client authentication methods.

Set it to `urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer`.

For more information, see "*OAuth 2.0 Client Authentication*".

Required: Yes, when using the JWT bearer client authentication method.

The endpoint requires a signed JWT that contains the following parameters:

aud

Specifies a string or an array of strings that is the intended audience of the JWT. Must be set to the authorization server's OAuth 2.0 endpoint, for example:

```
"aud": "http://openam.example.com:8080/openam/oauth2"
```

exp

Specifies the expiration time of the JWT in Unix time.

Providing a JWT with an expiry time greater than 30 minutes causes AM to return a `JWT expiration time is unreasonable` error message.

iss

Specifies the unique identifier of the JWT issuer.

The identifier must match the client ID of the OAuth 2.0 client in AM, for example *myCIBAClient*.

login_hint

Specifies the principal who is the subject of the JWT. It should be a string that identifies the resource owner.

> **Tip**
>
> You can provide a previously obtained ID token in a property named **id_token_hint** as the hint for determining the resource owner, rather than a string.

scope

Specifies a space-separated list of the requested scopes. Must include the `openid` scope.

acr_values

Specifies an identifier that maps to the authentication mechanism AM uses to obtain authorization from the end user.

binding_message

Specifies a message delivered to the user when obtaining authorization.

Should be a short (100 characters or fewer), description of the operation the end user is authorizing, and should include an identifier to match the authorization request to the client that initiated the request.

Note

If the binding message is sent using push notifications, the following additional limitations apply to the value:

1. Must begin with a letter, number, or punctuation mark.
2. Must **not** include line breaks or control characters.

For example:

```
Allow ExampleBank to transfer £50 from your 'Main' account to your 'Savings' account? (EB-0246326)
```

/oauth2/access_token

The `/oauth2/access_token` endpoint is the OAuth 2.0 token endpoint as defined in RFC 6749. Use this endpoint to acquire an access or refresh token when using the following flows:

- Authorization Code Grant (OAuth 2.0) | OpenID Connect)
- Authorization Code Grant with PKCE (OAuth 2.0) | OpenID Connect)
- Client Credentials Grant (OAuth 2.0)
- Resource Owner Password Credentials Grant (OAuth 2.0)
- Device Flow (OAuth 2.0)
- SAML v2.0 Profile for Authorization Grant (OAuth 2.0)
- Token Exchange Flows (OAuth 2.0/OpenID Connect)

You must compose the path to the token endpoint addressing the specific realm where the token will be issued. For example, `https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/token`.

The token endpoint supports the following parameters:

`grant_type`

Specifies the type of grant to send to the authorization server to acquire an access token.

The following types are supported:

- `password`, for the Resource Owner Credentials grant flow.
- `authorization_code`, for the Authorization Code Grant (OAuth 2.0) | OpenID Connect) and Authorization Code Grant with PKCE (OAuth 2.0) | OpenID Connect) grant flows.
- `client_credentials`, for the Client Credentials grant flow.

- `urn:ietf:params:oauth:grant-type:device_code`, for the Device Flow. An earlier specification, `http://oauth.net/grant_type/device/1.0`, is also supported.
- `urn:openid:params:grant-type:ciba`, for the Client Initiated Backchannel Authentication (CIBA) flow. For more information, see "Backchannel Request Grant" in the *OpenID Connect 1.0 Guide*.
- `urn:ietf:params:oauth:grant-type:uma-ticket`, for the UMA grant flow. For more information, see "*The UMA Grant Flow*" in the *User-Managed Access (UMA) 2.0 Guide*.
- `refresh_token`, to refresh an access token. For more information, see "*Refresh Tokens*".
- `urn:ietf:params:oauth:grant-type:saml2-bearer`, for the SAML v2.0 Profile for Authorization grant. For more information, see "SAML v2.0 Profile for Authorization Grant".
- `urn:ietf:params:oauth:grant-type:jwt-bearer`, for the JWT Profile for OAuth 2.0 Authorization grant. For more information, see "JWT Profile for OAuth 2.0 Authorization Grant".
- `urn:ietf:params:oauth:grant-type:token-exchange`, for the Token Exchange flows. For more information, see "Token Exchange Flows".

Required: Yes

`client_id`

Specifies the client ID unique to the application making the request.

Required: Yes.

`client_secret`

Specifies the secret of the client making the request. Do not use in conjunction with the `cnf_key` parameter.

Required: A form of password or credentials is required for confidential clients. However, the use of the `client_secret` parameter depends on the client authentication method used. For more information, see "*OAuth 2.0 Client Authentication*".

`cnf_key`

Specifies either a base64-encoded JWK used to support "JWK-Based Proof-of-Possession" or a base64-encoded SHA-256 hash of the DER-encoding of a full X.509 certificate to support "Certificate-Bound Proof-of-Possession".

Do not use in conjunction with the `client_secret` parameter.

Required: Yes, when using JWK proof-of-possession.

`username`

Specifies the username of the resource owner during the Resource Owner Credentials grant flow.

Required: Yes, when `grant_type` is set to `password`.

`password`

Specifies the password of the resource owner during the Resource Owner Credentials grant flow.

Required: Yes, when `grant_type` is set to `password`.

`code`

Specifies the authorization code obtained during the Authorization Code grant and Authorization Code with PKCE grant flows.

Required: Yes, when `grant_type` is set to `authorization_code`.

`device_code`

Specifies the device code obtained when requesting a user code during the Device flow.

Required: Yes, when `grant_type` is set to `urn:ietf:params:oauth:grant-type:device_code`.

`client_assertion`

Specifies the signed JWT that the client uses as a credential when using the JWT bearer client authentication method.

For more information, see "*OAuth 2.0 Client Authentication*".

Required: Yes, when using the JWT bearer client authentication method.

`client_assertion_type`

Specifies the type of assertion when the client is authenticating to the authorization server using JWT bearer client authentication. Do not use with other client authentication methods.

Set it to `urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer`.

For more information, see "*OAuth 2.0 Client Authentication*".

Required: Yes, when using the JWT bearer client authentication method.

`assertion`

Specifies a SAML v2.0 assertion. The assertion must be first base64-encoded, and then URL encoded. For more information, see "SAML v2.0 Profile for Authorization Grant".

Required: Yes, when using the SAML v2.0 Profile for Authorization grant.

`redirect_uri`

The URI to return the resource owner to after authorization is complete. Must match the `redirect_uri` configured in the client profile registered with AM, and the `redirect_uri` set when requesting authorization.

The URI must be an absolute URI, and must not contain a fragment component. For example, `https://www.example.com:443/callback/`.

Required: Yes, when `grant_type` is set to `authorization_code` and it was included in the authorization code grant, and during the Implicit grant.

`code_verifier`

Specifies a random string that correlates the authorization request to the token request in the Authorization Code with PKCE grant flow.

Required: Yes, when requesting an access code in the Authorization Grant with PKCE flow.

`subject_token`

The original token to be exchanged as part of delegation or impersonation Token Exchange flows.

Required: Yes, when requesting tokens during the delegation and/or impersonation Token Exchange flows.

`subject_token_type`

The type of the subject token. Possible values are:

- `urn:ietf:params:oauth:token-type:access_token`
- `urn:ietf:params:oauth:token-type:id_token`

Required: Yes, when requesting tokens during the delegation and/or impersonation Token Exchange flows.

`actor_token`

The original token that acts on behalf of the subject token during delegation Token Exchange flows.

Required: Yes, when requesting a token in the delegation Token Exchange flow.

`actor_token_type`

The type of the actor token. Possible values are:

- `urn:ietf:params:oauth:token-type:access_token`
- `urn:ietf:params:oauth:token-type:id_token`

Required: Yes, when requesting a token in the delegation Token Exchange flow.

`requested_token_type`

The type of token requested as part of an impersonation or delegation Token Exchange flow. Possible values are:

- `urn:ietf:params:oauth:token-type:access_token`
- `urn:ietf:params:oauth:token-type:id_token`

If not added to the request, it defaults to access tokens.

Required: No, but adding it is highly recommended when requesting tokens during the delegation and/or impersonation Token Exchange flows.

`scope`

Specify the scopes linked to the permissions requested by the client from the resource owner. If not specified, the default scopes specified in the client or the authorization server are requested.

Note that some grant flows, such as the Authorization Code grant, do not call the token endpoint with the scope. The scope is already defined in the authorization code. For details, see the specific grant flow documentation in "*OAuth 2.0 Grant Flows*".

For more information, see "*About Scopes*" and "*About Claims*" in the *OpenID Connect 1.0 Guide*.

Required: No.

`auth_chain`

Overrides the authentication tree or chain configured for the realm, and also the tree or chain configured in the OAuth 2.0 service in the realm, when supporting the Resource Owner Credentials grant flow.

By default, the Resource Owner Password Credentials grant flow uses the default authentication tree or chain in the relevant realm.

The selected tree or chain must be configured for requiring username and password only, without UI-based interaction from the resource owner. For example, using the `ldapService` chain or `Example` tree. If this is not the case, the server returns an HTTP 500 error message.

Required: No.

`refresh_token`

Specifies the refresh token that will be used to refresh an access token.

For more information, see "*Refresh Tokens*".

Required: No, only when refreshing access tokens.

/oauth2/device/code

Device Flow endpoint as defined by the Internet-Draft OAuth 2.0 Device Flow. Client devices use this endpoint to present a user code to the resource owner that can be exchanged for an access token in the Device Flow (OAuth 2.0) flow.

You must compose the path to the device code endpoint addressing the specific realm where the user code will be issued. For example, `https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/device/code`.

The device code endpoint supports the following parameters:

`response_type`

Specifies the response expected from the authorization server. Set to `device_code` to obtain a user code.

Required: Yes.

`client_id`

Specifies the client ID unique to the application making the request.

Required: Yes.

`state`

Value to maintain state between the request and the callback. During authentication, the client sends this parameter to the authorization server. The authorization server must send it back unchanged in the response.

The application should use this value to ensure the response belongs to the user that initiated the requests, which mitigates CSRF attacks.

The value of `state` is typically a base64-encoded string that contains user state and that is unique to a user and their request.

Required: No, but it is strongly recommended.

`scope`

The scopes attached to the permissions requested from the resource owner by the client. If not specified, the default scopes specified in the client or the authorization server are requested.

Required: Yes.

`code_challenge`

Specifies a string derived from the code verifier that is sent in the authorization request during the Device with PKCE flow.

Required: Yes, when obtaining a user code in the Device with PKCE Flow.

`code_challenge_method`

Contains the method used to derive the code challenge. Possible values are `plain` and `S256`. When unset, it defaults to `plain`.

Required: Yes, when obtaining a user code in the Device with PKCE Flow.

`nonce`

String value that associates the client session with the ID Token that also mitigates against replay attacks.

Required: No. OpenID Connect flows only.

`acr_values`

Authentication Context class Reference values used to communicate acceptable authentication chains or trees.

For more information, see "*Adding Authentication Requirements to ID Tokens*" in the *OpenID Connect 1.0 Guide*.

Required: No. OpenID Connect flows only.

`prompt`

A space-separated, case sensitive list of ASCII values that specifies whether AM should prompt the end user for authentication and consent. Possible values are:

- `none`. AM does not display authentication or consent pages.
- `login`. AM prompts the end user to authenticate.
- `consent`. AM prompts the end user to grant consent.

Required: No. OpenID Connect flows only.

`ui_locales`

Specifies a space-separated list of the end user preferred languages for the user interface, ordered by preference. For example, `en fr-CA fr`.

Required: No. OpenID Connect flows only.

`login_hint`

String value indicating the ID to use for login.

When provided as part of the OpenID Connect Authentication Request, the `login_hint` is set as the value of a cookie named `oidcLoginHint`, which is an HttpOnly cookie (only sent over HTTPS). Authentication modules can then retrieve the cookie's value.

For more information, see "*GSMA Mobile Connect*" in the *OpenID Connect 1.0 Guide*.

Required: No. OpenID Connect flows only.

`claims`

Specifies a JSON object containing specific attributes about users to be returned in the ID Token.

Required: No. OpenID Connect flows only.

`claim_token`

Specifies an ID token containing the claims gathered during the UMA grant flow. Use together with the `claim_token_format` parameter.

Required: No. UMA Grant flow only.

`claim_token_format`

Specifies that the type of the token gathered during the UMA grant flow. AM only supports ID tokens as claim tokens. Possible values are:

- `https://openid.net/specs/openid-connect-core-1_0.html#IDToken`

Use together with the `claim_token` parameter.

Required: No. UMA Grant flow only.

/oauth2/device/user

Device Flow AM-specific endpoint for user interaction. Client devices use this endpoint to exchange a user code with consent from the resource owner to access the resources in the Device Flow (OAuth 2.0).

You must compose the path to the device user endpoint addressing the specific realm where consent will be granted. For example, `https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/device/user`.

The device user endpoint supports the following parameters:

`user_code`

Specify the scopes linked to the permissions requested by the client to the resource owner. If not specified, the default scopes specified in the client or the authorization server are requested.

Required: Yes.

`csrf`

When interacting with the OAuth 2.0 consent page, this parameter helps prevent against Cross-Site Request Forgery (CSRF) attacks.

The parameter duplicates the contents of the `iPlanetDirectoryPro` cookie, which contains the SSO token of the resource owner giving consent.

When using the AM consent pages, this parameter is set in the consent page once the resource owner has authenticated, and it is sent to AM along with the consent.

When replacing AM consent pages with your own consent pages or when trying the flows without a browser, you must set this parameter manually. For an example of a curl command, see the Authorization Code Grant.

Required: Yes, for calls that are submitting consent response, unless you use the Remote Consent Service to gather consent.

`scope`

Specify the scopes linked to the permissions requested by the client to the resource owner. If not specified, the default scopes specified in the client or the authorization server are requested.

Required: No.

`decision`

Specifies whether the resource owner consents to the requested access. Set to `allow` to grant consent. Any other value denies consent.

Required: Yes, to submit consent on non-interactive calls, unless consent is already saved for the scope.

`save_consent`

Updates the resource owner's profile to avoid having to prompt the resource owner to grant authorization when the client issues subsequent authorization requests.

Set this parameter to `on` to save the consent.

To save the consent, you must have configured the *Saved Consent Attribute Name* property with a profile attribute in which to store the resource owner's consent decision.

For more information on setting this property in the OAuth2 Provider service, see "OAuth2 Provider" in the *Reference*.

Required: No.

/oauth2/token/revoke

Endpoint defined in RFC7009 - Token Revocation, used to revoke both access and refresh tokens.

Revoking a refresh token also revokes any other associated tokens that were issued with the same authorization grant. If a client has multiple access tokens for a single user that were obtained using different authorization grants, the client would need to make multiple calls to the revoke token endpoint to invalidate each token.

The revoke token endpoint supports the following parameters:

`token`

Specifies the token ID that will be revoked.

Required: Yes.

`client_id`

Specifies the client ID unique to the application making the request.

Required: Yes.

`client_secret`

Specifies the password of the private client making the request. Do not use in conjunction with the `cnf_key` parameter.

Required: A form of password or credentials is required for confidential clients. However, the use of the `client_secret` parameter depends on the client authentication method used. For more information, see "*OAuth 2.0 Client Authentication*".

`client_assertion`

Specifies the signed JWT that the client uses as a credential when using the JWT bearer client authentication method.

For more information, see "*OAuth 2.0 Client Authentication*".

Required: Yes, when using the JWT bearer client authentication method.

`client_assertion_type`

Specifies the type of assertion when the client is authenticating to the authorization server using JWT bearer client authentication. Do not use with other client authentication methods.

Set it to `urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer`.

For more information, see "*OAuth 2.0 Client Authentication*".

Required: Yes, when using the JWT bearer client authentication method.

You must compose the path to the revoke token endpoint addressing the specific realm where the user code was issued. For example, `https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/revoke`.

The following is an example of how to revoke a given token:

```
$ curl --request POST \
--data "token=xS3UjtuXMu77iNzl2XibpeMlw1g" \
--data "client_id=myClient" \
--data "client_secret=forgerock" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/token/revoke"
{}
```

/oauth2/introspect

Endpoint defined in RFC7662 - OAuth 2.0 Token Introspection, used to retrieve metadata about a token, such as, approved scopes, the user that authorized the token, the expiry time, the UMA RPT, or the proof-of-possession JWK.

As opposed to the `/oauth2/tokeninfo` endpoint, the `/oauth2/introspect` endpoint requires the resource server to authenticate to the authorization server.

+ *Introspecting macaroon and UMA RPT tokens*

- To introspect macaroon access tokens containing third-party caveats, use the `X-Discharge-Macaroon` header to pass the discharge macaroon.
- To introspect UMA RPT tokens, use the PAT of the resource owner in an `authorization: Bearer` header to authenticate to the endpoint.

Important

From AM 7.1.3 onwards, HTTP GET requests are disallowed on the `/oauth2/introspect` endpoint by default. Using `token` as a query parameter on this endpoint is also disallowed. To change this behavior to suit existing clients, use the `org.forgerock.openam.introspect.token.query.param.allowed` advanced server property.

The token introspection endpoint supports the following parameters:

`token`

Specifies the token ID.

Required: Yes.

`client_id`

Specifies the client ID unique to the application making the request.

Required: A form of credentials is required for confidential clients. However, the use of the `client_id` parameter depends on the client authentication method used. For more information, see "*OAuth 2.0 Client Authentication*".

`client_secret`

Specifies the secret of the client making the request.

Required: A form of password or credentials is required for confidential clients. However, the use of the `client_secret` parameter depends on the client authentication method used. For more information, see "*OAuth 2.0 Client Authentication*".

`client_assertion`

Specifies the signed JWT that the client uses as a credential when using the JWT bearer client authentication method.

For more information, see "*OAuth 2.0 Client Authentication*".

Required: Yes, when using the JWT bearer client authentication method.

`client_assertion_type`

Specifies the type of assertion when the client is authenticating to the authorization server using JWT bearer client authentication. Do not use with other client authentication methods.

Set it to `urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer`.

For more information, see "*OAuth 2.0 Client Authentication*".

Required: Yes, when using the JWT bearer client authentication method.

You must compose the path to the introspect endpoint addressing the specific realm where the token was issued. For example, `https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/introspect`.

By default, and for security reasons, clients can only introspect their own tokens. To let a client introspect access tokens issued to other clients, see "Special Scopes".

The following example shows AM returning token information:

```
$ curl \
--request POST \
--header "Authorization: Basic ZGVtbzpDaDRuZzMxdA==" \
--data "token=f9063e26-3a29-41ec-86de-1d0d68aa85e9" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/introspect"
{
    "active": true,
    "scope": "write",
    "client_id": "myClient",
    "user_id": "demo",
    "token_type": "Bearer",
    "exp": 1419356238,
    "sub": "(usr!demo)",
    "subname" : "demo",
    "iss": "https://openam.example.com:8443/openam/oauth2"
    "cnf": {
        "jwk": {
            "alg": "RS512",
            "e": "AQAB",
            "n": "k7qLlj...G2oucQ",
            "kty": "RSA",
            "use": "sig",
            "kid": "myJWK"
        },
    "auth_level": 0
    }
}
```

The introspection response, as specified in RFC7662, is a plain JSON object. However, AM also supports the JWT Response for OAuth Token Introspection Internet Draft, which adds signed JWT or signed then encrypted JWT responses.

To configure the response type, perform the following steps:

1. Navigate to Realms > *Realm Name* > Applications > OAuth 2.0 > Clients > *Client Name* > Signing and Encryption.
2. In the Token introspection response format drop-down list, select the type of response required by the client.
3. Configure the signing and/or encryption settings AM should use when returning introspection responses to this particular client in the following properties:
   - Token introspection response signing algorithm
   - Token introspection response encryption algorithm
   - Token introspection encrypted response encryption algorithm

   For more information about these properties, see Signing and Encryption Properties.

Tip

Even if the client has configured the response to be JSON-formatted, it can request a signed JWT by adding the `"Accept: application/jwt"` header to the request. For example:

```
$ curl \
--request POST \
--header "Accept: application/jwt" \
--header "Authorization: Basic ZGVtbzpDaDRuZzMxdA==" \
--data "token=f9063e26-3a29-41ec-86de-1d0d68aa85e9" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/introspect"
```

However, when a client that is configured to use either of the JWT-formatted responses requests a JSON response, AM returns an error.

The following is a list of the objects commonly returned by the token introspect endpoint:

`active`

Specifies whether the token is active (true) or not (false).

`scope`

Specifies a space-separated list of the scopes associated with the token.

`client_id`

Specifies the client that requested the token.

`user_id`

(Deprecated, defined in a previous draft of the spec) Specifies the user that authorized the token.

`token_type`

Specifies the type of token.

`exp`

Specifies the token expiration time in seconds since January 1 1970 UTC.

`expires_in`

Specifies the time, in seconds, that the token is valid for. This value is set at token creation time, and it depends on the configuration of the OAuth2 Provider Service.

During the introspection call, AM calculates the amount of seconds the token is still valid for and returns it in the `expires_in` object. Therefore, repeated calls to the endpoints return different values for the object.

However, the actual value of the `expires_in` object in the token does not change. Inspecting the token without using AM will show the value set at token creation time.

AM only returns this object for client-based tokens issued to a client configured in the same realm than that of the resource owner.

`sub`

Specifies the identifier of the identity or the OAuth 2.0/OpenID Connect client, that is the subject of the access token.

The subject claim is in the format `(type!subject)`, where:

- `subject` is the identifier of the user/identity, or the name of the OAuth 2.0/OpenID Connect client that is the subject of the token.
- `type` can be one of the following:
  - `age`. Specifies that the *subject* is an OAuth 2.0/OpenID Connect-related user-agent or client. For example, an OAuth 2.0 client, a Remote Consent Service agent, and a Web and Java Agent internal client.
  - `usr`. Specifies that the *subject* is a user/identity.

For example, `(usr!demo)`, or `(age!myOAuth2Client)`.

`iss`

Specifies the issuer of the token.

`cnf`

Specifies the confirmation key claim containing one of the following key types:

- **jwk**, which contains the decoded JSON web key (JWK) associated with the access token. For more information, see the "JWK-Based Proof-of-Possession" flow.
- **x5t#S256**, which contains the base64-encoded SHA-256 hash of the DER-encoding of a full X.509 certificate associated with the access token. For more information, see the "Certificate-Bound Proof-of-Possession" flow.

`macaroon`

Specifies the macaroon validated by the token, including any caveats appended to the macaroon.

`auth_level`

(AM-specific extension property) Specifies the authentication level of the resource owner that authenticated to authorize the token.

`permissions`

(UMA only). Specifies an array that contains the RPT token expiration time (exp), the resource scopes of the token, and the resource ID.

/json/token/macaroon

AM's macaroon endpoint can be used to inspect and manipulate macaroons. Macaroons are designed to be manipulated locally using a Macaroon library. This can be done securely by anybody in possession of the Macaroon token without needing access to any keys at all.

AM's macaroon endpoint supports two actions:

- `inspect`: returns details about the macaroon.
- `restrict`: adds a new caveat to the macaroon, returning a new macaroon.

You must compose the path to the introspect endpoint addressing the specific realm where the token was issued. For example, `https://openam.example.com:8443/openam/json/realms/root/realms/alpha/token/macaroon/?_action=inspect`.

The following example shows AM returning macaroon information:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "cache-control: no-cache" \
--data '{
    "macaroon":"AgEAAhtCRDFJSGhhLUktU21VbE5EQ0Y4MXVQRmlzUFUAAAYgnKhrEUFDwEwSPeTHwRSwTss7a4V0W68nL5Xw-
nnRhzQ"
    }' \
"https://openam.example.com:8443/openam/json/realms/root/realms/alpha/token/macaroon?_action=inspect"
   {
  "identifier": "1bmn1TQTONczbom-V2lCpaH4BUk",
  "location": "",
  "caveats": [
      {
        "type": "first-party",
        "identifier": {
         "scope": "openid profile"
        }
      }
  ],
   "signature": "kmVBwqpoi4nwakksk3b8KcSZvlYYmjCPdUTrFKFnhEY"
}
```

Tip

OpenID Connect clients must ensure that the following information is present in the JSON:

- The `openid` scope. For example, `"scopes": ["profile", "openid"]`.
- The `id_token` response type. For example, `"response_types": ["code", "id_token code"]`.

Legacy OAuth 2.0 endpoints

AM exposes the following *legacy* endpoints:

OAuth 2.0 Administration Endpoints

Endpoint	Description
`/frrest/oauth2/token (Legacy)`	Retrieve metadata about a token, revoke access and refresh tokens (AM-specific endpoint, legacy)
`/oauth2/tokeninfo (Legacy)`	Validate tokens and retrieve token metadata, such as scopes, to determine how to respond to requests for protected resources (AM-specific endpoint, legacy)

/frrest/oauth2/token (Legacy)

This AM-specific OAuth 2.0 token administration endpoint `/frrest/oauth2/token` lets you read, list, and delete (revoke) OAuth 2.0 tokens. OAuth 2.0 clients can also manage their own tokens.

Important

The `/frrest/oauth2/token` endpoint is labeled as *legacy* and it does not work with client-based OAuth 2.0 tokens.

Use the following endpoints instead:

- `/oauth2/introspect`. Use this endpoint to retrieve metadata from OAuth 2.0 tokens.
- `/oauth2/token/revoke`. Use this endpoint to delete (revoke) specific OAuth 2.0 tokens.
- `/users/user/oauth2/applications`. Use this endpoint to list clients that hold tokens granted by specific resource owners, and to delete tokens for resource owners and clients.

To list the contents of a specific token, send an HTTP GET request to `/frrest/oauth2/token/token-id` as follows:

```
$ curl --request POST \
--data "grant_type=password" \
--data "username=demo" \
--data "password=Ch4ng31t" \
--data "scope=cn" \
```

```
--data "client_id=myClient" \
--data "client_secret=forgerock" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token"
{
    "scope": "cn",
    "expires_in": 60,
    "token_type": "Bearer",
    "access_token": "f5fb4833-ba3d-41c8-bba4-833b49c3fe2c"
}


$ curl \
--request GET \
--header "iplanetDirectoryPro: AQIC5wM2LY4Sfcxs...EwNDU2NjE0*" \
https://openam.example.com:8443/openam/frrest/oauth2/token/f5fb4833-ba3d-41c8-bba4-833b49c3fe2c
{
    "expireTime": [
        "1418818601396"
    ],
    "tokenName": [
        "access_token"
    ],
    "scope": [
        "cn"
    ],
    "grant_type": [
        "password"
    ],
    "clientID": [
        "myClientID"
    ],
    "parent": [],
    "id": [
        "f5fb4833-ba3d-41c8-bba4-833b49c3fe2c"
    ],
    "tokenType": [
        "Bearer"
    ],
    "redirectURI": [],
    "nonce": [],
    "realm": [
        "/alpha"
    ],
    "userName": [
        "demo"
    ]
}
```

To list the tokens for the current user, send an HTTP GET request to `/frrest/oauth2/token/?_queryId=access_token`, including in the SSO token of the current user in a header. The following example shows a search for the demo user's access tokens:

```
$ curl \
 --request GET \
 --header "iplanetDirectoryPro: AQIC5wM2LY4Sfcw..." \
 "https://openam.example.com:8443/openam/frrest/oauth2/token/?_queryId=access_token"
{
    "result": [
```

```
    {
      "_rev": "1753454107",
      "tokenName": [
        "access_token"
      ],
      "expireTime": "Indefinitely",
      "scope": [
        "openid"
      ],
      "grant_type": [
        "password"
      ],
      "clientID": [
        "myClientID"
      ],
      "tokenType": [
        "Bearer"
      ],
      "redirectURI": [],
      "nonce": [],
      "realm": [
        "/alpha"
      ],
      "userName": [
        "bjensen"
      ],
      "display_name": "",
      "scopes": "openid"
    },
    {
      "_rev": "1753454107",
      "tokenName": [
        "access_token"
      ],
      "expireTime": "Indefinitely",
      "scope": [
        "openid"
      ],
      "grant_type": [
        "password"
      ],
      "clientID": [
        "myClientID"
      ],
      "tokenType": [
        "Bearer"
      ],
      "redirectURI": [],
      "nonce": [],
      "realm": [
        "/alpha"
      ],
      "userName": [
        "bjensen"
      ],
      "display_name": "",
      "scopes": "openid"
    }
  ],
```

```
  "resultCount": 2,
  "pagedResultsCookie": null,
  "totalPagedResultsPolicy": "NONE",
  "totalPagedResults": -1,
  "remainingPagedResults": -1
}
```

To list the tokens for a specific user tokens, send an HTTP GET request to `/frrest/oauth2/token/?_queryId=userName=`*`string`*, where *string* is the user, such as `bjensen`, and *realm* is the subrealm in which the user is located. You do not have to include the `realm` parameter if the user is in the top-level realm.

Include the SSO token of an *administrative* user, such as `amAdmin`, in a header. For example:

```
$ curl \
  --request GET \
  --header "iplanetDirectoryPro: AQIC5wM2LY4Sfcxs...EwNDU2NjE0*" \
  "https://openam.example.com:8443/openam/frrest/oauth2/token/?_queryId=userName=bjensen,realm=alpha"
{
    "result": [
    {
      "_id": "2aaddde8-586b-4cb7-b431-eb86af57aabc",
      "_rev": "549186065",
      "tokenName": [
        "access_token"
      ],
      "expireTime": "Indefinitely",
      "scope": [
        "openid"
      ],
      "grant_type": [
        "password"
      ],
      "authGrantId": [
        "50e9f80b-d193-4aeb-93e9-e383ea2cabd3"
      ],
      "clientID": [
        "myClientID"
      ],
      "parent": [],
      "refreshToken": [
        "5e1423a2-d2cd-40d5-8f54-5b695836cd44"
      ],
      "id": [
        "2aaddde8-586b-4cb7-b431-eb86af57aabc"
      ],
      "tokenType": [
        "Bearer"
      ],
      "auditTrackingId": [
        "6ac90d13-9cac-444b-bfbc-c7aca16713de-777"
      ],
      "redirectURI": [],
      "nonce": [],
      "realm": [
        "/alpha"
      ],
      "userName": [
        "bjensen"
```

```
      ],
      "display_name": "",
      "scopes": "openid"
    },
    {
      "_id": "5e1423a2-d2cd-40d5-8f54-5b695836cd44",
      "_rev": "1171292923",
      "tokenName": [
        "refresh_token"
      ],
      "expireTime": "Oct 18, 2016 10:51 AM",
      "scope": [
        "openid"
      ],
      "grant_type": [
        "password"
      ],
      "authGrantId": [
        "50e9f80b-d193-4aeb-93e9-e383ea2cabd3"
      ],
      "clientID": [
        "myClientID"
      ],
      "authModules": [],
      "id": [
        "5e1423a2-d2cd-40d5-8f54-5b695836cd44"
      ],
      "tokenType": [
        "Bearer"
      ],
      "auditTrackingId": [
        "6ac90d13-9cac-444b-bfbc-c7aca16713de-776"
      ],
      "redirectURI": [],
      "realm": [
        "/alpha"
      ],
      "userName": [
        "bjensen"
      ],
      "acr": [],
      "display_name": "",
      "scopes": "openid"
    },
  ],
  "resultCount": 2,
  "pagedResultsCookie": null,
  "totalPagedResultsPolicy": "NONE",
  "totalPagedResults": -1,
  "remainingPagedResults": -1
}
```


To delete (revoke) a token, perform an HTTP DELETE on `/frrest/oauth2/token/token-id`, including the SSO token of an administrative user, such as `amAdmin`, as in the following example:

```
$ curl --request POST \
--data "grant_type=password" \
--data "username=demo" \
--data "password=Ch4ng31t" \
--data "scope=cn" \
--data "client_id=myClient" \
--data "client_secret=forgerock" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token"
{
    "scope": "cn",
    "expires_in": 60,
    "token_type": "Bearer",
    "access_token": "f5fb4833-ba3d-41c8-bba4-833b49c3fe2c"
}


$ curl \
 --request DELETE \
 --header "iplanetDirectoryPro: AQIC5wM2LY4Sfcxs...EwNDU2NjE0*" \
 "https://openam.example.com:8443/openam/frrest/oauth2/token/f5fb4833-ba3d-41c8-bba4-833b49c3fe2c"
{
    "success": "true"
}
```

/oauth2/tokeninfo (Legacy)

AM-specific endpoint used to validate tokens and to retrieve information out of them, such as scopes, the grant type used when issuing the token, or the token expiration time.

Tip

The `/frrest/oauth2/tokeninfo` endpoint is labeled as *legacy*.

To validate tokens and retrieve information with a spec-based endpoint, see /oauth2/introspect.

Resource servers —or any party having the token ID— can obtain token information through this endpoint without authenticating.

The token information endpoint supports the following query parameter:

`access_token`

Specifies the token ID.

Required: Yes.

The following example shows AM issuing an access token, and then returning token information:

```
$ curl --request POST \
--data "grant_type=password" \
--data "username=demo" \
--data "password=Ch4ng31t" \
```

```
--data "scope=write" \
--data "client_id=myClient" \
--data "client_secret=forgerock" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token"
{
  "access_token": "sbQZuveFumUDV5R1vVBl6QAGNB8",
  "scope": "write",
  "token_type": "Bearer",
  "expires_in": 3599
}

$ curl \
--request GET \
--header "Authorization: Bearer sbQZuveFumUDV5R1vVBl6QAGNB8" \
"https://openam.example.com:8443/openam/oauth2/tokeninfo"
{
   "access_token":"sbQZuveFumUDV5R1vVBl6QAGNB8",
   "grant_type":"password",
   "auth_level":0,
   "scope":[
      "write"
   ],
   "realm":"/alpha",
   "token_type":"Bearer",
   "expires_in":2491,
   "write":"",
   "client_id":"myClient"
}
```

Note that AM returns a JSON object with the following properties:

`access_token`

Specifies the token ID.

`grant_type`

Specifies the OAuth 2.0 grant flow used to issue the token.

`auth_level`

Specifies the authentication level of the resource owner that authenticated to authorize the token.

`scope`

Specifies a JSON structure containing the scopes associated with the token.

`realm`

Specifies the realm from which the token was obtained.

`token_type`

Specifies the type of token.

`expires_in`

Specifies the time, in seconds, that the token is valid for. This value is set at token creation time, and it depends on the configuration of the OAuth2 Provider Service.

During the introspection call, AM calculates the amount of seconds the token is still valid for and returns it in the `expires_in` object. Therefore, repeated calls to the endpoints return different values for the object.

However, the actual value of the `expires_in` object in the token does not change. Inspecting the token without using AM will show the value set at token creation time.

AM does not return this object for client-based tokens issued to a client configured in a different realm that the resource owner's.

`client_id`

Specifies the client that requested the token.

Chapter 15

OAuth 2.0 Administration and Supporting REST Endpoints

AM exposes the following administration and supporting REST endpoints:

OAuth 2.0 Administration and Supporting Endpoints

Endpoint	Description
`/realm-config/agents/OAuth2Client`	Register, list, and delete OAuth 2.0 clients (AM specific-endpoint)
`/users/user/oauth2/resources/sets`	Retrieve data for UMA resources registered to a particular user (AM-specific endpoint)
`/users/user/oauth2/applications`	List Oauth 2.0 clients holding active tokens granted by specific resource owners, and delete tokens for a combination of resource owner and client (AM-specific endpoint)

/realm-config/agents/OAuth2Client

AM-specific endpoint that lets AM and agent administrators create, list, and delete OAuth 2.0 clients.

Tip

Use the AM API Explorer for detailed information about the parameters supported by this endpoint, and to test it against your deployed AM instance.

In the AM console, select the Help icon, and then go to API Explorer > /realm-config > /agents > /OAuth2Client.

The following example shows how to create a basic OAuth 2.0 client named `myClient` in a realm named `alpha`. Note that you must provide the SSO token of an administrative user as a header, and that the name of the client is appended to the URL:

```
$ curl \
 --request PUT \
 --header "Accept-API-Version: resource=1.0" \
 --header "Content-Type: application/json" \
 --header "Accept: application/json" \
 --header "iplanetDirectoryPro: AQIC5wM...3MTYxOA..*" \
--data '{
   "coreOAuth2ClientConfig":{
      "agentgroup":"",
```

```
      "status":{
         "inherited":true,
         "value":"string"
      },
      "userpassword":"forgerock",
      "clientType":{
         "inherited":false,
         "value":"Confidential"
      },
      "redirectionUris":{
         "inherited":false,
         "value":[
            "https://www.example.com:443/callback"
         ]
      },
      "scopes":{
         "inherited":false,
         "value":[
            "write",
            "read"
         ]
      },
      "defaultScopes":{
         "inherited":true,
         "value":[
            "write"
         ]
      },
      "clientName":{
         "inherited":true,
         "value":[
            "My Test Client"
         ]
      }
   },
   "advancedOAuth2ClientConfig":{
      "name":{
         "inherited":false,
         "value":[
            null
         ]
      },
      "grantTypes":{
         "inherited":true,
         "value":[
            "authorization_code",
            "client_credentials"
         ]
      },
      "tokenEndpointAuthMethod":{
         "inherited":true,
         "value":"client_secret_basic"
      }
   }
}' \
"https://openam.example.com:8443/openam/json/realms/root/realms/alpha/realm-config/agents/OAuth2Client/
testClient"
{
   "_id":"testClient",
```

```
   "_rev":"-60716879",
   "advancedOAuth2ClientConfig":{
      "descriptions":{
         "inherited":false,
         "value":[

         ]
      },

...

      "clientType":{
         "inherited":false,
         "value":"Confidential"
      },
...
      "_type":{
      "_id":"OAuth2Client",
      "name":"OAuth2 Clients",
      "collection":true
   }
}
```

The following example shows how to delete an OAuth 2.0 client named `myClient` in a realm named `alpha`. Note that you must provide the SSO token of an administrative user as a header, and that the name of the client is appended to the URL:

```
$ curl \
    --request DELETE \
    --header "Accept-API-Version: resource=1.0" \
    --header "iplanetDirectoryPro: AQIC5wM...3MTYxOA..*" \
    "https://openam.example.com:8443/openam/json/realms/root/realms/alpha/realm-config/agents/
OAuth2Client/myClient"
{
   "_id":"testClient",
   "_rev":"-60716879",
   "advancedOAuth2ClientConfig":{
      "descriptions":{
         "inherited":false,
         "value":[

         ]
      },

...

      "clientType":{
         "inherited":false,
         "value":"Confidential"
      },
...
      "_type":{
      "_id":"OAuth2Client",
      "name":"OAuth2 Clients",
      "collection":true
   }
}
```

You can use a similar PUT command to the one above to update an existing OAuth 2.0 client. However, ensure that you include all the attributes to be retained. Omitting an attribute in the resource amounts to deleting the attribute.

/users/user/oauth2/resources/sets

AM-specific endpoint for viewing and updating a resource registered to a particular user.

Tip

Use the AM API Explorer for detailed information about the parameters supported by this endpoint, and to test it against your deployed AM instance.

In the AM console, click the Help icon, and go to API Explorer > /users > /{user} > /oauth2 > /resources > /sets.

To call the endpoint, you must compose the path to the realm where the resource is registered.

This example reads an OAuth 2.0 resource and related policy in the `alpha` realm. Note that you must provide the SSO token of an administrative user or of the resource owner as a header, and that the name of the resource owner (`demo`, in this example) is part of the URL:

```
$ curl \
--request GET \
--header "iPlanetDirectoryPro: AQIC5wM2LY4Sfcxs...EwNDU2NjE0*" \
"https://openam.example.com:8443/openam/json/realms/root/realms/alpha/users/demo\
/oauth2/resources/sets/43225628-4c5b-4206-b7cc-5164da81decd0"
{
    "scopes": [
         "http://photoz.example.com/dev/scopes/view",
         "http://photoz.example.com/dev/scopes/comment"
    ],
    "_id": "43225628-4c5b-4206-b7cc-5164da81decd0",
    "resourceServer": "UMA-Resource-Server",
    "name": "My Videos",
    "icon_uri": "http://www.example.com/icons/cinema.png",
    "policy": {
        "permissions": [
            {
                "subject": "user.1",
                "scopes": [
                    "http://photoz.example.com/dev/scopes/view"
                ]
            },
            {
                "subject": "user.2",
                "scopes": [
                    "http://photoz.example.com/dev/scopes/comment",
                    "http://photoz.example.com/dev/scopes/view"
                ]
            }
        ]
    },
    "type": "http://www.example.com/rsets/videos"
```

```
}
```

Tip

You can specify the fields that are returned with the `_fields` query string filter. For example `?_fields=scopes, resourceServer, name`

On success, an HTTP 200 OK status code is returned, with a JSON body representing the resource. If a policy relating to the resource exists, a representation of the policy is also returned in the JSON.

If the specified resource does not exist, an HTTP 404 Not Found status code is returned, as follows:

```
{
    "code": 404,
    "reason": "Not Found",
    "message": "No resource set with id, bad-id-3e28-4c19-8a2b-36fc24c899df0, found."
}
```

If the SSO token used is not that of the resource owner or an administrator, an HTTP 403 Forbidden status code is returned, as follows:

```
{
    "code": 403,
    "reason": "Forbidden",
    "message": "User, user.1, not authorized."
}
```

/users/user/oauth2/applications

AM-specific endpoint for listing clients holding tokens granted by specific resource owners, and for deleting tokens for a combination of a resource owner and client.

Tip

Use the AM API Explorer for detailed information about the parameters supported by this endpoint, and to test it against your deployed AM instance.

In the AM console, select the Help icon, and then go to API Explorer > /users > /{user} > /oauth2 > /applications.

To call the endpoint, you must compose the path to the realm where the client is registered.

This example lists all the clients holding tokens granted in the `alpha` realm by the `demo` user. Note that you must provide the SSO token of an administrative user or of the resource owner as a header, and that the name of the resource owner (`demo`) is part of the URL:

```
$ curl --request GET \
--header "Accept-API-Version: resource=1.1" \
--header "iplanetDirectoryPro: Ua6fsH2vjgHqVY..." \
"https://openam.example.com:8443/openam/json/realms/root/realms/alpha/users/demo/oauth2/applications?
_queryFilter=true"
```

On success, AM returns an HTTP 200 code and a JSON structure containing information about the tokens, such as the client ID they belong to, the scopes they grants, and their expiration time:

```
{
   "result":[
      {
         "_id":"myClient",
         "_rev":"22274676",
         "name":null,
         "description":"This field describes myClient",
         "scopes":{
            "write":"write"
         },
         "expiryDateTime":"2018-11-14T10:48:55.395Z",
         "logoUri":null
      }
   ],
   "resultCount":1,
   "pagedResultsCookie":null,
   "totalPagedResultsPolicy":"NONE",
   "totalPagedResults":-1,
   "remainingPagedResults":-1
```

The following example shows how to delete all tokens held by the client `myClient` granted in the `alpha` realm by the `demo` user. Note that you must provide the SSO token of an administrative user or of the resource owner as a header, and that the name of the resource owner (`demo`) and the name of the client (`myClient`) are part of the URL:

```
$ curl --request DELETE \
--header "Accept-API-Version: resource=1.1" \
--header "iplanetDirectoryPro: Ua6fsH2vjgHqVY..." \
"https://openam.example.com:8443/openam/json/realms/root/realms/alpha/users/demo/oauth2/applications/
myClient"
```

On success, AM returns an HTTP 200 code and a JSON structure containing information about the deleted tokens, such as the client ID they belonged to, the scopes they granted, and their expiration time:

```
{
  "_id": "myClient",
  "_rev": "22274676",
  "name": null,
  "description":"This field describes myClient",
  "scopes": {
    "write": "write"
  },
  "expiryDateTime": "2018-11-14T10:48:55.395Z",
  "logoUri": null
}
```

Chapter 16

Modifying the Content of Access Tokens

You can modify the key-value pairs contained within an OAuth 2.0 access token by using a script. For example, you could make a REST call to an external service, and add or change a key-value pair in the access token based on the response, before issuing the token to the resource owner.

Modification works for client-based and CTS-based access tokens, and are stored permanently in either the issued JWT or in the CTS respectively. It also works when macaroons are used in place of regular tokens, in which case, you can use scripts to both modify the key pairs in the token and/or to add caveats.

Use access token modification scripts with the OAuth 2.0 default scope validator class.

AM includes an example script that demonstrates some of the functionality available. To examine the contents of the example access token modification script, in the AM console, go to Realms > Top Level Realm > Scripts, and then select OAuth2 Access Token Modification Script.

For general information about scripting in AM, see Getting Started with Scripting.

For information about the API available for modifying access tokens with scripts, see the following:

- "Accessing HTTP Services" in the *Getting Started with Scripting*
- "Debug Logging" in the *Getting Started with Scripting*
- "AccessToken" interface in the *AM 7.1.4 Public API Javadoc*
- "AMIdentity" interface in the *AM 7.1.4 Public API Javadoc*
- "SSOToken" interface in the *AM 7.1.4 Public API Javadoc*
- "MacaroonToken" interface in the *AM 7.1.4 Public API Javadoc*

When issuing modified access tokens, consider the following important points:

- Removing or changing native properties may render the access token unreadable

  AM requires that certain native properties are present in the access token in order to consider it valid. Removing or modifying these properties may cause the OAuth 2.0 flows to break.

Tip

Native properties are marked in the AM 7.1.4 Public API Javadoc with a warning about loss of functionality if they are edited or removed.

- Modifying access tokens affects the size of the client-based token or CTS entry

  Changes made to OAuth 2.0 access tokens directly impacts the size of the CTS tokens when using CTS-based tokens, or the size of the JSON web tokens (JWT) if client-based is enabled.

  You must ensure that the token size remains within your client or user-agent size limits.

  For more information, see "About Token Storage Location".

Preparing AM to Modify Access Tokens

AM requires a small amount of configuration before trying the default access token modification script. The script requires that the authenticated user of the access token has an email address and telephone number in their profile. The script adds the values of these fields to the access token.

Perform the steps in the following procedures to prepare AM for testing scripted modification of OAuth 2.0 access tokens:

- "To Add an Email Address and Telephone Number to the Demo User"
- "To Modify the Default Access Token Modification Script"
- "To Configure AM to Issue Access Tokens Using the Default Access Token Modification Script"

To Add an Email Address and Telephone Number to the Demo User

In this procedure, add an email address and telephone number value to the `demo` user's profile. The access token modification script injects the values provided into the OAuth 2.0 access token before it is issued to the resource owner.

1. Log in as an AM administrator. For example `amAdmin`.
2. Select Realms > Top Level Realm > Identities.
3. On the Identities tab, select the `demo` user.
   a. In Email Address, enter a valid address. For example:
      ```
      demo.user@example.com
      ```
   b. In Telephone Number, enter a value. For example:
      ```
      +44 117 496 0228
      ```

4. Select Save Changes.

To Modify the Default Access Token Modification Script

In this procedure, uncomment functionality in the default access token modification script in order to demonstrate how to modify access tokens.

1. Log in as an AM administrator. For example `amAdmin`.
2. Navigate to Realms > Top Level Realm > Scripts, and then click OAuth2 Access Token Modification Script.
3. In the Script field:
   a. Uncomment line 34 of the script, by surrounding the line with a pair of `*/` and `/*` strings:
   ```
   */
   accessToken.setField("hello", "world")
   /*
   ```
   b. Uncomment lines 59 to 61 of the script, by surrounding them with a pair of `*/` and `/*` strings:
   ```
   */
   def attributes = identity.getAttributes(["mail", "telephoneNumber"].toSet())
   accessToken.setField("mail", attributes["mail"])
   accessToken.setField("phone", attributes["telephoneNumber"])
   /*
   ```
4. Select Save Changes.

To Configure AM to Issue Access Tokens Using the Default Access Token Modification Script

In this procedure, create an OAuth 2.0 provider that uses the default access token modification script, as well as an OAuth 2.0 client. Obtaining an access token as the `demo` user can then be performed to test the script functionality.

1. Log in as an AM administrator. For example `amAdmin`.
2. Create an OAuth 2.0 provider by performing the following steps:
   a. Navigate to Realms > Top Level Realm > Configure OAuth Provider, and then click Configure OAuth 2.0.
   b. Keep the suggested settings, click Create, and then click OK.

      The default setting for a new OAuth 2.0 provider is to use the default access token modification script.

      For information on OAuth 2.0 provider properties, see "OAuth2 Provider" in the *Reference*.
3. Create an OAuth 2.0 client by performing the following steps:

a. Navigate to Realms > Top Level Realm > Applications > OAuth 2.0 > Clients, and then click Add Client.

b. Enter the following values:

- **Client ID**: `myClient`
- **Client secret**: `forgerock`
- **Redirection URIs**: `https://www.example.com:443/callback`
- **Scope(s)**: `access|Access to your data`

c. Click Create.

AM is now configured to issue access tokens using the default access token modification script.

Trying the Default Access Token Modification Script

This section demonstrates obtaining an OAuth 2.0 access token which has been modified by a script.

First, we will use the Authorization Code Grant flow to authenticate with AM as the resource owner, allow the client to access our profile data, and receive the authorization code.

In the second procedure, we will exchange the authorization code for an access token, which will have been altered by the default access token modification script to include:

- The resource owner's telephone number and email address, taken from their profile in AM, which is acting as the authorization server.
- A `hello:world` key-value pair.

In the final procedure, we will introspect the access token to verify that it does include the modified values.

To Obtain an Authorization Code to Test Access Token Modification

1. In a web browser, navigate to the /oauth2/authorize endpoint, including the parameters and values configured for the OAuth 2.0 client in the previous section.

   For example:

   ```
   https://openam.example.com:8443/openam/oauth2/realms/root/authorize?
   client_id=myClient&response_type=code&scope=access&state=abc123&redirect_uri=https://
   www.example.com:443/callback
   ```

   The AM sign in page is displayed.

2. Log in as the `demo` user, with password `Ch4ng31t`.

   The AM OAuth 2.0 consent page is displayed.

3. Review the scopes being requested, and then click Allow.

   AM redirects the browser to the location specified in the `redirect_uri` parameter, `https://www.example.com:443/callback` in this example, and appends a number of query parameters. For example:

4. Record the value of the `code` query parameter.

   This is the authorization code and is exchanged for an access token in the next procedure.

To Exchange an Authorization Code for an Access Token to Test Access Token Modification

1. Create a POST request to the /oauth2/access_token endpoint, including the authorization code obtained in the previous procedure, and the parameters and values configured for the OAuth 2.0 client earlier.

   For example:

```
$ curl --request POST \
  --data "grant_type=authorization_code" \
  --data "code=tH_s2obVRt2_yB6x4OxH1J3eMkU" \
  --data "client_id=myClient" \
  --data "client_secret=forgerock" \
  --data "redirect_uri=https://www.example.com:443/callback" \
  "https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token"
{
  "access_token": "sbQZuveFumUDV5R1vVBl6QAGNB8",
  "scope": "access",
  "token_type": "Bearer",
  "expires_in": 3599
}
```

2. Record the value of the `access_token` property.

   This is the access token, the properties of which have been modified by the access token modification script. Follow the steps in the next procedure to introspect the token to verify the properties have been modified.

To Introspect an Access Token to Verify Access Token Modification

- Create a POST request to the /oauth2/introspect endpoint, including the access token obtained in the previous procedure, and the credentials of the OAuth 2.0 client earlier.

  For example:

```
$ curl \
  --request POST \
  --data "client_id=myClient" \
  --data "client_secret=forgerock" \
  --data "token=sbQZuveFumUDV5R1vVBl6QAGNB8" \
  "https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/introspect"
{
  "active": true,
  "scope": "access",
  "client_id": "myClient",
  "user_id": "demo",
  "token_type": "Bearer",
  "exp": 1556289970,
  "sub": "(usr!demo)",
  "subname": "demo",
  "iss": "https://openam.example.com:8443/openam/oauth2",
  "auth_level": 0,
  "auditTrackingId": "c6e22be7-6166-402b-9d72-a03134f08c22-8605",
  "hello": "world",
  "mail": [
    "demo.user@example.com"
  ],
  "phone": [
    "+44 117 496 0228"
  ]
}
```

Notice that the output includes a `hello:world` key-value pair, as well as `mail` and `phone` properties, containing values taken from the user's profile data.

OAuth 2.0 Access Token Modification Scripting API

The following properties are available to scripts:

`clientProperties`

A map of properties configured in the relevant client profile. Only present if the client was correctly identified.

The keys in the map are as follows:

`clientId`

The URI of the client.

`allowedGrantTypes`

The list of the allowed grant types (`org.forgerock.oauth2.core.GrantType`) for the client.

`allowedResponseTypes`

The list of the allowed response types for the client.

`allowedScopes`

The list of the allowed scopes for the client.

`customProperties`

A map of any custom properties added to the client.

Lists or maps are included as sub-maps.

For example, a custom property of

`customMap[Key1]=Value1`

is returned as

`customMap` > `Key1` > `Value1`.

To add custom properties to a client, go to OAuth 2.0 > Clients > *Client ID* > Advanced, and update the Custom Properties field. The custom properties can be added in the format shown in these examples:

```
customproperty=custom-value1
customList[0]=customList-value-0
customList[1]=customList-value-1
customMap[key1]=customMap-value-1
customMap[key2]=customMap-value-2
```

From within the script, you can then access the custom properties in the following way:

```
var customProperties = clientProperties.get("customProperties");
var property = customProperties.get(PROPERTY_KEY);
```

`requestProperties`

A map of the properties present in the request. Always present.

The keys in the map are as follows:

`requestUri`

The URI of the request.

`realm`

The realm to which the request was made.

`requestParams`

The request parameters, and/or posted data. Each value in this map is a list of one, or more, properties.

> **Important**
>
> To mitigate the risk of reflection-type attacks, use OWASP best practices when handling these properties. For example, see Unsafe use of Reflection.

`scopes`

Contains a set of the requested scopes. For example:

```
[
  "read",
  "transfer",
  "download"
]
```

`scriptName`

The display name of the script. Always present.

Glossary

Access control	Control to grant or to deny access to a resource.
Account lockout	The act of making an account temporarily or permanently inactive after successive authentication failures.
Actions	Defined as part of policies, these verbs indicate what authorized identities can do to resources.
Advice	In the context of a policy decision denying access, a hint to the policy enforcement point about remedial action to take that could result in a decision allowing access.
Agent administrator	User having privileges only to read and write agent profile configuration information, typically created to delegate agent profile creation to the user installing a web or Java agent.
Agent authenticator	Entity with read-only access to multiple agent profiles defined in the same realm; allows an agent to read web service profiles.
Application	In general terms, a service exposing protected resources. In the context of AM policies, the application is a template that constrains the policies that govern access to protected resources. An application can have zero or more policies.
Application type	Application types act as templates for creating policy applications. Application types define a preset list of actions and functional logic, such as policy lookup and resource comparator logic.

	Application types also define the internal normalization, indexing logic, and comparator logic for applications.
Attribute-based access control (ABAC)	Access control that is based on attributes of a user, such as how old a user is or whether the user is a paying customer.
Authentication	The act of confirming the identity of a principal.
Authentication chaining	A series of authentication modules configured together which a principal must negotiate as configured in order to authenticate successfully.
Authentication level	Positive integer associated with an authentication module, usually used to require success with more stringent authentication measures when requesting resources requiring special protection.
Authentication module	AM authentication unit that handles one way of obtaining and verifying credentials.
Authorization	The act of determining whether to grant or to deny a principal access to a resource.
Authorization Server	In OAuth 2.0, issues access tokens to the client after authenticating a resource owner and confirming that the owner authorizes the client to access the protected resource. AM can play this role in the OAuth 2.0 authorization framework.
Auto-federation	Arrangement to federate a principal's identity automatically based on a common attribute value shared across the principal's profiles at different providers.
Bulk federation	Batch job permanently federating user profiles between a service provider and an identity provider based on a list of matched user identifiers that exist on both providers.
Circle of trust	Group of providers, including at least one identity provider, who have agreed to trust each other to participate in a SAML v2.0 provider federation.
Client	In OAuth 2.0, requests protected web resources on behalf of the resource owner given the owner's authorization. AM can play this role in the OAuth 2.0 authorization framework.
Client-based OAuth 2.0 tokens	After a successful OAuth 2.0 grant flow, AM returns a token to the client. This differs from CTS-based OAuth 2.0 tokens, where AM returns a *reference* to token to the client.
Client-based sessions	AM sessions for which AM returns session state to the client after each request, and require it to be passed in with the subsequent

	request. For browser-based clients, AM sets a cookie in the browser that contains the session information. For browser-based clients, AM sets a cookie in the browser that contains the session state. When the browser transmits the cookie back to AM, AM decodes the session state from the cookie.
Conditions	Defined as part of policies, these determine the circumstances under which which a policy applies. Environmental conditions reflect circumstances like the client IP address, time of day, how the subject authenticated, or the authentication level achieved. Subject conditions reflect characteristics of the subject like whether the subject authenticated, the identity of the subject, or claims in the subject's JWT.
Configuration datastore	LDAP directory service holding AM configuration data.
Cross-domain single sign-on (CDSSO)	AM capability allowing single sign-on across different DNS domains.
CTS-based OAuth 2.0 tokens	After a successful OAuth 2.0 grant flow, AM returns a *reference* to the token to the client, rather than the token itself. This differs from client-based OAuth 2.0 tokens, where AM returns the entire token to the client.
CTS-based sessions	AM sessions that reside in the Core Token Service's token store. CTS-based sessions might also be cached in memory on one or more AM servers. AM tracks these sessions in order to handle events like logout and timeout, to permit session constraints, and to notify applications involved in SSO when a session ends.
Delegation	Granting users administrative privileges with AM.
Entitlement	Decision that defines which resource names can and cannot be accessed for a given identity in the context of a particular application, which actions are allowed and which are denied, and any related advice and attributes.
Extended metadata	Federation configuration information specific to AM.
Extensible Access Control Markup Language (XACML)	Standard, XML-based access control policy language, including a processing model for making authorization decisions based on policies.
Federation	Standardized means for aggregating identities, sharing authentication and authorization data information between trusted providers, and

	allowing principals to access services across different providers without authenticating repeatedly.
Fedlet	Service provider application capable of participating in a circle of trust and allowing federation without installing all of AM on the service provider side; AM lets you create Java Fedlets.
Hot swappable	Refers to configuration properties for which changes can take effect without restarting the container where AM runs.
Identity	Set of data that uniquely describes a person or a thing such as a device or an application.
Identity federation	Linking of a principal's identity across multiple providers.
Identity provider (IDP)	Entity that produces assertions about a principal (such as how and when a principal authenticated, or that the principal's profile has a specified attribute value).
Identity repository	Data store holding user profiles and group information; different identity repositories can be defined for different realms.
Java agent	Java web application installed in a web container that acts as a policy enforcement point, filtering requests to other applications in the container with policies based on application resource URLs.
Metadata	Federation configuration information for a provider.
Policy	Set of rules that define who is granted access to a protected resource when, how, and under what conditions.
Policy agent	Java, web, or custom agent that intercepts requests for resources, directs principals to AM for authentication, and enforces policy decisions from AM.
Policy Administration Point (PAP)	Entity that manages and stores policy definitions.
Policy Decision Point (PDP)	Entity that evaluates access rights and then issues authorization decisions.
Policy Enforcement Point (PEP)	Entity that intercepts a request for a resource and then enforces policy decisions from a PDP.
Policy Information Point (PIP)	Entity that provides extra information, such as user profile attributes that a PDP needs in order to make a decision.
Principal	Represents an entity that has been authenticated (such as a user, a device, or an application), and thus is distinguished from other entities.

	When a Subject successfully authenticates, AM associates the Subject with the Principal.
Privilege	In the context of delegated administration, a set of administrative tasks that can be performed by specified identities in a given realm.
Provider federation	Agreement among providers to participate in a circle of trust.
Realm	AM unit for organizing configuration and identity information. Realms can be used for example when different parts of an organization have different applications and identity stores, and when different organizations use the same AM deployment. Administrators can delegate realm administration. The administrator assigns administrative privileges to users, allowing them to perform administrative tasks within the realm.
Resource	Something a user can access over the network such as a web page. Defined as part of policies, these can include wildcards in order to match multiple actual resources.
Resource owner	In OAuth 2.0, entity who can authorize access to protected web resources, such as an end user.
Resource server	In OAuth 2.0, server hosting protected web resources, capable of handling access tokens to respond to requests for such resources.
Response attributes	Defined as part of policies, these allow AM to return additional information in the form of "attributes" with the response to a policy decision.
Role based access control (RBAC)	Access control that is based on whether a user has been granted a set of permissions (a role).
Security Assertion Markup Language (SAML)	Standard, XML-based language for exchanging authentication and authorization data between identity providers and service providers.
Service provider (SP)	Entity that consumes assertions about a principal (and provides a service that the principal is trying to access).
Authentication Session	The interval while the user or entity is authenticating to AM.
Session	The interval that starts after the user has authenticated and ends when the user logs out, or when their session is terminated. For browser-based clients, AM manages user sessions across one or more applications by setting a session cookie. See also CTS-based sessions and Client-based sessions.

Session high availability	Capability that lets any AM server in a clustered deployment access shared, persistent information about users' sessions from the CTS token store. The user does not need to log in again unless the entire deployment goes down.
Session token	Unique identifier issued by AM after successful authentication. For a CTS-based sessions, the session token is used to track a principal's session.
Single log out (SLO)	Capability allowing a principal to end a session once, thereby ending her session across multiple applications.
Single sign-on (SSO)	Capability allowing a principal to authenticate once and gain access to multiple applications without authenticating again.
Site	Group of AM servers configured the same way, accessed through a load balancer layer. The load balancer handles failover to provide service-level availability. The load balancer can also be used to protect AM services.
Standard metadata	Standard federation configuration information that you can share with other access management software.
Stateless Service	Stateless services do not store any data locally to the service. When the service requires data to perform any action, it requests it from a data store. For example, a stateless authentication service stores session state for logged-in users in a database. This way, any server in the deployment can recover the session from the database and service requests for any user. All AM services are stateless unless otherwise specified. See also Client-based sessions and CTS-based sessions.
Subject	Entity that requests access to a resource When an identity successfully authenticates, AM associates the identity with the Principal that distinguishes it from other identities. An identity can be associated with multiple principals.
Identity store	Data storage service holding principals' profiles; underlying storage can be an LDAP directory service or a custom `IdRepo` implementation.
Web Agent	Native library installed in a web server that acts as a policy enforcement point with policies based on web page URLs.