



# OAuth 2.0 Guide

/ ForgeRock Access Management 6.5

Latest update: 6.5.5

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

---

Copyright © 2011-2022 ForgeRock AS.

## 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.



This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.

To view a copy of this license, visit <https://creativecommons.org/licenses/by-nc-nd/3.0/> or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

ForgeRock® and ForgeRock Identity Platform™ are trademarks of ForgeRock Inc. or its subsidiaries in the U.S. and in other countries. Trademarks are the property of their respective owners.

UNLESS OTHERWISE MUTUALLY AGREED BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

DejaVu Fonts

Bitstream Vera Fonts Copyright

Copyright (c) 2003 by Bitstream, Inc. All Rights Reserved. Bitstream Vera is a trademark of Bitstream, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of the fonts accompanying this license ("Fonts") and associated documentation files (the "Font Software"), to reproduce and distribute the Font Software, including without limitation the rights to use, copy, merge, publish, distribute, and/or sell copies of the Font Software, and to permit persons to whom the Font Software is furnished to do so, subject to the following conditions:

The above copyright and trademark notices and this permission notice shall be included in all copies of one or more of the Font Software typefaces.

The Font Software may be modified, altered, or added to, and in particular the designs of glyphs or characters in the Fonts may be modified and additional glyphs or characters may be added to the Fonts, only if the fonts are renamed to names not containing either the words "Bitstream" or the word "Vera".

This License becomes null and void to the extent applicable to Fonts or Font Software that has been modified and is distributed under the "Bitstream Vera" names.

The Font Software may be sold as part of a larger software package but no copy of one or more of the Font Software typefaces may be sold by itself.

THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL BITSTREAM OR THE GNOME FOUNDATION BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.

Except as contained in this notice, the names of Gnome, the Gnome Foundation, and Bitstream Inc., shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Font Software without prior written authorization from the Gnome Foundation or Bitstream Inc., respectively. For further information, contact: fonts at gnome dot org.

Arev Fonts Copyright

Copyright (c) 2006 by Tavmjong Bah. All Rights Reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of the fonts accompanying this license ("Fonts") and associated documentation files (the "Font Software"), to reproduce and distribute the modifications to the Bitstream Vera Font Software, including without limitation the rights to use, copy, merge, publish, distribute, and/or sell copies of the Font Software, and to permit persons to whom the Font Software is furnished to do so, subject to the following conditions:

The above copyright and trademark notices and this permission notice shall be included in all copies of one or more of the Font Software typefaces.

The Font Software may be modified, altered, or added to, and in particular the designs of glyphs or characters in the Fonts may be modified and additional glyphs or characters may be added to the Fonts, only if the fonts are renamed to names not containing either the words "Tavmjong Bah" or the word "Arev".

This License becomes null and void to the extent applicable to Fonts or Font Software that has been modified and is distributed under the "Tavmjong Bah Arev" names.

The Font Software may be sold as part of a larger software package but no copy of one or more of the Font Software typefaces may be sold by itself.

THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL TAVMJONG BAH BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.

Except as contained in this notice, the name of Tavmjong Bah shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Font Software without prior written authorization from Tavmjong Bah. For further information, contact: tavmjong @ free . fr.

FontAwesome Copyright

Copyright (c) 2017 by Dave Gandy, <https://fontawesome.com/>.

This Font Software is licensed under the SIL Open Font License, Version 1.1. See <https://opensource.org/licenses/OFL-1.1>.

---

# Table of Contents

Preface .....	v
1. Introducing OAuth 2.0 .....	1
AM as the OAuth 2.0 Authorization Server .....	2
AM and OAuth 2.0 Scopes .....	4
AM as an OAuth 2.0 Client and Resource Server .....	5
Using Your Own Client and Resource Server .....	6
Security Considerations .....	7
About Token Storage Location .....	7
2. Configuring AM for OAuth 2.0 .....	11
Configuring the OAuth 2.0 Provider Service .....	11
Registering OAuth 2.0 Clients With the OAuth 2.0 Provider Service .....	15
Configuring AM as an Authorization Server and Client .....	33
Configuring AM for Client-Based OAuth 2.0 Tokens .....	39
3. Authenticating OAuth 2.0 Clients .....	44
Authenticating Clients Using Form Parameters .....	44
Authenticating Clients Using Authorization Headers .....	45
Authenticating Clients Using JWT Profiles .....	45
Authenticating Clients Using Mutual TLS .....	49
Authenticating Clients when Using OpenID Connect 1.0 .....	53
4. Implementing OAuth 2.0 Grant Flows .....	55
Authorization Code Grant .....	56
Authorization Code Grant with PKCE .....	63
Implicit Grant .....	72
Resource Owner Password Credentials Grant .....	78
Client Credentials Grant .....	82
Device Flow .....	84
SAML v2.0 Profile for Authorization Grant .....	93
JWT Profile for OAuth 2.0 Authorization Grant .....	96
5. Managing OAuth 2.0 Refresh Tokens .....	102
6. Implementing OAuth 2.0 Proof-of-Possession .....	105
JWK-Based Proof-of-Possession .....	105
Certificate-Bound Proof-of-Possession .....	110
7. Managing OAuth 2.0 Consent .....	120
Allowing Clients To Skip Consent .....	120
Allowing the OAuth 2.0 Provider to Save Consent .....	121
Allowing Users to Revoke Consent .....	122
OAuth 2.0 Remote Consent Service .....	122
8. OAuth 2.0 Endpoints .....	133
/oauth2/authorize .....	133
/oauth2/bc-authorize .....	139
/oauth2/access_token .....	142
/oauth2/device/code .....	145
/oauth2/device/user .....	147
/oauth2/token/revoke .....	149

/oauth2/introspect .....	150
Legacy OAuth 2.0 endpoints .....	153
9. OAuth 2.0 Administration and Supporting REST Endpoints .....	162
/oauth2/register .....	162
/json/realm-config/agents/OAuth2Client .....	163
/users/user/oauth2/resources/sets .....	165
/users/user/oauth2/applications .....	167
10. Customizing OAuth 2.0 .....	169
Customizing OAuth 2.0 Scope Handling .....	169
Modifying the Content of Access Tokens .....	174
11. Reference .....	182
OAuth 2.0 Standards .....	182
OAuth 2.0 Sample Mobile Applications .....	183
OAuth2 Provider .....	183
Remote Consent Service .....	209
OAuth 2.0 and OpenID Connect 1.0 Client Settings .....	210
OAuth 2.0 Remote Consent Agent Settings .....	223
A. About the REST API .....	230
Introducing REST .....	230
About ForgeRock Common REST .....	230
Cross-Site Request Forgery (CSRF) Protection .....	248
REST API Versioning .....	249
Specifying Realms in REST API Calls .....	252
Authentication and Logout using REST .....	253
Using the Session Token After Authentication .....	272
Server Information .....	273
Token Encoding .....	274
Logging .....	274
Reference .....	276
B. About Scripting .....	279
The Scripting Environment .....	279
Global Scripting API Functionality .....	284
Managing Scripts .....	286
Scripting .....	299
C. Getting Support .....	303
Glossary .....	304

# Preface

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

This guide is written for anyone using OAuth 2.0 with Access Management to manage and federate access to web applications and web-based resources.

## 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

# Introducing OAuth 2.0

This chapter covers AM support for the OAuth 2.0 authorization framework.

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

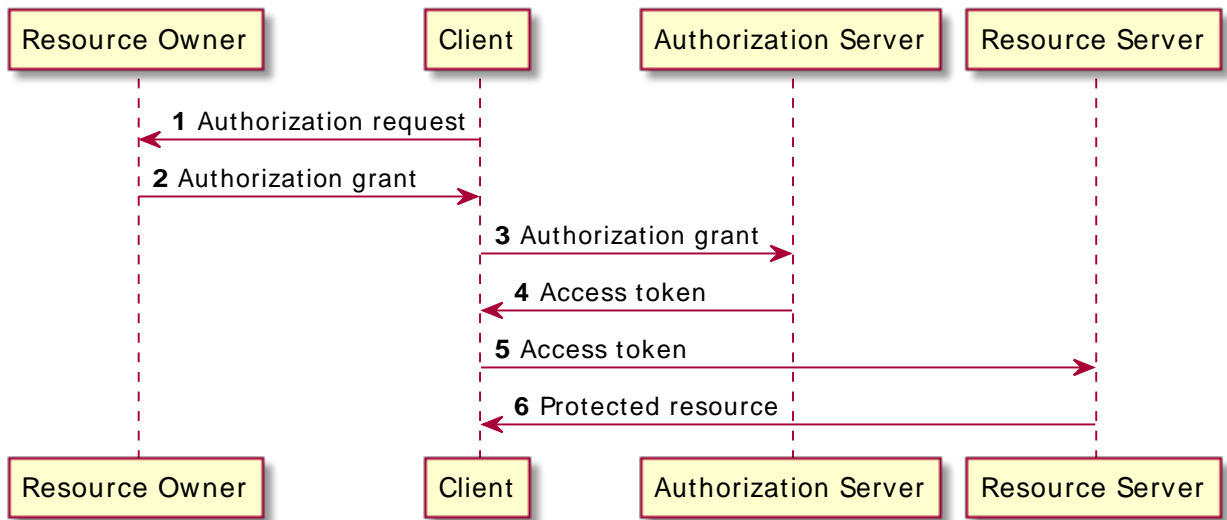
The main actors in the OAuth 2.0 authorization framework are the following:

### *OAuth 2.0 Framework Actors*

Actor	Description
<b>Resource Owner (RO)</b>	<p>The owner of the resources. For example, a user that has several photos stored in a photo-sharing service.</p> <p>The resource owner uses a <i>user-agent</i>, usually a web-browser, to communicate with the client.</p>
<b>Client</b>	<p>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.</p> <p>AM can act as a client.</p>
<b>Authorization Server (AS)</b>	<p>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.</p> <p>AM can act as the authorization server.</p>
<b>Resource Server (RS)</b>	<p>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.</p> <p>A website protected by a web or a Java agent can act as the resource server.</p>

The following sequence diagram demonstrates the basic OAuth 2.0 flow:

### OAuth 2.0 Protocol 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 that AM supports relating to OAuth 2.0.

## AM as the OAuth 2.0 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.

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 "Registering OAuth 2.0 Clients With the OAuth 2.0 Provider Service".

AM supports the following OAuth 2.0 standards and grant types:

### 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 "*Implementing OAuth 2.0 Grant Flows*".

#### Client Authentication Standards

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

For more information, see "*Authenticating OAuth 2.0 Clients*".

#### Other OAuth 2.0 Standards

- **JWT Proof-of-Possession**

For more information, see "*Implementing OAuth 2.0 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

For a complete list of all the supported OAuth 2.0 standards, see "OAuth 2.0 Standards".

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 "OAuth 2.0 Remote Consent Service".

- **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 "*Introducing Authorization*" and "*Implementing Authorization*" in the *Authorization Guide*.

## AM and OAuth 2.0 Scopes

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

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

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.

Every OAuth 2.0 flow requires scopes to limit the client's access to the resource owner's resources. For security reasons, AM only accepts scopes preconfigured in the `Scope(s)` or in the `Default Scope(s)` fields (Realms > *Realm Name* > Applications > OAuth 2.0 > Clients > *Client Name* > Core).

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 returns the `No scope requested` error. If the client requests a scope that is not preconfigured, AM returns an error, such as `Unknown/invalid scope(s)`.

AM does not use the default scopes in any other circumstance.

Since scopes 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. 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
<p>MY CLIENT</p> <p>This application is requesting the following private information:</p> <div style="border: 1px solid #ccc; padding: 5px; margin: 5px 0;">write</div> <p>You are signed in as: demo</p> <p style="text-align: right;"><input type="button" value="Deny"/> <input type="button" value="Allow"/></p>	<p>MY CLIENT</p> <p>This application is requesting the following private information:</p> <div style="border: 1px solid #ccc; padding: 5px; margin: 5px 0;">Allow the application to store the bill in your profile?</div> <p>You are signed in as: demo</p> <p style="text-align: right;"><input type="button" value="Deny"/> <input type="button" value="Allow"/></p>	<p>MY CLIENT</p> <p>This application is requesting access to your account</p> <p>You are signed in as: demo</p> <p style="text-align: right;"><input type="button" value="Deny"/> <input type="button" value="Allow"/></p>

You can configure how scopes appear in the consent screen by client or by realm, in the OAuth 2.0 provider service. For more information, see the Supported Scopes field in the provider's "Advanced" reference section or the Scope(s) field in "OAuth 2.0 and OpenID Connect 1.0 Client Settings".

Client level configuration overrides that at provider level.

For examples of requesting scopes from the authorization server, see any of the grant flows in "Implementing OAuth 2.0 Grant Flows".

### Tip

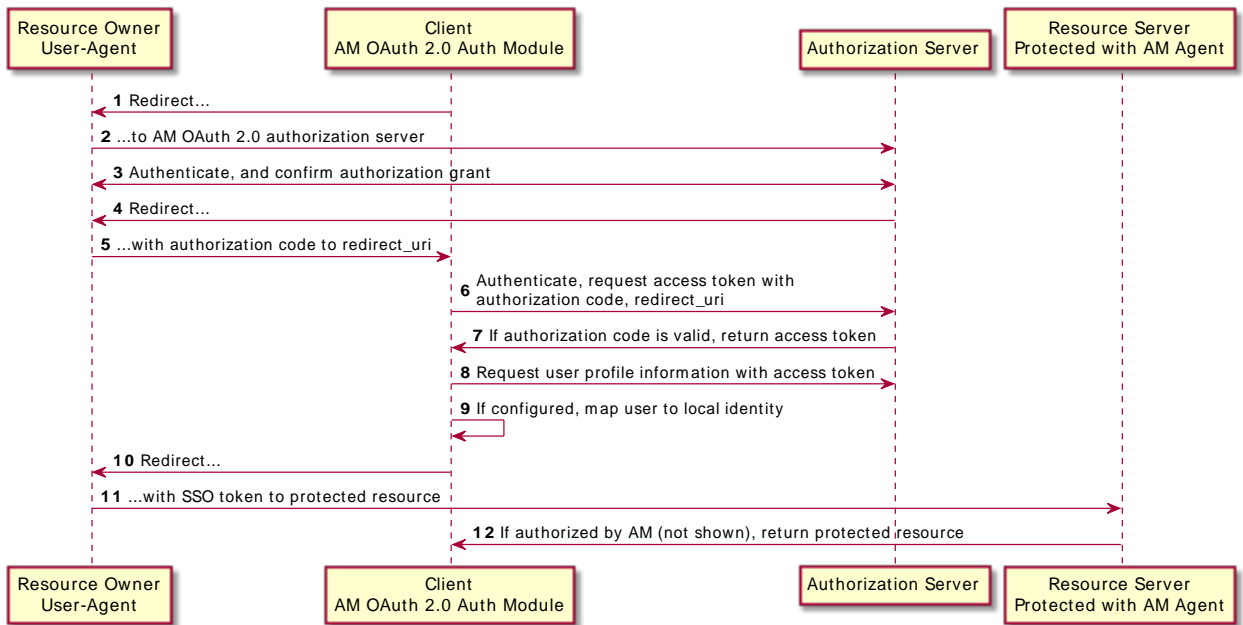
You can configure AM to grant scopes dynamically by evaluating authorization policies at runtime. For more information, see "Introducing Authorization" and "Implementing Authorization" in the *Authorization Guide*.

## AM as an OAuth 2.0 Client and Resource Server

AM can function as an OAuth 2.0 client for installations where the resources are protected by AM. To configure AM as an OAuth 2.0 client, you set up an OAuth 2.0 social authentication module instance, and then integrate the authentication module into your authentication chains as necessary.

When AM functions as an OAuth 2.0 client, AM provides an AM SSO session after successfully authenticating the resource owner and obtaining authorization. This means the client can then access resources protected by agents. In this respect the AM OAuth 2.0 client is just like any other authentication module, one that relies on an OAuth 2.0 authorization server to authenticate the resource owner and obtain authorization. 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 for example.

### OAuth 2.0 Client and Authorization Server



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

To configure AM as an OAuth 2.0 client, see the section "Social Authentication Modules" in the *Authentication and Single Sign-On Guide*.

## 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 allow you to 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.

## 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.

## 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 allows you to 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, retrieve 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 *Installation Guide*.

- 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": "eyJ0eXAiOiJKV1QiOiJ0OT05FIiwiaWF0IjoiYXNNTM5MDEzMzYyLsbSI6Ii8iLCj... ",
  "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: "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: "/",
  exp: 1539079567,
  iat: 1539075967,
  expires_in: 3600,
  jti: "FTQT6eZkDhm6PHEaSth0RoTLB80"
}
[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

# Configuring AM for OAuth 2.0

This chapter covers configuring AM support for OAuth 2.0.

## Configuring the OAuth 2.0 Provider Service

When you configure an OAuth 2.0 provider in a realm, AM creates an OAuth2 Provider Service in the realm and exposes the endpoints specified in "*OAuth 2.0 Endpoints*" and "*OAuth 2.0 Administration and Supporting REST Endpoints*".

The following procedure shows how to configure an OAuth 2.0 provider using the wizard, which populates several configuration options by default.

Note that an OAuth 2.0 provider created with this wizard will fail to provide ID tokens. If you need to support OpenID connect, see "*Configuring AM as an OpenID Connect Provider*" in the *OpenID Connect 1.0 Guide* instead.

### To Set Up the OAuth 2.0 Provider Service

Perform the steps in this procedure to set up the service with the Configure OAuth Provider wizard:

1. In the AM console, navigate to Realms > *Realm Name* > Dashboard > Configure OAuth Provider > Configure OAuth 2.0.
2. On the Configure OAuth2 page, select the Realm for the provider service.
3. (Optional) 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.
4. (Optional) Select Issue Refresh Tokens if you want the provider to supply a refresh token when returning an access token.
5. (Optional) Select Issue Refresh Tokens on Refreshing Access Tokens if you want the provider to supply a new refresh token when refreshing an access token.
6. (Optional) 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. For an example, see "*Customizing OAuth 2.0 Scope Handling*".



7. Select Create to save your changes. If an OAuth2 provider service already exists, it will be overwritten.

AM creates an OAuth2 provider service.

### *OAuth 2.0 Provider Server Additional Configuration*

The OAuth 2.0 provider is highly configurable:

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

Consider the following configuration options:

- To configure the OAuth 2.0 service to interact with AM's Authorization service to dynamically provide scopes, enable Use Policy Engine for Scope decisions.

Ensure you configure **OAuth2 Scope** resource type policies in the Default OAuth2 Scopes Policy Set. For more information, see "Resource Types, Policy Sets, and Policies" and "Configuring Policies" in the *Authorization Guide*.

- To configure response type plugins, add or remove lines from the Response Type Plugins field. Response type plugins let the provider issue access tokens, ID tokens, authorization codes, device codes, and others.

The following is a list of the plugins included in AM:

```
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** response type is required in OpenID Connect flows. AM uses it to issue ID tokens.
- The **none** response type is required in OpenID Connect flows using the **id\_token\_hint** parameter.
- The **code** response type is required in the Authorization Code grant flow.
- The **device\_code** response type is required in the Device grant flow.
- The **token** response type is required in all flows. AM uses it to issue access and refresh tokens.
- To change how scopes appear in the OAuth 2.0 consent pages, configure the Supported Scopes field on the Advanced tab.

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.`

For more information, see "AM and OAuth 2.0 Scopes".

- To change the OAuth 2.0 flows the provider allows, configure them in the Grant Types field on the Advanced tab.
- To allow users to save consent so the OAuth 2.0 provider remembers their consented scopes, see "Allowing the OAuth 2.0 Provider to Save Consent".
- To allow clients to skip consent so no consent page is displayed to the resource owners, see "To Allow Clients To Skip Consent".
- To change the attribute used to retrieve the user profile, for example, in cases where the resource owner should log in with their email address instead of with a username, see "To Change the Attribute Used to Retrieve the User Profile".
- To configure different types of client authentication, including JWT profiles and mutual TLS, see "Authenticating OAuth 2.0 Clients".
- To configure client-based tokens, such that clients can directly introspect the tokens without making a call to AM, see "Configuring AM for Client-Based OAuth 2.0 Tokens".
- To configure proof-of-possession, see "Implementing OAuth 2.0 Proof-of-Possession".
- To register clients or configure dynamic client registration, see "Dynamic Client Registration".

#### Tip

For more information about OpenID Connect configurations, see "OpenID Connect Provider Additional 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. In 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`.
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 "Configuring the Default Authentication Tree or Chain" in the *Authentication and Single Sign-On Guide*.

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

## Registering OAuth 2.0 Clients With the OAuth 2.0 Provider Service

You can register an OAuth 2.0 client with the AM OAuth 2.0 authorization service by creating and configuring an OAuth 2.0 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. 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 an OAuth 2.0 Client Profile Group".

### *To Create an OAuth 2.0 Client Profile*

Use the following procedure to create an OAuth 2.0 client profile:

- In the AM console, navigate to Realms > *Realm Name* > Applications > OAuth 2.0. 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.

To configure the client, see "To Configure an OAuth 2.0 Client Profile".

### *To Configure an OAuth 2.0 Client Profile*

1. In the AM console, navigate to Realms > *Realm Name* > Applications > OAuth 2.0 > *Client Name* to open the OAuth 2.0 Client page.
2. Adjust the configuration as needed using the inline help for hints, and also the documentation section "OAuth 2.0 and OpenID Connect 1.0 Client Settings".

Examine the client type option. An important decision to make at this point is whether your client is a confidential client or a public client. This depends on whether your client can keep its credentials confidential, or whether its credentials can be exposed to the resource owner or other parties. If your client is a web-based application running on a server, such as the AM OAuth 2.0 client, then you can keep its credentials confidential. If your client is a user-agent based client, such as a JavaScript client running in a browser, or a native application installed on a device used by the resource owner, then the credentials can be exposed to the resource owner or other parties.

3. When finished, save your work.

## To Configure an OAuth 2.0 Client Profile Group

1. In the AM console, navigate 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 using the inline help for hints, and also the documentation section "OAuth 2.0 and OpenID Connect 1.0 Client Settings".
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 an OAuth 2.0 Client Profile and Inherit Properties".

## To Assign a Group to an OAuth 2.0 Client Profile and Inherit Properties

1. In the AM console, navigate 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, for example, the Client secret property.

### OAuth 2.0 Client Profile Group Inheritance

Core
Advanced
OpenID Connect
Signing and Encryption
UMA

**Group**

Add the client to a group to allow inheritance of property values from the group. Changing the group will update inherited property values. Remove the group by selecting the name and pressing **BACKSPACE**. Inherited property values are copied to the client.

**Status**   ?

**Client secret**  ?

**Client type**   ?

**Redirection URIs**   ?

**Scope(s)**   ?

**Default Scope(s)**   ?

**Client Name**   ?

**Authorization Code Lifetime (seconds)**   ?

**Refresh Token Lifetime (seconds)**   ?

**Access Token Lifetime (seconds)**   ?

- 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.

## Dynamic Client Registration

AM supports dynamic registration as defined by RFC 7591, the *OAuth 2.0 Dynamic Client Registration Protocol*, and as defined by the the *OpenID Connect Dynamic Client Registration 1.0* specification. The protocols describe how authorization servers can allow OAuth 2.0 clients to register:

- Openly, without an access token, providing only their client metadata as a JSON resource.

AM generates `client_id` and `client_secret` values. AM ignores any values provided in the client metadata for these properties.

- Providing either a self-signed or a CA-signed X.509 certificate as authentication (OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens internet-draft) and its client metadata as JSON.
- By gaining authorization using an OAuth 2.0 access token, and with their client metadata.

The specification does not describe how the client obtains the access token. In AM, you can manually register an initial OAuth 2.0 client that obtains the access token on behalf of the client requesting registration.

You can use this method to provide a client with a specific client ID, and then allow the client to modify its metadata.

### Note

The `logo_uri`, `client_uri`, and `policy_uri` parameters are only accepted during dynamic client registration if the access token contains the `dynamic_client_registration` special scope.

- With client metadata that includes a *software statement*.

A software statement is a JWT that holds registration claims about the client, such as the issuer and the redirection URIs that it will register.

A software statement is issued by a *software publisher*. The software publisher encrypts and signs the claims in the software statement.

In AM, you store software publisher details as an agent profile. The software publisher profile identifies the issuer included in software statements, and holds information required to decrypt

software statement JWTs and to verify their signatures. When the client presents a software statement as part of the dynamic registration data, AM uses the software publisher profile to determine whether it can trust the software statement.

The protocol specification does not describe how the client obtains the software statement JWT. AM expects the software publisher to construct the JWT according to the settings in its agent profile.

## To Configure AM for Dynamic Client Registration

Perform the following steps to configure AM for dynamic client registration:

1. Configure an authorization service.

For details, see "To Set Up the OAuth 2.0 Provider Service".

2. In AM console under *Realm* > Services > OAuth2 Provider > Client Dynamic Registration, edit the relevant settings:

- To allow clients to register without an access token, enable Allow Open Dynamic Client Registration.

If you enable this option, consider some form of rate limiting. Also consider requiring a software statement.

- To require that clients present a software statement upon registration, enable Require Software Statement for Dynamic Client Registration, and edit the Required Software Statement Attested Attributes list to include the claims that must be present in a valid software statement. In addition to the elements listed, the issuer (*iss*) must be specified in the software statement's claims, and the issuer value must match the Software publisher issuer value for a registered software publisher agent.

As indicated in the protocol specification, AM rejects registration with an invalid software statement.

If the issuer is 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 *Installation Guide*.

For additional details, see "Client Dynamic Registration".

3. (Optional) If the clients will authenticate using mTLS with CA-signed (PKI) certificates, configure AM to hold the certificates belonging to the certificate authorities you want the instance of AM to trust. For more information, see "Mutual TLS Using Public Key Infrastructure".
4. (Optional) If you enabled Require Software Statement for Dynamic Client Registration, then you must register a software publisher:
  - a. In the AM console under *Realm* > Applications > Agents > Software Publisher, add a new software publisher agent.



If the publisher uses HMAC (symmetric) encryption for the software statement JWT, then the software publisher's password is also the symmetric key. This is called the Software publisher secret in the profile.

- b. In the software publisher profile, configure the appropriate security settings.

#### Important

- The Software publisher issuer value must match the `iss` value in claims of software statements issued by this publisher.
- If the publisher uses symmetric encryption, including `HS256`, `HS384`, and `HS512`, then the Software publisher secret must match the `k` value in the JWK.
- If you provide the JWK by URI rather than by value, AM must be able to access the JWK when processing registration requests.

5. (Optional) Review the following dynamic client registration examples:

- "Open Dynamic Client Registration Example"
- "Dynamic Client Registration Example with Mutual TLS Authentication"
- "Dynamic Client Registration Example With Access Token"
- "Dynamic Client Registration Example With Software Statement"

#### Note

AM returns an error when a client tries to register with an unsupported signing or encryption algorithm as part of its configuration.

For example, it will return an error if there is a typo in an algorithm, or if a public client tries to send a symmetric signing or encryption algorithm as part of its configuration: these algorithms are derived from the client's secret, which public clients do not have.

### Open Dynamic Client Registration Example

The following example shows dynamic registration with the Allow Open Dynamic Client Registration option enabled (Realms > *Realm Name* > Services > OAuth2 Provider > Client Dynamic Registration).

The client registers with its metadata as the JSON body of an HTTP POST to the registration endpoint. When specifying client metadata, be sure to include a `client_name` property that holds the human-readable name presented to the resource owner during consent:

```
$ curl \
```

```

--request POST \
--header "Content-Type: application/json" \
--data '{
"redirect_uris": ["https://client.example.com:8443/callback"],
"client_name#en": "My Client",
"client_name#ja-Jpan-JP": "\u30AF\u30E9\u30A4\u30A2\u30F3\u30C8\u540D",
"client_uri": "https://client.example.com/"
}' \
"https://openam.example.com:8443/openam/oauth2/register"
{
"request_object_encryption_alg": "",
"default_max_age": 1,
"application_type": "web",
"client_name#en": "My Client",
"registration_client_uri": "https://openam.example.com:8443/openam/oauth2/register?
client_id=2aeff083-83d7-4ba1-ab16-444ced02b535",
"client_type": "Confidential",
"userinfo_encrypted_response_alg": "",
"registration_access_token": "4637ee46-51df-4901-af39-fec5c3a1054c",
"client_id": "2aeff083-83d7-4ba1-ab16-444ced02b535",
"token_endpoint_auth_method": "client_secret_basic",
"userinfo_signed_response_alg": "",
"public_key_selector": "x509",
"authorization_code_lifetime": 0,
"client_secret": "6efb5636-6537-4573-b05c-6031cc54af27",
"user_info_response_format_selector": "JSON",
"id_token_signed_response_alg": "HS256",
"default_max_age_enabled": false,
"subject_type": "public",
"jwt_token_lifetime": 0,
"id_token_encryption_enabled": false,
"redirect_uris": ["https://client.example.com:8443/callback"],
"client_name#ja-jpan-jp": "#####",
"id_token_encrypted_response_alg": "RSA1_5",
"id_token_encrypted_response_enc": "A128CBC_HS256",
"client_secret_expires_at": 0,
"access_token_lifetime": 0,
"refresh_token_lifetime": 0,
"request_object_signing_alg": "",
"response_types": ["code"]
}
    
```

### 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"]`.

## Dynamic Client Registration Example with Mutual TLS Authentication

The following example shows the different properties required for clients using mTLS for authentication depending on the type of certificate presented:

## Clients using CA-signed X.509 certificates (PKI)

To use CA-signed certificates (PKI), you must configure AM to hold the certificates belonging to the certificate authorities you want the instance of AM to trust. For more information, see "Mutual TLS Using Public Key Infrastructure".

Include the following properties as part of the client metadata:

- **token\_endpoint\_auth\_method**, which must be set to `tls_client_auth`.
- **tls\_client\_auth\_subject\_dn**, which must be set to the distinguished name as it appears in the subject field of the certificate. For example, `CN=my0auth2Client`.

## Clients using self-signed X.509 certificates

Clients authenticating using self-signed certificates can provide their certificates for validation in one of the following ways:

- As a JWKS.

In this scenario, the client provides as part of its metadata the JWKS containing its certificate(s).

Include the following properties as part of the client metadata:

- **token\_endpoint\_auth\_method**, which must be set to `self_signed_tls_client_auth`.
- **A JWKS, configured as per RFC 7517**, which includes certificate information.
- As a JWKS URI, which AM will check periodically to retrieve the certificates from.

In this scenario, the client provides as part of its metadata the URI from where AM will retrieve the certificate(s) for validation.

Include the following properties as part of the client metadata:

- **token\_endpoint\_auth\_method**, which must be set to `self_signed_tls_client_auth`.
- **jwt\_uri**, which must be set to the URI from where AM will retrieve the certificates. For example, `https://www.example.com/mysecureapps/certs`.
- As an X.509 certificate.

In this scenario, the client provides as part of its metadata a single X.509 certificate in PEM format.

Include the following properties as part of the client metadata:

- **token\_endpoint\_auth\_method**, which must be set to `self_signed_tls_client_auth`.
- **tls\_client\_auth\_x509\_cert**, which must be set to an X.509 certificate in PEM format. You can choose to include or exclude the `-----BEGIN CERTIFICATE-----` and `-----END CERTIFICATE-----` labels.

The following example shows dynamic registration of a client that will provide their self-signed ECDSA P-256 certificate in a JWKS:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--data '{
  "jwks": {
    "keys": [{
      "kty": "EC",
      "crv": "P-256",
      "x": "9BmRru-6AYQ8U_9tUFhMGVG-BvC4vRthzLJTntfSdBA",
      "y": "MqPzVSeVNzzgcR-zZeLGog3GJ4d-doRE9eiGkCKrB48",
      "kid": "a4:68:90:1c:f6:c1:43:c0",
      "x5c": [
        "MIIBZTCCAQugAwIB....xgASSpAQc83FVBawjmbv6k4CN95G8zHsA=="
      ]
    }
  ]
},
"client_type": "Confidential",
"grant_types": ["authorization_code", "client_credentials"],
"response_types": ["code", "token"],
"redirect_uris": ["https://client.example.com:8443/callback"],
"token_endpoint_auth_method": "self_signed_tls_client_auth",
"tls_client_auth_subject_dn": "CN=my0auth2Client",
"tls_client_certificate_bound_access_tokens": true
}' \
"https://openam.example.com:8443/openam/oauth2/register"
{
  "request_object_encryption_alg": "",
  "default_max_age": 1,
  "jwks": {
    "keys": [
      {
        "kty": "EC",
        "crv": "P-256",
        "x": "9BmRru-6AYQ8U_9tUFhMGVG-BvC4vRthzLJTntfSdBA",
        "y": "MqPzVSeVNzzgcR-zZeLGog3GJ4d-doRE9eiGkCKrB48",
        "kid": "a4:68:90:1c:f6:c1:43:c0",
        "x5c": [
          "MIIBZTCCAQugAwIB....xgASSpAQc83FVBawjmbv6k4CN95G8zHsA=="
        ]
      }
    ]
  }
},
"application_type": "web",
"tls_client_auth_subject_dn": "CN=my0auth2Client",
"registration_client_uri": "https://openam.example.com:8443/openam/oauth2/register?
client_id=83635999-2794-4fcd-b6b3-67e2d86c1952",
"client_type": "Confidential",
"userinfo_encrypted_response_alg": "",
"registration_access_token": "tu4KR0j03iGn0ub00Y0YCSfyPmk",
"client_id": "83635999-2794-4fcd-b6b3-67e2d86c1952",
"token_endpoint_auth_method": "self_signed_tls_client_auth",
"userinfo_signed_response_alg": "",
"public_key_selector": "jwks",
...
}
```

Note that the example sets `tls_client_certificate_bound_access_tokens` to `true` to allow the client to obtain certificate-bound access tokens. For more information, see "Certificate-Bound Proof-of-Possession".

### 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"]`.

## Dynamic Client Registration Example With Access Token

The following example shows dynamic registration with default OAuth 2.0 provider service settings, providing an access token issued to a statically registered client.

In this example the statically registered client has the following profile settings:

### Client ID

```
masterClient
```

### Client secret

```
password
```

### Scope(s)

```
dynamic_client_registration
```

Prior to registration, obtain an access token:

```
$ curl \
--request POST \
--user "masterClient:password" \
--data "grant_type=password&username=admin&password=password&scope=dynamic_client_registration" \
https://openam.example.com:8443/openam/oauth2/realms/root/access_token
{
  "access_token": "5e7d1019-b752-43f1-af97-0d6fe2753105",
  "scope": "dynamic_client_registration",
  "token_type": "Bearer",
  "expires_in": 3599
}
```

The client registers with its metadata, providing the access token. When specifying client metadata, be sure to include a `client_name` property that holds the human-readable name presented to the resource owner during consent:

```
$ curl \
```

```

--request POST \
--header "Content-Type: application/json" \
--header "Authorization: Bearer 5e7d1019-b752-43f1-af97-0d6fe2753105" \
--data '{
  "redirect_uris": ["https://client.example.com/callback"],
  "client_name#en": "My Client",
  "client_name#ja-jpan-jp": "\u30AF\u30E9\u30A4\u30A2\u30F3\u30C8\u540D",
  "client_uri": "https://client.example.com/"
}' \
"https://openam.example.com:8443/openam/oauth2/register"
{
  "request_object_encryption_alg": "",
  "default_max_age": 1,
  "application_type": "web",
  "client_name#en": "My Client",
  "registration_client_uri": "https://openam.example.com:8443/openam/oauth2/register?
client_id=d58ba00b-da55-4fa3-9d2a-afe197207be5",
  "client_type": "Confidential",
  "userinfo_encrypted_response_alg": "",
  "registration_access_token": "5e7d1019-b752-43f1-af97-0d6fe2753105",
  "client_id": "d58ba00b-da55-4fa3-9d2a-afe197207be5",
  "token_endpoint_auth_method": "client_secret_basic",
  "userinfo_signed_response_alg": "",
  "public_key_selector": "x509",
  "authorization_code_lifetime": 0,
  "client_secret": "4da529de-3a18-4fb7-a0a9-07e05a394aa4",
  "user_info_response_format_selector": "JSON",
  "id_token_signed_response_alg": "HS256",
  "default_max_age_enabled": false,
  "subject_type": "public",
  "jwt_token_lifetime": 0,
  "id_token_encryption_enabled": false,
  "redirect_uris": ["https://client.example.com:8443/callback"],
  "client_name#ja-jpan-jp": "#####",
  "id_token_encrypted_response_alg": "RSA1_5",
  "id_token_encrypted_response_enc": "A128CBC_HS256",
  "client_secret_expires_at": 0,
  "access_token_lifetime": 0,
  "refresh_token_lifetime": 0,
  "request_object_signing_alg": "",
  "response_types": ["code"]
}
    
```

### 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"]`.

## Dynamic Client Registration Example With Software Statement

The following example extends "Dynamic Client Registration Example With Access Token" to demonstrate dynamic registration with a software statement.

In this example the software publisher has the following profile settings:

### Name

My Software Publisher

### Software publisher secret

secret

### Software publisher issuer

https://client.example.com

### Software statement signing Algorithm

HS256

### Public key selector

JWKS

### Json Web Key

```
{"keys": [{"kty": "oct", "k": "secret", "alg": "HS256"}]}
```

Notice that the value is a key set rather than a single key.

In this example, the software statement JWT is as shown in the following listing, with lines folded for legibility:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.  
eyJpc3MiOiJodHRwczovL2NsaWVudC5leGFtcGxlLmNvbSIsImhdCI6MTUwNjY3MTg1MSwiZX  
hwIjoxNTM4MjA3ODUxLCJhdWQiOiJvcGVuYW0uZXhhbXBsZS5jb20iLCJzdWIiOiI0TlJCMS0w  
WFpBQlplJ0U2LTVTTTNSIiwicmVkaXJlY3RfdXJpcyI6WyJodHRwczovL2NsaWVudC5leGFtcG  
xlLmNvbS9jYXN5YmFjayJdfQ.  
IOxZawTOzSPkEkrXC9nj8RDrpulzzMuZ-4R7_0l_jhw
```

This corresponds to the HS256 encrypted and signed JWT with the following claims payload.:

```
{  
  "iss": "https://client.example.com",  
  "iat": 1506671851,  
  "exp": 1538207851,  
  "aud": "openam.example.com",  
  "sub": "4NRB1-0XZABZI9E6-5SM3R",  
  "redirect_uris": [  
    "https://client.example.com/callback"  
  ]  
}
```

Prior to registration, obtain an access token:

```
$ curl --request POST \
--data "grant_type=password" \
--data "username=demo" \
--data "password=changeit" \
--data "scope=dynamic_client_registration" \
--data "client_id=masterClient" \
--data "client_secret=password" \
"https://openam.example.com:8443/openam/oauth2/realms/root/access_token"
{
  "access_token": "06bfc193-1f7b-49a1-9926-ffe19e2f5f70",
  "scope": "dynamic_client_registration",
  "token_type": "Bearer",
  "expires_in": 3599
}
```

The client registers with its metadata that includes the software statement, providing the access token. When specifying client metadata, be sure to include a `client_name` property that holds the human-readable name presented to the resource owner during consent:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "Authorization: Bearer 06bfc193-1f7b-49a1-9926-ffe19e2f5f70" \
--data '{
  "redirect_uris": ["https://client.example.com:8443/callback"],
  "client_name#en": "My Client",
  "client_name#ja-Jpan-JP": "\u30AF\u30E9\u30A4\u30A2\u30F3\u30C8\u540D",
  "client_uri": "https://client.example.com/",
  "software_statement": "eyJ0eXAiOiJKV1QiLCJ6Ww...9nj8RDrpulzzMuZ-4R7_0l_jhw"
}' \
"https://openam.example.com:8443/openam/oauth2/register"
{
  "request_object_encryption_alg": "",
  "default_max_age": 1,
  "application_type": "web",
  "client_name#en": "My Client",
  "registration_client_uri": "https://openam.example.com:8443/openam/oauth2/register?
client_id=086658c1-0517-4667-bc2d-6786224eb126",
  "client_type": "Confidential",
  "userinfo_encrypted_response_alg": "",
  "registration_access_token": "06bfc193-1f7b-49a1-9926-ffe19e2f5f70",
  "client_id": "086658c1-0517-4667-bc2d-6786224eb126",
  "token_endpoint_auth_method": "client_secret_basic",
  "userinfo_signed_response_alg": "",
  "software_statement": "eyJ0eXAiOiJKV1QiLCJ6Ww...9nj8RDrpulzzMuZ-4R7_0l_jhw",
  "public_key_selector": "x509",
  "authorization_code_lifetime": 0,
  "client_secret": "272e26a4-b4ea-4033-bfd3-8b1be2c9aa22",
  "user_info_response_format_selector": "JSON",
  "id_token_signed_response_alg": "HS256",
  "default_max_age_enabled": false,
  "subject_type": "public",
  "jwt_token_lifetime": 0,
  "id_token_encryption_enabled": false,
  "redirect_uris": ["https://client.example.com:8443/callback"],
  "client_name#ja-jpan-jp": "#####",
  "id_token_encrypted_response_alg": "RSA1_5",
  "id_token_encrypted_response_enc": "A128CBC_HS256",
```



```
"client_secret_expires_at": 0,  
"access_token_lifetime": 0,  
"refresh_token_lifetime": 0,  
"request_object_signing_alg": "",  
"response_types": ["code"]  
}
```

### 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"]`.

## Dynamic Client Registration Management

AM allows clients to manage their information dynamically, as per RFC 7592 *OAuth 2.0 Dynamic Client Registration Management Protocol* and OpenID Connect Dynamic Client Registration 1.0 incorporating errata set 1. This RFC is an extension of the Dynamic Client Registration Protocol (RFC 7591)

Note that, during dynamic client registration, AM supplies clients with the following information:

- `registration_client_uri`. The FQDN of the client configuration endpoint the client can use to update their data. This endpoint always contains the client ID as a query parameter.
- `registration_access_token`. The token clients must use to authenticate to the client configuration endpoint.

Clients must store this information, since it is mandatory to read, update, and modify their profile information.

### To Configure Client Dynamic Registration Management

1. Configure AM for dynamic client registration.

For more information, see "To Configure AM for Dynamic Client Registration".

2. Log in to the AM console as an administrative user, such as `amAdmin`.
3. Navigate to Realms > *Realm Name* > Services > OAuth2 Provider > Client Dynamic Registration, and ensure that Generate Registration Access Tokens is enabled.
4. Save your changes, if required.
5. (Optional) Review the following examples:

- "Client Read Request Example"
- "Client Update Request Example"
- "Client Deletion Request Example"

### Client Read Request Example

Clients can use the read request to retrieve their current configuration from AM.

In this example, a client dynamically registered using the following command:

```
$ curl \
  --request POST \
  --header "Content-Type: application/json" \
  --data '{
    "redirect_uris": ["https://client.example.com:8443/callback"],
    "client_name#en": "My Client",
    "client_name#ja-Jpan-JP": "\u30AF\u30E9\u30A4\u30A2\u30F3\u30C8\u540D",
    "client_uri": "https://client.example.com/"
  }' \
  "https://openam.example.com:8443/openam/oauth2/register"
```

Among the returned information, the OAuth 2.0/OpenID Connect provider supplied the client with the following data:

```
...
"registration_client_uri":"https://openam.example.com:8443/openam/oauth2/register?
client_id=77f296b3-5293-4219-981d-128322c1e173",
"registration_access_token":"o5IlBX0xC7RGPTIe8is_zzz6Yqg",
...
```

To request the information stored in its client profile, clients perform an HTTP GET request to the client registration endpoint. Use the registration token to authenticate to the endpoint by sending the token as an authorization bearer header. For example:

```
$ curl \
  --request GET \
  --header "Authorization: Bearer o5IlBX0xC7RGPTIe8is_zzz6Yqg" \
  "https://openam.example.com:8443/openam/oauth2/register?client_id=77f296b3-5293-4219-981d-128322c1e173"
{
  "request_object_encryption_alg": "",
  "default_max_age": 1,
  "application_type": "web",
  "client_name#en": "My Client",
  "userinfo_encrypted_response_enc": "A128CBC_HS256",
  "client_type": "Confidential",
  "userinfo_encrypted_response_alg": "",
  "token_endpoint_auth_method": "client_secret_basic",
  "userinfo_signed_response_alg": "",
  "client_id": "77f296b3-5293-4219-981d-128322c1e173",
  "public_key_selector": "x509",
  "client_secret": "vXxY3HJ7_...84qfRQHYW3QbZfDSXieAgIVa2tg",
  ....
}
```

If the OAuth 2.0/OpenID Connect provider is configured for a subrealm rather than the top-level realm, you must specify it in the endpoint. For example, if the provider is configured for the `/customers` realm, then use `/oauth2/realms/root/realms/customers/register`.

The server does not return the `registration_client_uri` nor the `registration_access_token` attributes.

### Client Update Request Example

Clients can use the update request to modify their client profile while retaining their client ID.

In this example, a client dynamically registered using the following command:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--data '{
"redirect_uris": ["https://client.example.com:8443/callback"],
"client_name#en": "My Client",
"client_name#ja-Jpan-JP": "\u30AF\u30E9\u30A4\u30A2\u30F3\u30C8\u540D",
"client_uri": "https://client.example.com/"
}' \
"https://openam.example.com:8443/openam/oauth2/register"
```

Among the returned information, the OAuth 2.0/OpenID Connect provider supplied the client with the following data:

```
...
"registration_client_uri":"https://openam.example.com:8443/openam/oauth2/register?
client_id=77f296b3-5293-4219-981d-128322c1e173",
"registration_access_token":"o5IlBX0xC7RGPTIe8is_zzz6Yqg",
"grant_types":["authorization_code"],
...
```

The client performs a read request to the OAuth 2.0/OpenID Connect provider to retrieve its current configuration.

When updating the client's metadata, AM:

- Resets to the default value any client setting not sent with the update request.
- Returns a new registration access token to the client.
- Rejects requests where the client secret does not match with the one already registered.
- Rejects requests containing the following metadata (as per RFC 7592):
  - `registration_access_token`
  - `registration_client_uri`
  - `client_secret_expires_at`

- `client_id_issued_at`

To update the metadata, clients make an HTTP PUT request to the registration endpoint. The request contains all the metadata returned from the read request minus the information that, as specified by the spec, should not be sent. Use the registration token to authenticate to the endpoint by sending the token as an authorization bearer header.

In the following example, the OAuth 2.0 client sends the required metadata to AM, updating its grant types to `"authorization_code","implicit"`:

```
$ curl \
--request PUT \
--header "Authorization: Bearer o5lLBX0xC7RGPTIe8is_zzz6Yqg" \
--data '{
  "request_object_encryption_alg": "",
  "default_max_age": 1,
  "application_type": "web",
  "client_name#en": "My Client",
  "userinfo_encrypted_response_enc": "",
  "client_type": "Confidential",
  "userinfo_encrypted_response_alg": "",
  "token_endpoint_auth_method": "client_secret_basic",
  "userinfo_signed_response_alg": "",
  "client_id": "77f296b3-5293-4219-981d-128322c1e173",
  "public_key_selector": "x509",
  "scope": "write",
  "authorization_code_lifetime": 0,
  "client_secret": "vXxY3HJ7...84qfRQHfW3QbZfDSXieAgIVa2tg",
  "user_info_response_format_selector": "JSON",
  "tls_client_certificate_bound_access_tokens": false,
  "id_token_signed_response_alg": "RS256",
  "default_max_age_enabled": false,
  "subject_type": "public",
  "grant_types": ["authorization_code", "implicit"],
  "jwt_token_lifetime": 0,
  "id_token_encryption_enabled": false,
  "redirect_uris": ["https://client.example.com:8443/callback"],
  "client_name#ja-jpan-jp": "#####",
  "id_token_encrypted_response_alg": "RSA-OAEP-256",
  "id_token_encrypted_response_enc": "A128CBC-HS256",
  "access_token_lifetime": 0,
  "refresh_token_lifetime": 0,
  "scopes": ["write"],
  "request_object_signing_alg": "",
  "response_types": ["code"]
}' \
"https://openam.example.com:8443/openam/realms/root/oauth2/register?
client_id=77f296b3-5293-4219-981d-128322c1e173"
{
  "request_object_encryption_alg": "",
  "default_max_age": 1,
  "application_type": "web",
  "client_name#en": "My Client",
  "userinfo_encrypted_response_enc": "",
  "client_type": "Confidential",
  "userinfo_encrypted_response_alg": "",
  "registration_access_token": "NrvX2bqydMgr...EGI32YuvyrkxDpD_xJVHtHo6fXQ",
```

```

"client_id": "77f296b3-5293-4219-981d-128322c1e173",
...
}

```

If the OAuth 2.0/OpenID Connect provider is configured for a subrealm rather than the top-level realm, you must specify it in the endpoint. For example, if the provider is configured for the `/customers` realm, then use `/oauth2/realms/root/realms/customers/register`.

When successful, AM returns an HTTP 200 message and the profile data with the changes. *Note that the registration access token has changed; the client must store the new token securely.*

### Client Deletion Request Example

Clients can use the delete request to deprovision themselves from AM.

In this example, a client dynamically registered using the following command:

```

$ curl \
--request POST \
--header "Content-Type: application/json" \
--data '{
"redirect_uris": ["https://client.example.com:8443/callback"],
"client_name#en": "My Client",
"client_name#ja-Jpan-JP": "\u30AF\u30E9\u30A4\u30A2\u30F3\u30C8\u540D",
"client_uri": "https://client.example.com/"
}' \
"https://openam.example.com:8443/openam/oauth2/register"

```

Among the returned information, the OAuth 2.0/OpenID Connect provider supplied the client with the following data:

```

...
"registration_client_uri": "https://openam.example.com:8443/openam/oauth2/register?
client_id=77f296b3-5293-4219-981d-128322c1e173",
"registration_access_token": "o5IlBX0xC7RGPTIe8is_zzz6Yqg",
...

```

To deprovision themselves, clients send an HTTP DELETE request to the client registration endpoint. Use the registration token to authenticate to the endpoint by sending the token as an authorization bearer header. For example:

```

$ curl \
--request DELETE \
--header "Authorization: Bearer o5IlBX0xC7RGPTIe8is_zzz6Yqg" \
"https://openam.example.com:8443/openam/oauth2/register?client_id=77f296b3-5293-4219-981d-128322c1e173"

```

If the OAuth 2.0/OpenID Connect provider is configured for a subrealm rather than the top-level realm, you must specify it in the endpoint. For example, if the provider is configured for the `/customers` realm, then use `/oauth2/realms/root/realms/customers/register`.

If the deprovision is successful, AM returns an HTTP 204 No Content message. AM does not invalidate the authorization grants or active tokens associated with the client, which will expire in time. However, new requests to OAuth 2.0 endpoints will fail since the client no longer exists.

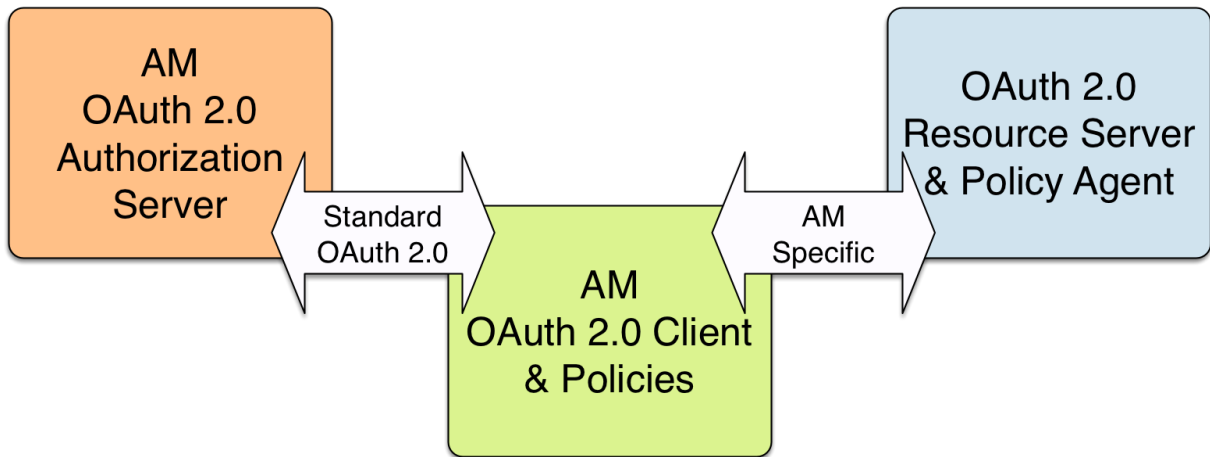
## Configuring AM as an Authorization Server and Client

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*

<http://authz.example.com:8080/openam/>

<http://www.example.com:8080/examples/>



<http://client.example.com:8080/openam/>

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 to be 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 "Configuring the OAuth 2.0 Provider Service".
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 "Registering OAuth 2.0 Clients With the OAuth 2.0 Provider Service".
5. Log out and access the protected resources to see the process in action.

## Web Site Protected 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 [Web Policy Agents](#) documentation or the [Java Policy Agents](#) documentation 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 "Implementing 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 `changeit`. 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

### Apache Tomcat Examples

- [Servlets examples](#)
- [JSP Examples](#)
- [WebSocket 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 "Configuring the OAuth 2.0 Provider Service".

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, 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 From a 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, 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 "Registering OAuth 2.0 Clients With the OAuth 2.0 Provider Service".

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 instruction 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 `changeit`, AM presents you with an authorization decision page.

### *Presenting Authorization Decision Page to Resource Owner*

#### **OAuth authorization page**

#### **Application requesting scope**

:

**The following private info is requested**

Save Consent:  Allow  Deny

When you click Allow, the authorization service creates an SSO session, and redirects the client back to the resource, thus allowing the client to 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

### Apache Tomcat Examples

- [Servlets examples](#)
- [JSP Examples](#)
- [WebSocket Examples](#)

## 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. Log in to the AM console as an administrative user, for example, `amAdmin`.
2. Navigate to Realms > *Realm Name* > Services > OAuth2 Provider.
3. In the Core tab, enable Use Client-Based Access & Refresh Tokens.
4. (Optional) Enable Issue Refresh Tokens and/or Issue Refresh Tokens on Refreshing Access Tokens.
5. Save your changes.
6. Configure client-based token blacklisting. For more information, see "Configuring Client-Based OAuth 2.0 Token Blacklisting".
7. 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. Log in to the AM console as an administrative user, for example, `amAdmin`.
2. Navigate to Configure > Global Services > Global > OAuth2 Provider.
3. 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

4. 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

5. 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

6. 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*

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 *Setup and Maintenance 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 its JSON payload, which is outlined in the JSON Web Algorithm specification (RFC 7518).

### 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.

### 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

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	rsajwt signingkey	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 *Setup and Maintenance Guide*.

3. Save your changes.

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



## Chapter 3

# Authenticating OAuth 2.0 Clients

AM can authenticate 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.

For information about the different authentication methods for those clients with a secret or a JWT, see the following sections.

## 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.

## 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 `ZXhhbXBsZS5jb206cyUzRGNyJTl1dA==`

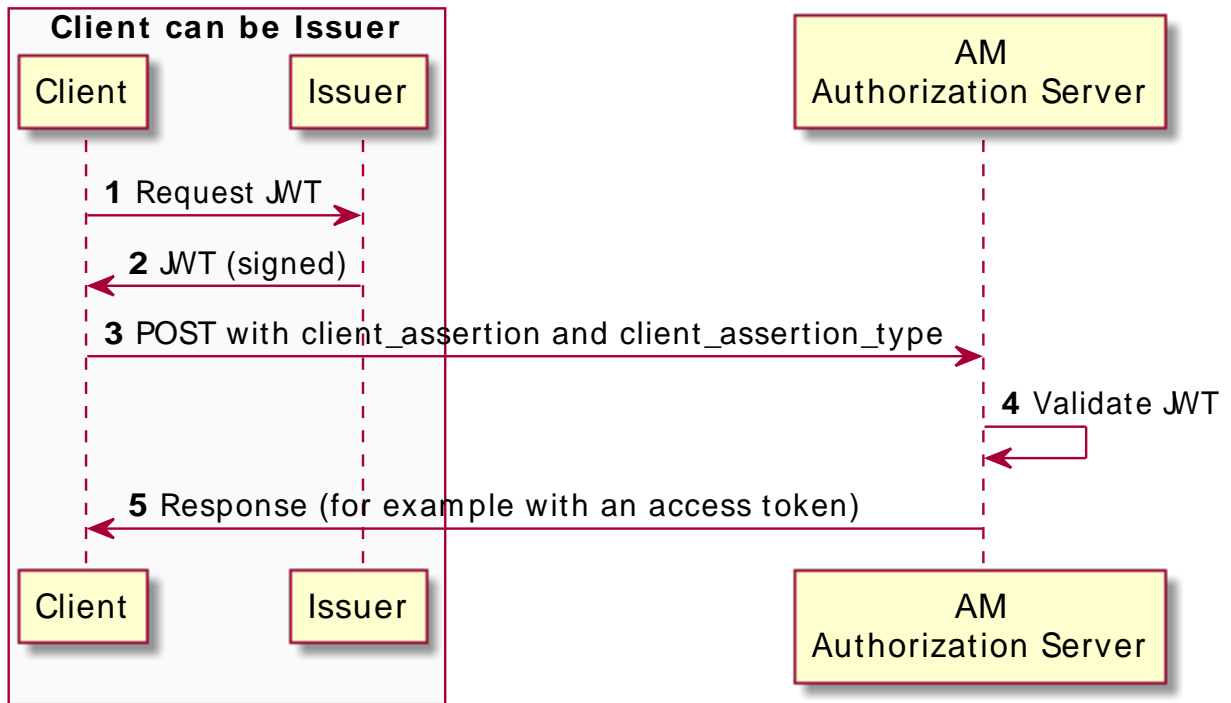
Ensure that communication with the authorization server happens over a secure protocol to help protect the credentials.

## 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. The client itself can be the issuer, in which case it will create a JWT for itself before starting the OAuth 2.0 flow.
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/access\\_token](https://openam.example.com:8443/openam/oauth2/realms/root/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.

For more information about the JWT, see 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.

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. 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. 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": "AL4kjjz74rDo3VQ3Wx...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. 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. 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 length 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 "Authenticating Clients when Using OpenID Connect 1.0".

#### Note

AM does not support non-string JWT header parameters, such as `jku` and `jwe`:

- AM 6.5.2 and later parse the JWT but ignores the non-string header parameters.
- AM versions earlier than 6.5.2 do not parse the JWT, and will return an HTTP 500 error message.

Configure the public keys/certificates in AM instead of adding the headers, as explained in the relevant sections of the documentation.

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=eyJYbnIjogILJTMjU2IiB9.eyJhc3ViIjogImp3..."
...
```

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 OpenAM 12.x, 13.x and 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 "Registering OAuth 2.0 Clients With the OAuth 2.0 Provider Service".

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 administration console, navigate 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 *Setup and Maintenance 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 administration console, navigate 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 administration console, navigate 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.

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 "Registering OAuth 2.0 Clients With the OAuth 2.0 Provider Service".

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.

### 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/access_token"  
{  
  "access_token": "sbQZuveFumUDV5R1vVB16QAGNB8",  
  "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 PEM-encoded certificate to AM in the trusted header.
- 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 administration console, navigate 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 allow an attacker to attempt to send malicious header values. A trusted header name that is difficult to guess makes this type of attack more difficult.

# Authenticating Clients when Using OpenID Connect 1.0

When using OpenID Connect, you must specify in the client profile the type of authentication the client is using. To configure the authentication method, navigate to Realms > *Realm Name* >

Applications > OAuth 2.0 > Advanced, and select one of the following options in the Token Endpoint Authentication Method drop down:

- **client\_secret\_post**, if the client sends its credentials as form parameters.
- **client\_secret\_basic**, if the client sends its credentials in a basic authorization header.
- **private\_key\_jwt**, if the client sends its credentials as a JWT.
- **tls\_client\_auth**, if the client uses a CA-signed certificate for mutual TLS authentication.
- **self\_signed\_tls\_client\_auth**, if the client uses a self-signed certificate for mutual TLS authentication.
- **none**, if the client is public.

AM will not require a public client to authenticate even if the authentication method is set to a value different from **none**.

## Chapter 4

# Implementing 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 <code>.war</code> application.	<b>Authorization Code</b>	<i>(RFC 6749)</i> 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.	<b>Authorization Code with PKCE</b>	<p>Authorization Code with PKCE</p> <p><i>(RFC 6749, RFC 7636)</i> 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.</p> <p>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.</p>
The client is a SPA. For example, a JavaScript application.	<b>Implicit</b>	<p><i>(RFC 6749)</i> 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.</p> <p>For security purposes, you should use the Authorization Code grant with PKCE when possible.</p>
The client is trusted with the resource owner credentials. For example, the resource owner's operating system.	<b>Resource Owner Password Credentials</b>	<i>(RFC 6749)</i> 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.	<b>Client Credentials</b>	(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.	<b>Device Flow</b>	(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 client. For example, an application in an environment where a SAML v2.0 ecosystem coexists with an OAuth 2.0 one.	<b>SAML v2.0 Profile</b>	(RFC 7522 ) The client uses the resource owner's SAML 2.0 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 client 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.	<b>JWT Bearer Profile</b>	(RFC 7523 ) The client uses a signed JWT to obtain an access token from the authorization server without interacting with the resource owner.

### Tip

AM supports associating a confirmation key or a certificate with an access token to support proof-of-possession interactions.

For more information, see "[Implementing OAuth 2.0 Proof-of-Possession](#)"

## Authorization Code Grant

### Endpoints

- /oauth2/authorize
- /oauth2/access\_token

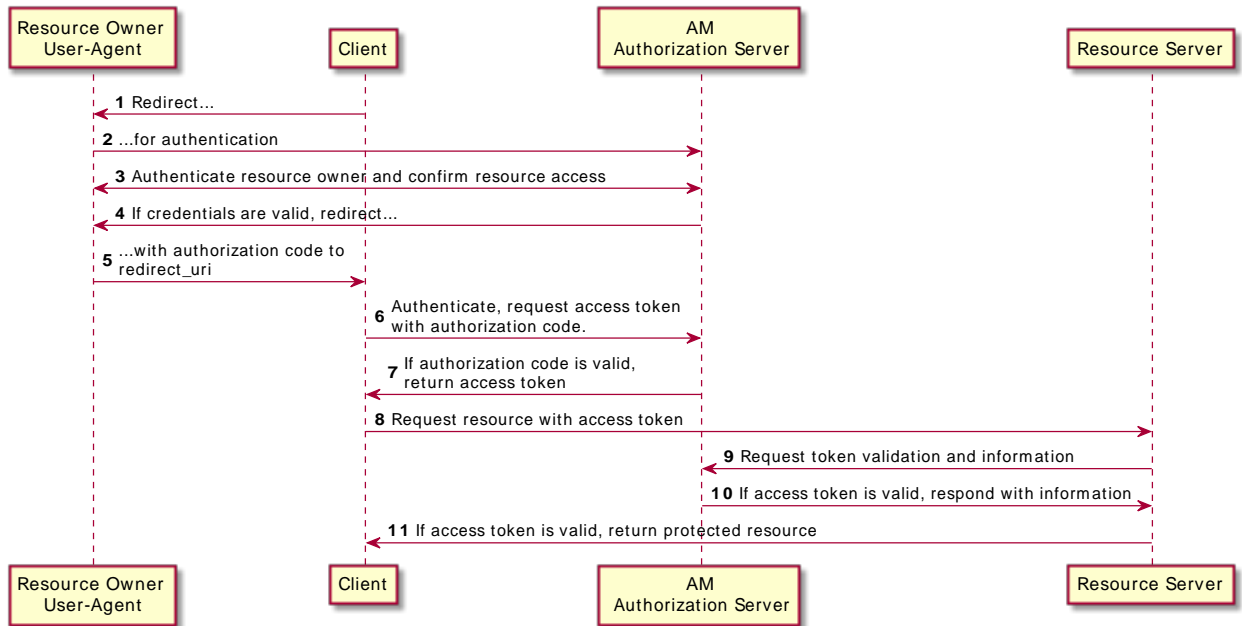
The Authorization Code grant is a two-step interactive process used when the client, for example, a Java application running on a server, requires access to protected resources.

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.

The following diagram demonstrates the Authorization Code grant flow:

### OAuth 2.0 Authorization Code Grant Flow



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 access to 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 in the top-level realm. Ensure that:
  - The `code` Response Type Plugins is configured.
  - The `Authorization Code` Supported Grant Type is configured.

For more information, see "Configuring the OAuth 2.0 Provider Service".

- 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 "Registering OAuth 2.0 Clients With the OAuth 2.0 Provider Service".

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 `/customers` realm, then use `/oauth2/realms/root/realms/customers/authorize`.

For example:

```
https://openam.example.com:8443/openam/oauth2/realms/root/authorize \
?client_id=myClient \
&response_type=code \
&scope=write \
&state=abc123 \
&redirect_uri=https://www.example.com:443/callback
```

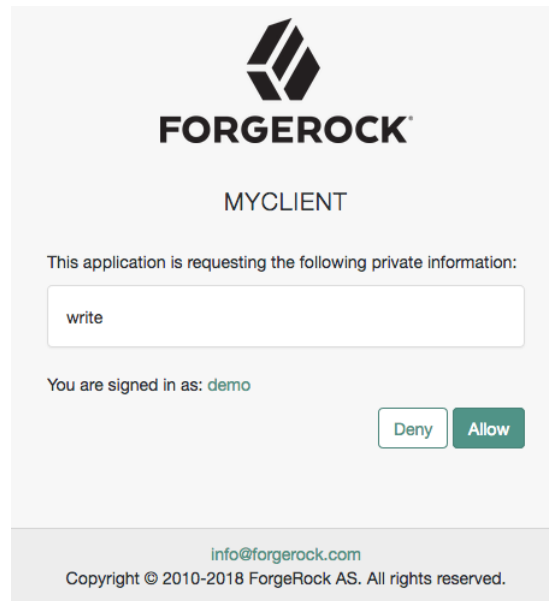
Note that the URL is split and spaces have been added for readability purposes and that the `scope` and `state` parameters have been included. Scopes are not required, since they can be configured by default in the authorization server and the client, and have been added only as an example. The `state` 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



- 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.

- 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
```

- 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 in the top-level realm.
  - The **code** Response Type Plugins is configured.
  - The **Authorization Code** Supported Grant Type is configured.

For more information, see "Configuring the OAuth 2.0 Provider Service".

- 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 "Registering OAuth 2.0 Clients With the OAuth 2.0 Provider Service".

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: changeit" \
--header "Accept-API-Version: resource=2.0, protocol=1.0" \
'https://openam.example.com:8443/openam/json/realms/root/authenticate'
{
  "tokenId": "AQIC5wM...TU30Q*",
  "successUrl": "/openam/console",
  "realm": "/"
}
```

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 `/customers` realm, then use `/oauth2/realms/root/realms/customers/authorize`.

For example:

```
$ curl --dump-header - \
--request POST \
--Cookie "iPlanetDirectoryPro=AQIC5wM...TU30Q*" \
--data "scope=write" \
--data "response_type=code" \
--data "client_id=myClient" \
--data "csrf=AQIC5wM...TU30Q*" \
--data "redirect_uri=https://www.example.com:443/callback" \
--data "state=abc123" \
--data "decision=allow" \
"https://openam.example.com:8443/openam/oauth2/realms/root/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 *"Authenticating OAuth 2.0 Clients"*.

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 `/customers` realm, then use `/oauth2/realm/root/realm/customers/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/realm/root/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 *"Managing OAuth 2.0 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.

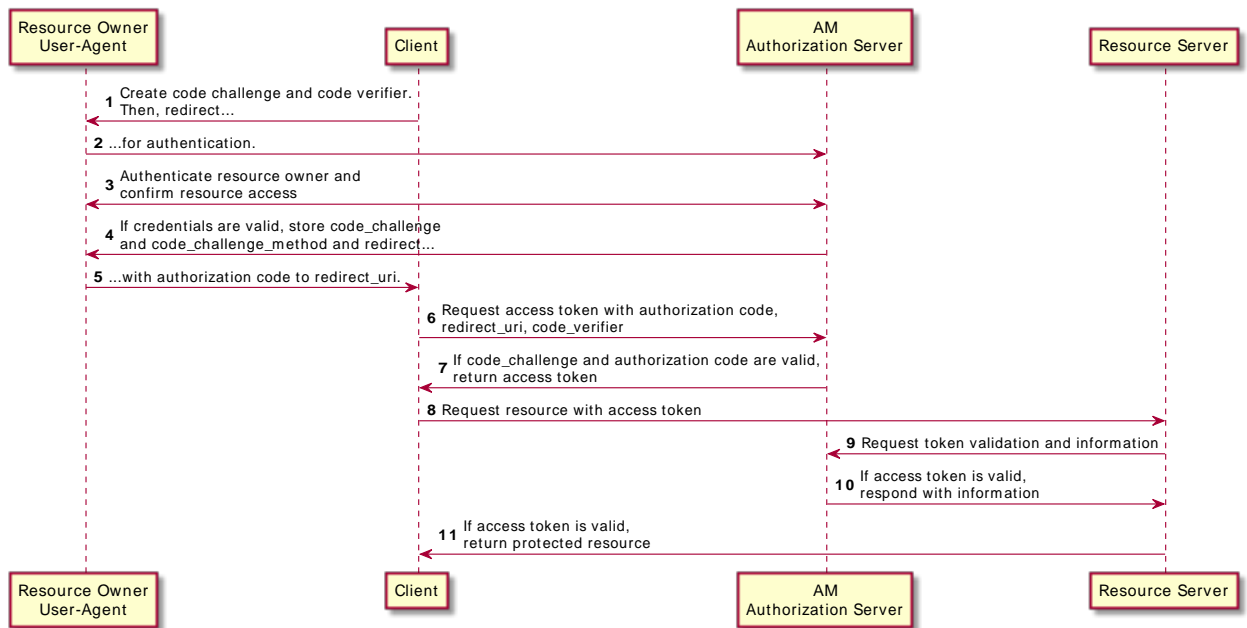
Since communication between the client and the authorization server is not secure, clients are usually *public* so their secrets do not get compromised. Also, browser-based clients making OAuth 2.0 requests to different domains must implement Cross-Origin Resource Sharing (CORS) calls to access OAuth 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.

The following diagram demonstrates the Authorization Code grant with PKCE flow:

### OAuth 2.0 Authorization Code Grant with PKCE Flow



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 access to 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_F40duTI4` 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 with the following configuration:
  - **Supported Scopes:** `write`

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

For more information, see "Registering OAuth 2.0 Clients With the OAuth 2.0 Provider Service".

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 `/customers` realm, then use `/oauth2/realms/root/realms/customers/authorize`.

For example:

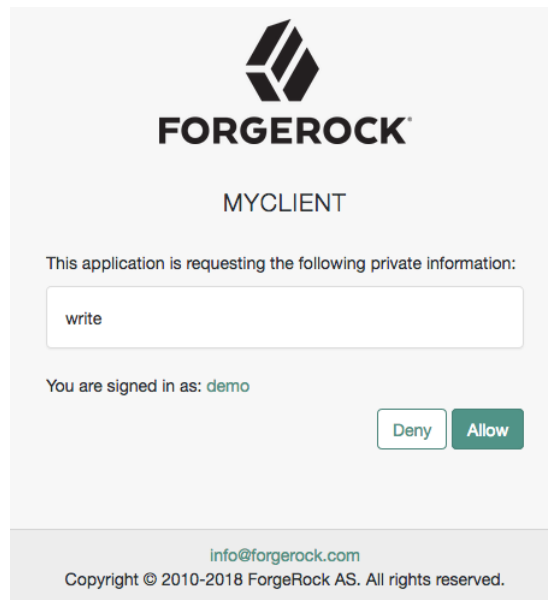
```
https://openam.example.com:8443/openam/oauth2/realms/root/authorize \
?client_id=myClient \
&response_type=code \
&scope=write \
&redirect_uri=https://www.example.com:443/callback \
&code_challenge=j3wKnK2Fa_mc2tgdqa6GtUfCYjdWSA5S23JKTTtPF8Y \
&code_challenge_method=S256 \
&state=abc123
```

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-6N0CYKQgvDPqtg&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 with the following configuration:

- **Supported Scopes:** **write**

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 token**
- **Grant Types:** **Authorization Code**

For more information, see "Registering OAuth 2.0 Clients With the OAuth 2.0 Provider Service".

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: changeit" \
--header "Accept-API-Version: resource=2.0, protocol=1.0" \
'https://openam.example.com:8443/openam/json/realms/root/authenticate'
{
  "tokenId": "AQIC5wM...TU30Q*",
  "successUrl": "/openam/console",
  "realm": "/"
}
```

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 `/customers` realm, then use `/oauth2/realms/root/realms/customers/authorize`.

For example:

```
$ curl --dump-header - \
--request POST \
--Cookie "iPlanetDirectoryPro=AQIC5wM...TU30Q*" \
--data "redirect_uri=https://www.example.com:443/callback" \
--data "scope=write" \
--data "response_type=code" \
--data "client_id=myClient" \
--data "csrf=AQIC5wM...TU30Q*" \
--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/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 "*Authenticating OAuth 2.0 Clients*".

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_G0SE9WlxzS69Cq0mQh8uyFaeEbI1LlW8tHs62SmEE6n7Nke0XJGx_F40duTI4" \  
"https://openam.example.com:8443/openam/oauth2/realms/root/access_token"
```

The `client_id` and the `redirect_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 "[Managing OAuth 2.0 Refresh Tokens](#)".

## Implicit Grant

### Endpoints

- `/oauth2/authorize`

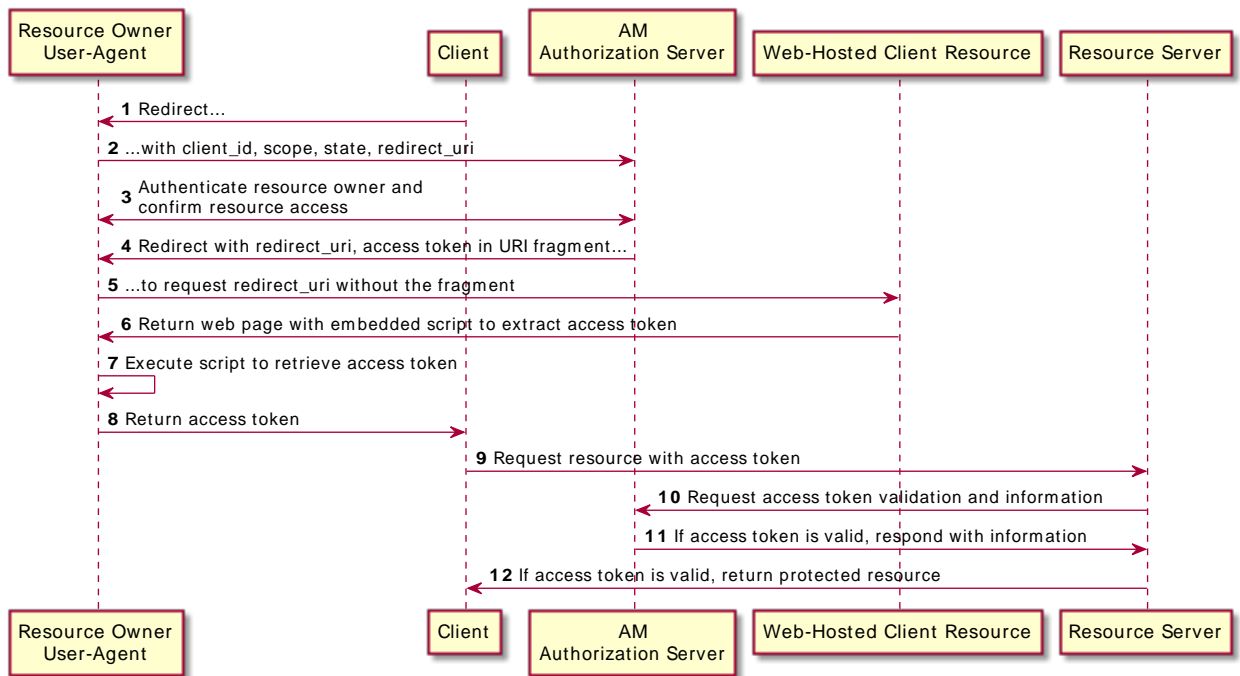
The Implicit grant is designed for public clients that run inside the resource owner's user-agent, for example, JavaScript applications.

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.

The following diagram demonstrates the Implicit grant flow:

### OAuth 2.0 Implicit Grant Flow



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 access to 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 in the top-level realm.

For more information, see "Configuring the OAuth 2.0 Provider Service".

- 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 "Registering OAuth 2.0 Clients With the OAuth 2.0 Provider Service".

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 `/customers` realm, then use `/oauth2/realms/root/realms/customers/authorize`.

For example:

```
https://openam.example.com:8443/openam/oauth2/realms/root/authorize \
?client_id=myClient \
&response_type=token \
&scope=write \
&redirect_uri=https://www.example.com:443/callback \
&state=abc123
```

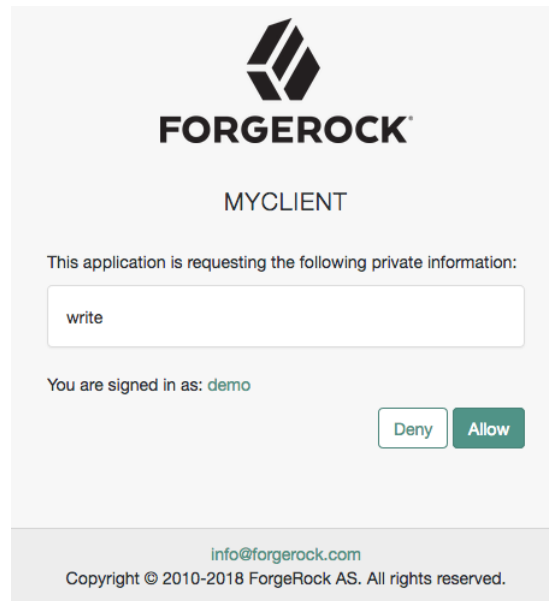
Note that the URL is split for readability purposes and that the `scope` and `state` parameters have been included. Scopes are not required, since they can be configured by default in the authorization server and the client, and have been added only as an example. The `state` parameter is added to protect 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 in the top-level realm, and it accepts the **write** scope.

For more information, see "Configuring the OAuth 2.0 Provider Service".

- 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 "Registering OAuth 2.0 Clients With the OAuth 2.0 Provider Service".

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: changeit" \
--header "Accept-API-Version: resource=2.0, protocol=1.0" \
'https://openam.example.com:8443/openam/json/realms/root/authenticate'
{
  "tokenId": "AQIC5wM...TU30Q*",
  "successUrl": "/openam/console",
  "realm": "/"
}
```

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 `/customers` realm, then use `/oauth2/realms/root/realms/customers/authorize`.

For example:

```
curl --dump-header - \  
--Cookie "iPlanetDirectoryPro=AQIC5wM...TU30Q*" \  
--request POST \  
--data "client_id=myClient" \  
--data "response_type=token" \  
--data "scope=write" \  
--data "state=123abc" \  
--data "decision=allow" \  
--data "csrf=AQIC5wM...TU30Q*" \  
--data "redirect_uri=https://www.example.com:443/callback" \  
"https://openam.example.com:8443/openam/oauth2/realms/root/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=Pas0dWcVnb5W8uuBT12H62Rvmro&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`

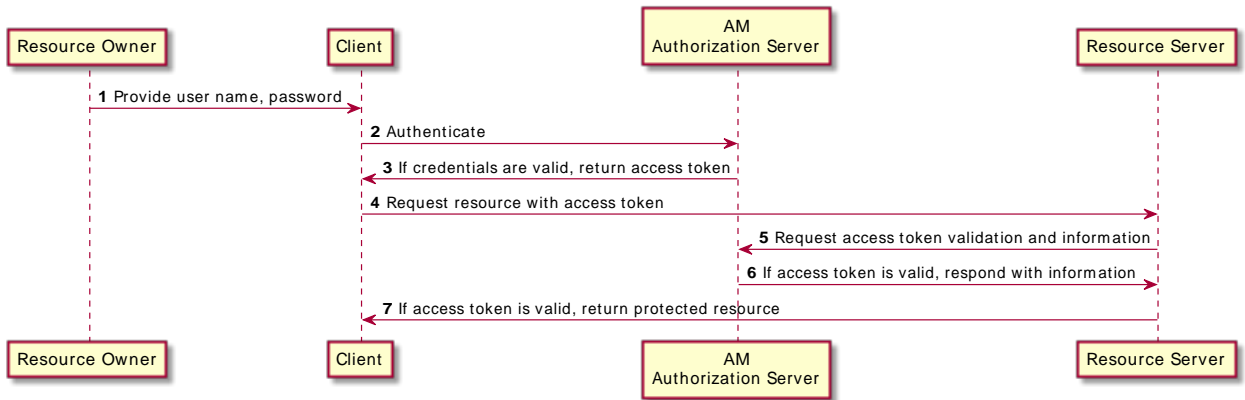
The Resource Owner Password Credentials (ROPC) grant flow allows the client to use the resource owner's user name and password to get an access token.

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).

The following diagram demonstrates the ROPC grant flow:

### OAuth 2.0 Resource Owner Password Credentials Grant Flow



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 access to 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 "Configuring the Default Authentication Tree or Chain" in the *Authentication and Single Sign-On Guide*.

- AM is configured as an OAuth 2.0 authorization server in the top-level realm.

For more information, see "Configuring the OAuth 2.0 Provider Service".

- A confidential client called `myClient` is registered in AM with the following configuration:

- **Client secret:** `forgerock`
- **Scopes:** `write`
- **Response Types:** `token`
- **Grant Types:** `Resource Owner Password Credentials`

For more information, see "Registering OAuth 2.0 Clients With the OAuth 2.0 Provider Service".

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"`.

Private 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 *"Authenticating OAuth 2.0 Clients"*.

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 `/customers` realm, then use `/oauth2/realms/root/realms/customers/access_token`.

For example:

```
$ curl --request POST \  
--data "grant_type=password" \  
--data "username=demo" \  
--data "password=changeit" \  
--data "scope=write" \  
--data "client_id=myClient" \  
--data "client_secret=forgerock" \  
"https://openam.example.com:8443/openam/oauth2/realms/root/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 "[Managing OAuth 2.0 Refresh Tokens](#)".

## Client Credentials Grant

### Endpoints

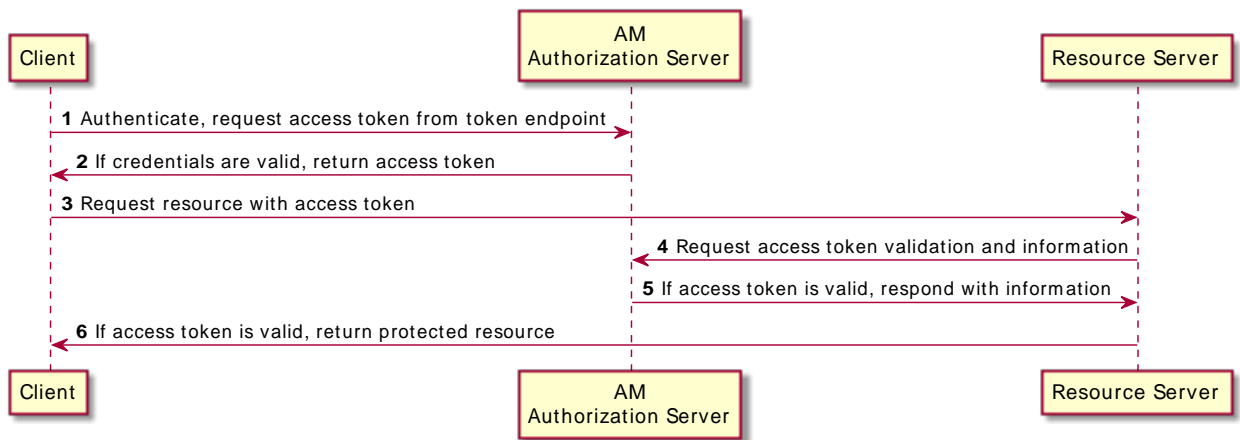
- /oauth2/access\_token

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.

The following diagram demonstrates the Client Credentials grant flow:

*OAuth 2.0 Client Credentials Grant Flow*



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 access to 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 in the top-level realm.  
For more information, see "Configuring the OAuth 2.0 Provider Service".
- A confidential client called `myClient` is registered in AM with the following configuration:
  - **Client secret:** `forgerock`
  - **Scopes:** `write`
  - **Response Types:** `token`
  - **Grant Types:** `Client Credentials`

For more information, see "Registering OAuth 2.0 Clients With the OAuth 2.0 Provider Service".

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`".

Private 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 "Authenticating OAuth 2.0 Clients".

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 `/customers` realm, use `/oauth2/realms/root/realms/customers/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/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": "sbQZuveFumUDV5R1vVB16QAGNB8",
  "scope": "write",
  "token_type": "Bearer",
  "expires_in": 3599
}
```

## Device Flow

### Endpoints

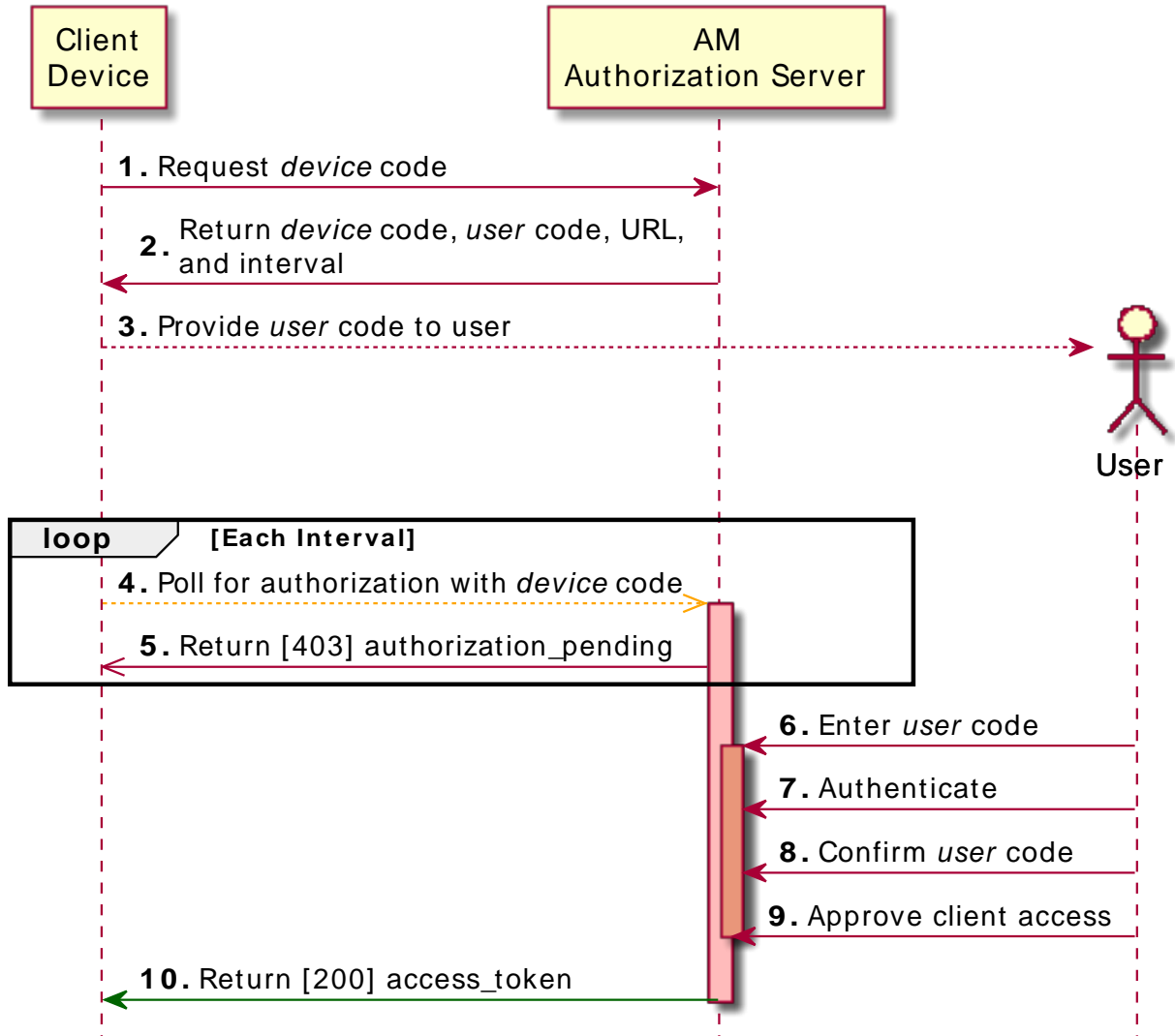
- `/oauth2/device/code`
- `/oauth2/device/user`
- `/oauth2/access_token`

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.

The following diagram demonstrates the Device Flow:

### OAuth 2.0 Device Flow



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 Without Using a Browser in the Device Flow".
- "To Grant Consent with a User Code 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:

```
234567ABCDEFGHijklmnopqrstuvwxyz
```

This procedure assumes the following configuration:

- AM is configured as an OAuth 2.0 authorization server in the top-level realm. scope.

For more information, see "Configuring the OAuth 2.0 Provider Service".

- A public client called `myClient` is registered in AM with the following configuration:

- **Scopes:** `write`
- **Response Types:** `device_code token`
- **Grant Types:** `Device Code`

For more information, see "Registering OAuth 2.0 Clients With the OAuth 2.0 Provider Service".

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 *"Authenticating OAuth 2.0 Clients"*.

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/device/code"
{
  "interval": 5,
  "device_code": "7a95a0a4-6f13-42e3-ac3e-d3d159c94c55...",
  "verification_uri": "https://openam.example.com:8443/openam/oauth2/device/user",
  "verification_url": "https://openam.example.com:8443/openam/oauth2/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.

- 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 Without 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 Using a Browser in the Device Flow".
- 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:

- 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: changeit" \
--header "Accept-API-Version: resource=2.0, protocol=1.0" \
'https://openam.example.com:8443/openam/json/realms/root/authenticate'
{
  "tokenId": "AQIC5wM...TU30Q*",
  "successUrl": "/openam/console",
  "realm": "/"
}
```

- 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...TU30Q*" \
--data "user_code=VAL12e0v" \
--data "decision=allow" \
--data "csrf=AQIC5wM...TU30Q*" \
"https://openam.example.com:8443/openam/oauth2/realms/root/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 : "/",
  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 : "/",
  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.

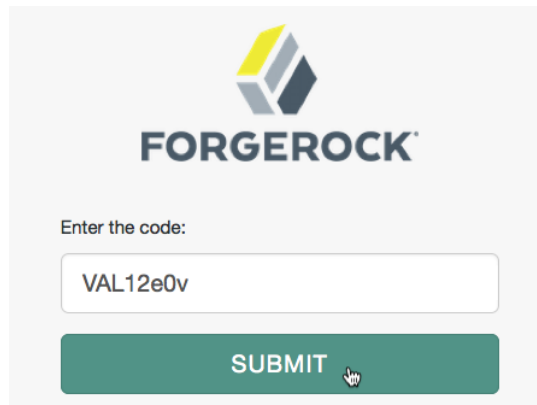
### *To Grant Consent with a User Code Using a Browser in the Device Flow*

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/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*



FORGEROCK

Enter the code:

SUBMIT

#### **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

**FORGEROCK**

MYCLIENT

This application is requesting the following private information:

write

You are signed in as: demo

Deny Allow

info@forgerock.com  
Copyright © 2010-2018 ForgeRock AS. All rights reserved.

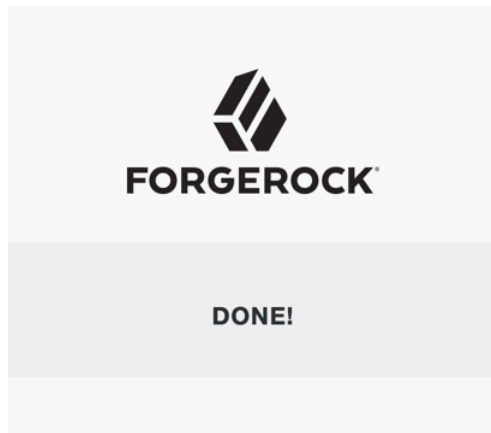
#### Note

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/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 "[Managing OAuth 2.0 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"}
```

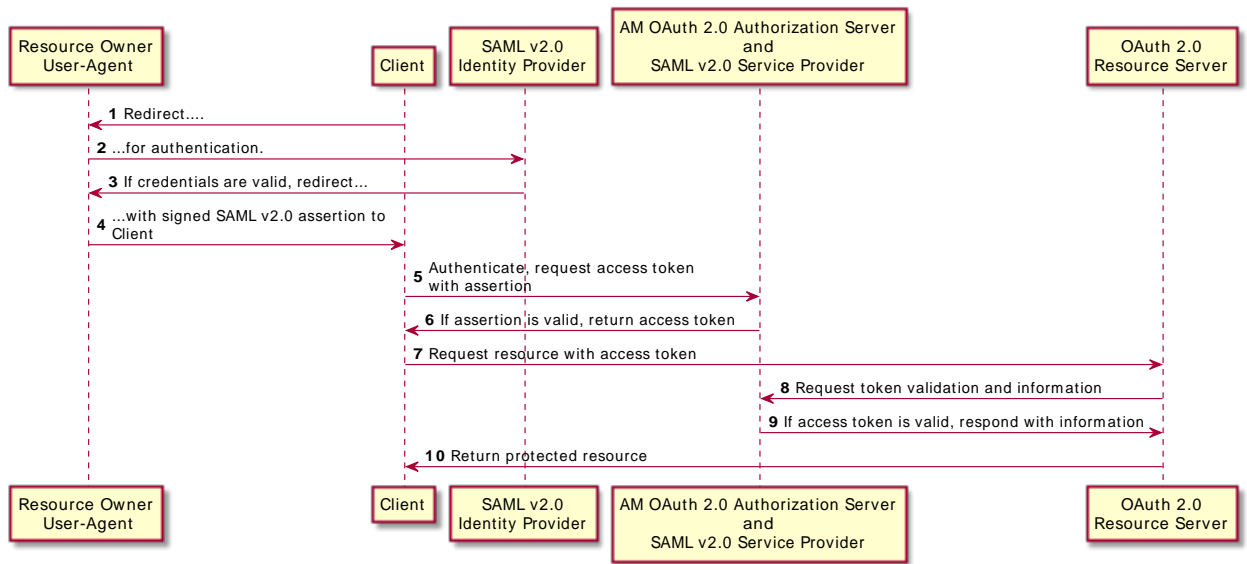
AM may fail to determine the resource owner if the assertion contains an opaque name ID during transient federation. Because the opaque reference is never stored during a transient flow, the OAuth 2.0 provider cannot determine the resource owner it relates to.

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:
  - **Response Types:** `token`
  - **Grant Types:** `SAML2`

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_password=forgerock" \
--data-urlencode "assertion=PHNhbWxwO1...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/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 access to 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

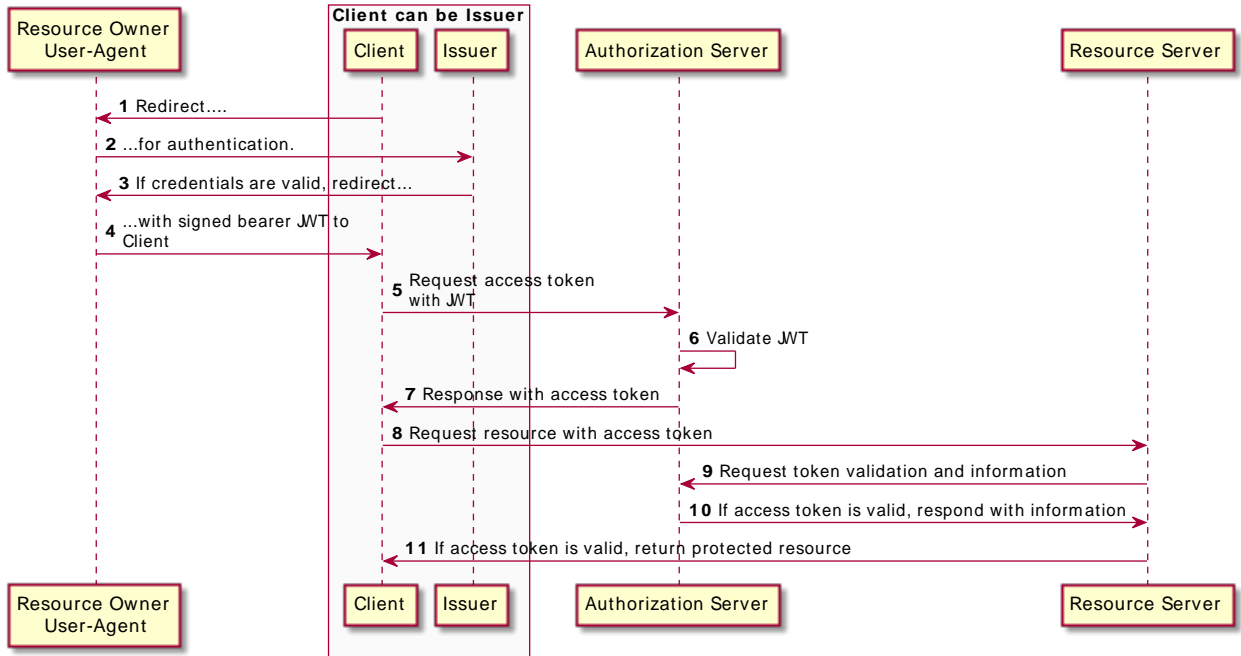
The *RFC 7523* defines the use of JWT bearer tokens for both requesting access tokens as well as for client authentication.

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 a special type of agent.

The following diagram demonstrates the JWT Bearer Profile for Authorization Grant flow:

### 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_username` 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=eyJWxnIjogIlJTMjU2IiB9.eyJhc3ViIjogImp3..." \
--data "redirect_uri=http://www.example.com" \
--data "scope=write" \
"https://openam.example.com:8443/openam/oauth2/realms/root/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 "*Authenticating OAuth 2.0 Clients*".

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 access to 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. Log in to the AM console as an administrative user, for example, `amAdmin`.
2. Navigate to Realms > *Realm Name* > Applications > Agents > Trusted JWT Issuer.
3. Add a new trusted JWT issuer agent.
4. 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.

5. 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](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-rd0mi5lC3pn3y5sTgYiLLFVFY7XxDLinWneHEaAXA",
    "y": "mxmqquaiq44INGyyPP2vATt3IKDL_6W5CAcfAMSZl8k",
    "kid": "signing_key",
    "x5c": [
      "MIIBSjCB76ADAgEC...955PByPrfLZkQ0C/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.

6. (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.

## 7. Save your changes.

The trusted JWT issuer agent is ready for use.

## Chapter 5

# Managing OAuth 2.0 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.

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 allowing the clients to 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 allows you to 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 "OAuth2 Provider" and "OAuth 2.0 and OpenID Connect 1.0 Client Settings".

Refresh tokens can be revoked. For more information, see ["/oauth2/token/revoke"](/oauth2/token/revoke)).

This section contains the following procedures:

- "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 top-level realm with the following configuration:
  - **Issue Refresh Tokens** is enabled.
  - **Issue Refresh Tokens on Refreshing Access Tokens** is enabled.

For more information, see "Configuring the OAuth 2.0 Provider Service".

- A confidential client called **myClient** is registered in AM with the following configuration:
  - **Client secret:** `forgerock`
  - **Scopes:** `write read`
  - **Response Types:** `token`
  - **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-9AY0kRp3AWcCZULvPitpM`.
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`.

Private 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 *"Authenticating OAuth 2.0 Clients"*.

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 `/customers` realm, then use `/oauth2/realms/root/realms/customers/access_token`.

For example:

```
$ curl --request POST \
--data "grant_type=refresh_token" \
--data "refresh_token=qz1qx-9AY0kRp3AWcCZULvPitpM" \
--data "client_id=myClient" \
--data "client_secret=forgerock" \
--data "scope=read" \
"https://openam.example.com:8443/openam/oauth2/realms/root/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 6

# Implementing OAuth 2.0 Proof-of-Possession

AM supports associating a JSON Web Key (JWK) key or an X.509 certificate with an access token to support proof-of-possession interactions. This allows the presenter of a bearer token to prove that it was originally issued the access token.

AM supports "JWK-Based Proof-of-Possession" and "Certificate-Bound Proof-of-Possession".

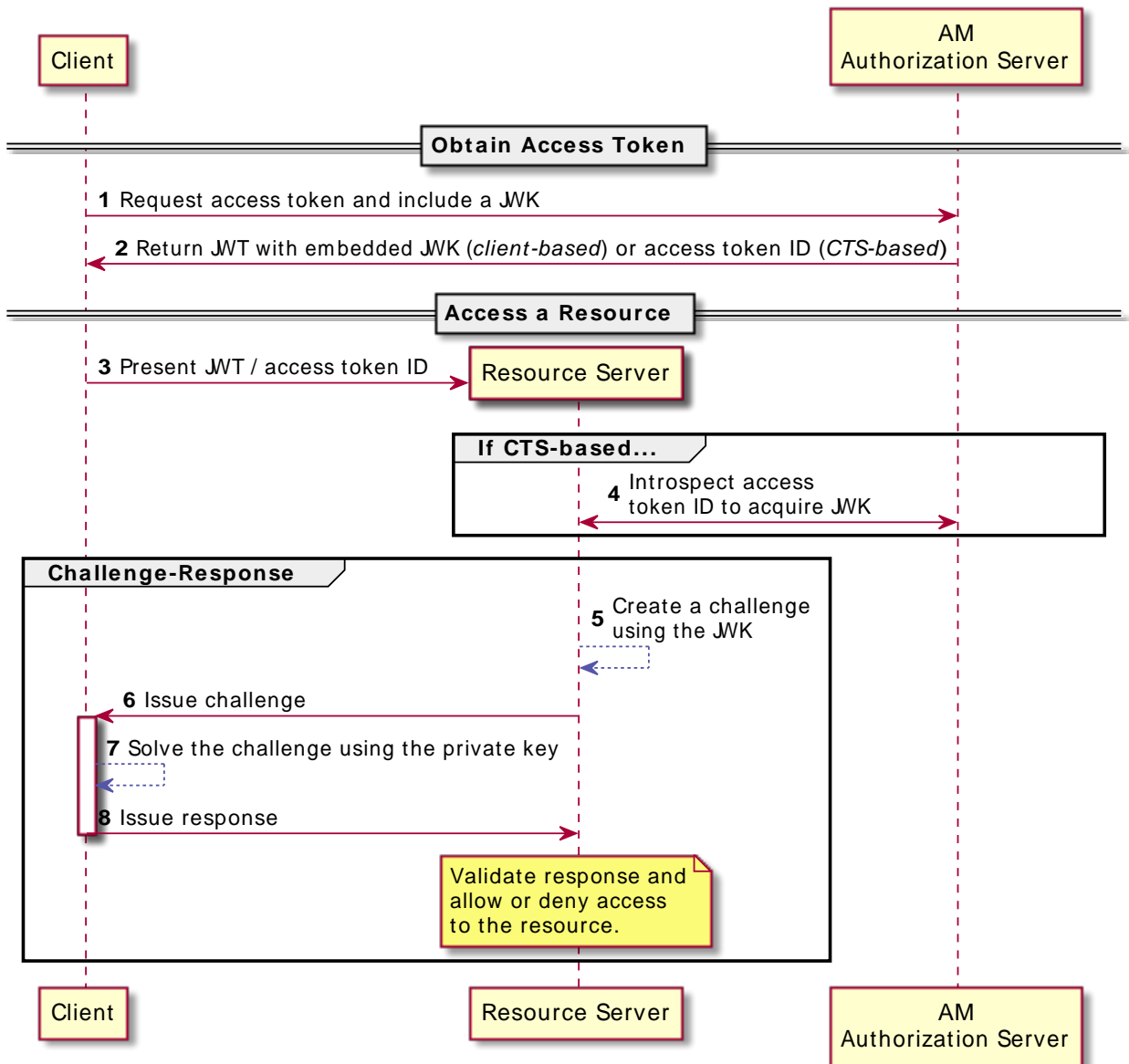
## JWK-Based Proof-of-Possession

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.

The following diagram demonstrates the JWK-based proof-of-possession flow:

### OAuth 2.0 JWK-Based Proof-of-Possession Flow



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, the authorization server 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*

Perform the steps in this procedure to obtain an access token using OAuth 2.0 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.

- Base64-encode the JWK. For example:

```
ew0KICAgICJKV0si0iB7DQogICAgICAgICJhbGciOiAiU1MyNTYiLA0KICAgICAgICAIzSI6IC
JBUUFUIiwNDQogICAgICAgICJraWQiOiAiAibXlQdWJsaWNKU090V2ViS2V5Ig0KICAgIH0NCn0=
```

- 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`

Private 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 "[Authenticating OAuth 2.0 Clients](#)".

For example:

```
$ curl \
--request POST \
--data "grant_type=client_credentials" \
--data "client_id=myClient" \
--data "client_secret=forgerock" \
--data "cnf_key=ew0KICAgICJKV0si0iB7DQogICAgICAgICJhbGciOiAiU1MyNTYiLA0KICAgICAgICAIzSI6IC
JBUUFUIiwNDQogICAgICAgICJraWQiOiAiAibXlQdWJsaWNKU090V2ViS2V5Ig0KICAgIH0NCn0=" \
"https://openam.example.com:8443/openam/oauth2/realms/root/access_token"
```

For more information about how to use the different OAuth 2.0 grant flows, see "[Implementing 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": "eyJ0eXAiOiJKV1QiLCJi51zbE3t...zc2NjI3NDgsInNjb3zU0CVKXCX0Se0",
  "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/introspect"
{
  "active": true,
  "scope": "profile",
  "client_id": "myClient",
  "user_id": "myClient",
  "username": "myClient",
  "token_type": "access_token",
  "exp": 1477666348,
  "sub": "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
  }
}
    
```

- 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

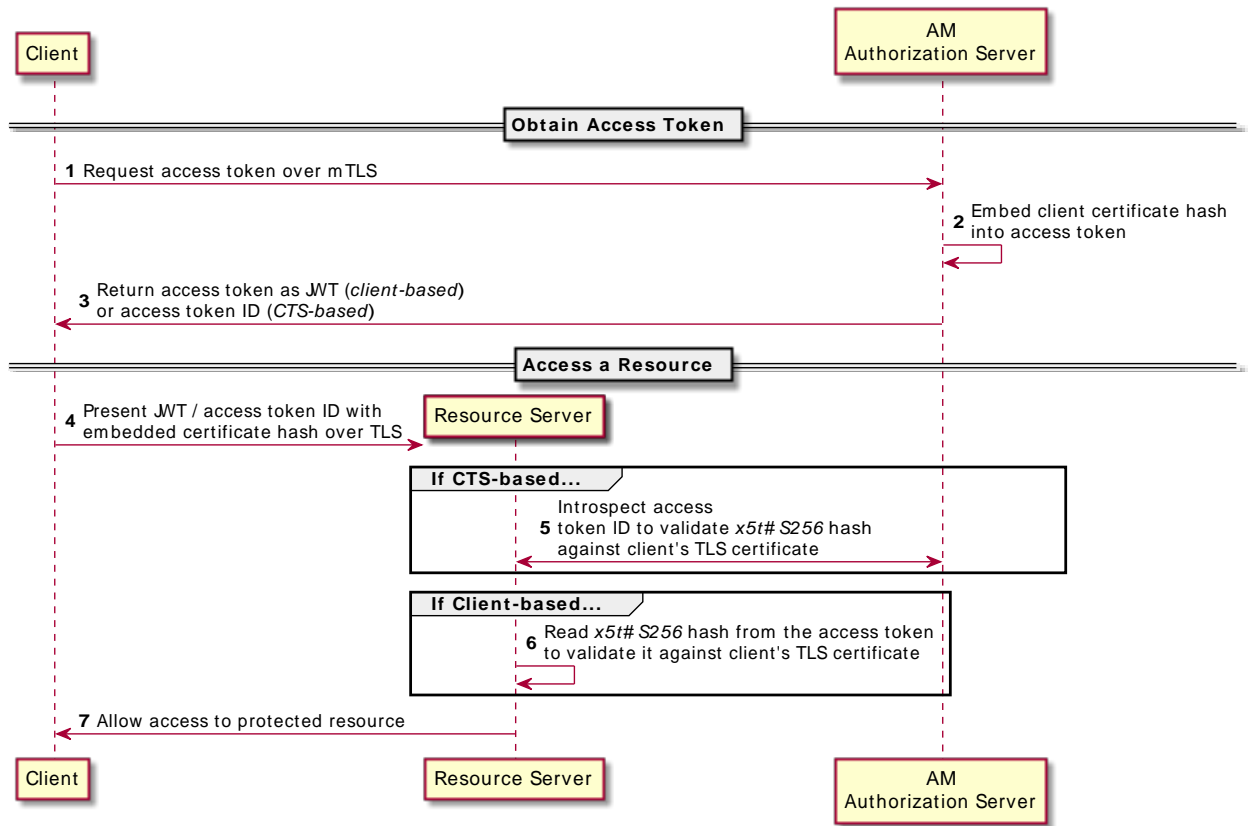
AM supports associating an X.509 certificate with an access token to support proof-of-possession interactions, as per version 12 of the [OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens internet-draft](#).

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.

The following diagram demonstrates the certificate-bound proof-of-possession flow:

### OAuth 2.0 Certificate-Bound Proof-of-Possession Flow



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:

- The server must accept the scopes that are relevant for your environment. The `write` scope will be used in the examples of this procedure.
- 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`
- **Response Types:** `token`
- **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 \
--cert myClientCertificate.pem \
--key myClientCertificate.key.pem \
"https://openam.example.com:8443/openam/oauth2/realms/root/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": "eyJ0eXAiOiJKV1QiLCJhbGciOiJI1zbtE3t...zc2NjI3NDgsInNjb3zU0CVKcX0Se0",
  "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/introspect"
{
  "active":true,
  "scope":"write",
  "client_id":"myClient",
  "user_id":"myClient",
  "username":"myClient",
  "token_type":"Bearer",
  "exp":1547079953,
  "sub":"myClient",
  "iss":"https://openam.example.com:8443/openam/oauth2",
  "cnf":{"
    "x5t#S256":"m8UcWBSPNtaKN19TdR8zUHvWW0SCSX9nsa5vU6fscd0"
  }
}
```

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": "0AUTH2_STATELESS_GRANT",
  ....
  "cnf": {
    "x5t#S256": "m8UcWBSPNtaKN19TdR8zUHvWW0SCSX9nsa5vU6fscd0"
  },
  "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:

- The server must accept the scopes that are relevant for your environment. The `write` scope will be used in the examples of this procedure.
- 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 when 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`
- **Response Types:** `token`
- **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.

- 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/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:

```
m8UcWBSPNtaKN19TdR8zUHvWW0SCSX9nsa5vU6fscd0
```

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

```
{"x5t#S256": "m8UcWBSPNtaKN19TdR8zUHvWW0SCSX9nsa5vU6fscd0"}
```

- 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=eyJ4NXQjUzI1NiI6Im04VWNXQlNQtnRhS04xOVRkUjh6VUh2V1dPU0NTWdluc2E1dLU2ZnNjZDAifQ==" \
"https://openam.example.com:8443/openam/oauth2/realms/root/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": "eyJ0eXAiOiJKV1QiLCJi51zbE3t...zc2NjI3NDgsInNjb3ZU0CVKXC0Se0",
  "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/introspect"
{
  "active": true,
  "scope": "write",
  "client_id": "myClient",
  "user_id": "myClient",
  "username": "myClient",
  "token_type": "Bearer",
  "exp": 1547079953,
  "sub": "myClient",
  "iss": "https://openam.example.com:8443/openam/oauth2",
  "cnf": {
    "x5t#S256": "m8UcWBSPNtaKN19TdR8zUHVvW0SCSX9nsa5vU6fscd0"
  }
}
```

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": "m8UcWBSNPtaKN19TdR8zUHvVW0SCSX9nsa5vU6fscd0"
    },
    "exp": 1547083590,
    "iat": 1547079990,
    "expires_in": 3600,
    "jti": "sLzkRiayAQKsrXN0Gu_vwFog3Rs"
  }
}
```

## Chapter 7

# Managing OAuth 2.0 Consent

Most 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.

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 "OAuth 2.0 Remote Consent Service".

AM allows clients to 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 allows users to 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.

This chapter includes the following sections:

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

## 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. Log in to the AM console with an administrative user. For example, `amAdmin`.

2. 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.
3. 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 and Maintenance Guide*.

2. Log in to the AM console with an administrative user. For example, `amAdmin`.
3. Navigate to Realms > *Realm Name* > Services > OAuth 2.0 provider > Consent.
4. In the Saved Consent Attribute field, add the name of the attribute you created in the identity store.
5. 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

#### Authorized Apps

APPLICATION	SCOPES	EXPIRES	
Example Client	Access to your data, Ability to share yo...	08/05/2019, 16:58:26	<b>x</b>

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

## OAuth 2.0 Remote Consent Service

AM supports OAuth 2.0 remote consent services (RCS), 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* JSON Web Token (JWT) that contains the necessary information to render a consent gathering page.

The consent request JWT contains the following properties:

**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 following is an example of the consent JWT:

```
{
  "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=I234zy",
  "username": "demo"
}
```

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

A 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.

**Note**

AM sends only the information required to create a consent gathering page to the remote consent service. It does not send the actual values of the requested scopes.

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

The consent response JWT contains the following properties:

**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. The response JWT must be sent as a `consent_response` form parameter in a POST operation to this URI.

The following is an example of the consent response JWT:

```
{
  "consent_response" : {
    "clientId": "myClient",
    "iss": "rcs",
    "csrf": "gjeH2C43nFJwW+IrlzL3hl8kux9oatSZRso7aCzI0vk=",
    "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=I234zy",
    "username": "demo",
    "decision": true
  },
}
```

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 *Installation Guide*.

`{am_abbr}` and the remote consent service make their required public keys available from two `jwt_` `uris`, to enable signing and encryption between the two servers.

The OAuth 2.0 flow continues as if AM had gathered the consent itself.

## Configuring Remote Consent Services

This section describes how to configure AM to use a Remote Consent Service (RCS). It also demonstrates use of the built-in sample remote consent service.

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

1. 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 `jwt_uri` URI details of the remote consent service.

For details, see "To Configure AM to use a Remote Consent Service".

2. Enable remote consent and specify the agent profile in AM's OAuth 2.0 provider service.

For details, see "To Configure the OAuth 2.0 Provider to Use a Remote Consent Agent Profile".

3. Configure the remote consent service with AM's `jwt_uri` URI details, so that it can obtain signature and decryption keys.

AM includes an example remote consent service. For details, 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, select Realms, and then select the realm that you are working with.
2. Navigate to Applications > Remote Consent and select Add Remote Consent Agent.
3. Enter an Agent ID, for example `myRCSAgent`, and then select Create.
4. (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.

5. Select the Remote Consent Agent, and then configure the properties as required.

For information on the available properties, see "OAuth 2.0 Remote Consent Agent Settings".

6. 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, select Realms, and then select the realm that you are working with.
2. Navigate to Services, and then select OAuth2 Provider.

#### **Tip**

If you have not yet configured an OAuth 2.0 provider, follow the steps in "To Set Up the OAuth 2.0 Provider Service".

3. 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`.
4. (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".

The result may resemble the following:

## Configuring RCS in an OAuth 2.0 Provider

SERVICE

OAuth2 Provider

✕ Delete

OpenID Connect
Advanced OpenID Connect
Device Flow
Consent

**Saved Consent Attribute Name**

**Allow Clients to Skip Consent**

**Enable Remote Consent**

**Remote Consent Service ID**

**Remote Consent Service Request Signing Algorithms Supported**

**Remote Consent Service Request Encryption Algorithms Supported**

**Remote Consent Service Request Encryption Methods Supported**

**Remote Consent Service Response Signing Algorithms Supported**

**Remote Consent Service Response Encryption Algorithms Supported**

**Remote Consent Service Response Encryption Methods Supported**

Save Changes

### 5. 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 to demonstrate and test AM's remote consent feature.

OAuth 2.0 Guide ForgeRock Access Management 6.5 (2022-10-11)  
Copyright © 2011-2022 ForgeRock AS. All rights reserved.

129

**Note**

The Remote Consent Service in AM is not intended for use in production environments, for example the encryption and signing algorithms are not configurable. It serves as an example of configuring 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.example.com:8443/openam>.

Perform the following steps to configure your environment:

1. Log in to the instance that acts as the example remote consent service with an administrative user, for example, `amAdmin`.
2. Navigate to Realms > *Realm Name* > Services, and then select 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 Signing Key Alias, enter the alias of key that will sign the consent response. Ensure the selected key matches the supported signing methods and algorithms configured for the remote consent service in the OAuth 2.0 provider in AM.  
For this example, enter `rsajwt signingkey`. This test key alias will work with the default signing settings, and is provided by default in AM's default key store.
  - c. In Encryption Key Alias, enter the alias of key that will encrypt the consent response. Ensure the selected key matches the supported encryption methods and algorithms configured for the remote consent service in the OAuth 2.0 provider in AM.  
For this example, enter `test`. This test key alias will work with the default encryption settings, and is provided by default in AM's default key store.
  - d. In Authorization Server `jwt_uri`, 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`.

- e. Select Create.
  - f. Verify the configuration. For more information about the available properties, see "Remote Consent Service".
5. Log in to the instance acting as the authorization server as an administrative user, for example, `amAdmin`.
  6. Configure a remote consent service agent by performing the steps in "To Configure AM to use a Remote Consent Service".

#### Note

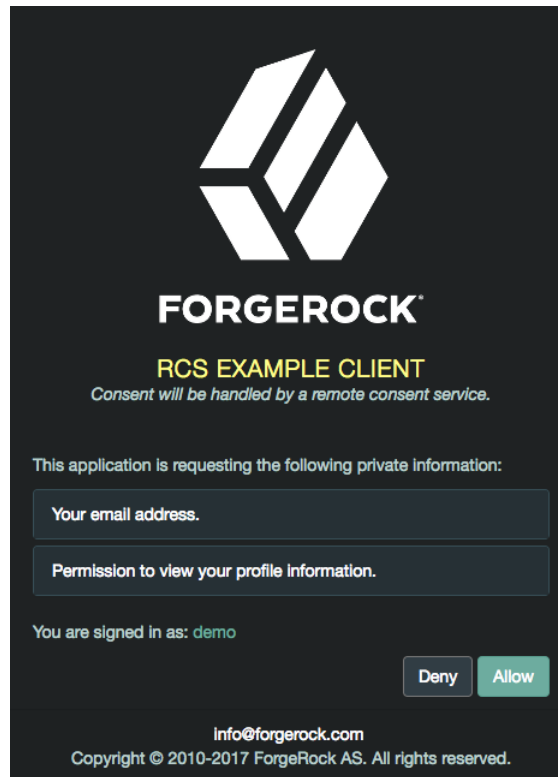
The example remote consent service provides an `/oauth2/consent/jwk_uri` path to provide its public keys to the authorization server. In this example, configure `https://rcs.example.com:8443/openam/oauth2/consent/jwk_uri` in the Json Web Key URI field.

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 the remote consent service for the purpose of this demonstration.

For more information on customizing the user interface, see "*Customizing the User Interface*" in the *UI Customization Guide*.

## Chapter 8

# OAuth 2.0 Endpoints

When acting as an OAuth 2.0 authorization server, AM exposes the following endpoints:

### *OAuth 2.0 Endpoints*

Endpoint	Description
<code>/oauth2/authorize</code>	Obtain consent and an authorization grant (RFC 6749 authorization endpoint)
<code>/oauth2/bc-authorize</code>	Initiate backchannel authorization (Backchannel flow endpoint)
<code>/oauth2/access_token</code>	Obtain an access token (RFC 6749 token endpoint)
<code>/oauth2/device/code</code>	Obtain a device code (Device flow endpoint)
<code>/oauth2/device/user</code>	Obtain consent and authorization grant (Device flow endpoint)
<code>/oauth2/token/revoke</code>	Revoke both access and refresh tokens (RFC 7009 endpoint)
<code>/oauth2/introspect</code>	Retrieve metadata about a token, such as approved scopes and the context in which the token was issued (RFC 7662 endpoint)

#### Tip

As an OAuth 2.0/OpenID Connect 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*.

## `/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)

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/subrealm1/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.
- `token`, to request an access token.
- `id_token`, to request an ID token.
- `code token`, to request an authorization code and an access token.
- `token id_token`, to request an access token and an ID token.
- `code id_token`, to request an authorization code and an ID token.
- `code token id_token`, to request an authorization code, an access token, and an ID token.
- `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".

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 "Retrieving Session State without the Check Session Endpoint" 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 "Configuring for 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/myRealm",
  "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.

#### Note

AM does not support non-string JWT header parameters, such as `jku` and `jwe`:

- AM 6.5.2 and later parse the JWT but ignores the non-string header parameters.
- AM versions earlier than 6.5.2 do not parse the JWT, and will return an HTTP 500 error message.

Configure the public keys/certificates in AM instead of adding the headers, as explained in the relevant sections of the documentation.

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 *Installation 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*.

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/subrealm1/bc-authorize>.

The backchannel authorization 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)
```

The endpoint also supports a number of form parameters for client authentication, as follows:

**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 "*Authenticating OAuth 2.0 Clients*".

**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 "*Authenticating OAuth 2.0 Clients*".

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:ietf:params:oauth:client-assertion-type:jwt-bearer`.

For more information, see "*Authenticating OAuth 2.0 Clients*".

Required: Yes, when using the JWT bearer client authentication method.

## /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)

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/subrealm1/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](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 "Test the UMA Grant Flow" in the *User-Managed Access (UMA) 2.0 Guide*.
- `refresh_token`, to refresh an access token. For more information, see "*Managing OAuth 2.0 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"

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 "*Authenticating OAuth 2.0 Clients*".

#### `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 "*Authenticating OAuth 2.0 Clients*".

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:ietf:params:oauth:client-assertion-type:jwt-bearer`.

For more information, see "*Authenticating OAuth 2.0 Clients*".

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.

#### 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 "*Implementing OAuth 2.0 Grant Flows*".

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, 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 "*Managing OAuth 2.0 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).

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/subrealm1/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.

### 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 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 "*Configuring for 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.

## /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/subrealm1/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".

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 "*Authenticating OAuth 2.0 Clients*".

### 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 "*Authenticating OAuth 2.0 Clients*".

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:ietf:params:oauth:client-assertion-type:jwt-bearer`.

For more information, see "*Authenticating OAuth 2.0 Clients*".

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/subrealm1/revoke`.

The following is an example of how to revoke a given token:

```
$ curl --request POST \  
--data "token=xS3UjtuXMu77iNzl2XibpeMLwlg" \  
--data "client_id=myClient" \  
--data "client_secret=forgerock" \  
"https://openam.example.com:8443/openam/oauth2/realms/root/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 client 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 6.5.5 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 "*Authenticating OAuth 2.0 Clients*".

**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 "*Authenticating OAuth 2.0 Clients*".

**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 "*Authenticating OAuth 2.0 Clients*".

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:ietf:params:oauth:client-assertion-type:jwt-bearer`.

For more information, see "*Authenticating OAuth 2.0 Clients*".

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/subrealm1/introspect`.

**Tip**

To allow a client to introspect access tokens issued to other clients in the same realm, set the special scope, `am-introspect-all-tokens`, in the client profile.

The following example shows AM returning token information:

```
$ curl \
--request POST \
--header "Authorization: Basic ZGVtbzpjagFuZ2VpdA==" \
--data "token=f9063e26-3a29-41ec-86de-1d0d68aa85e9"
"https://openam.example.com:8443/openam/oauth2/introspect"
{
  "active": true,
  "scope": "write",
  "client_id": "myClient",
  "user_id": "demo",
  "username": "demo",
  "token_type": "Bearer",
  "exp": 1419356238,
  "sub": "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
  }
}
```

AM returns a JSON object with different properties depending on the type of token. The most common objects are:

#### 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.

#### username

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.

(Introduced in AM 6.5.4) 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 subject of the token.

**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.

**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.

## Legacy OAuth 2.0 endpoints

AM exposes the following legacy endpoints:

## OAuth 2.0 Administration Endpoints

Endpoint	Description
<code>/frrest/oauth2/token</code> (Legacy)	Retrieve metadata about a token, revoke both access and refresh tokens (AM-specific endpoint, legacy)
<code>/oauth2/tokeninfo</code> (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)

The AM-specific OAuth 2.0 token administration endpoint `/frrest/oauth2/token` lets administrators 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 labelled 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 holding tokens granted by specific resource owners, and for deleting tokens for a combination of a resource owner and client.

To list the contents of a specific token, perform an HTTP GET on `/frrest/oauth2/token/token-id` as in the following example:

```
$ curl --request POST \  
--data "grant_type=password" \  
--data "username=demo" \  
--data "password=changeit" \  
--data "scope=cn" \  
--data "client_id=myClient" \  
--data "client_secret=forgerock" \  
"https://openam.example.com:8443/openam/oauth2/realms/root/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": [
    "/"
  ],
  "userName": [
    "demo"
  ]
}
```

To list the tokens for the current user, perform an HTTP GET on `/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 "ipPlanetDirectoryPro: 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": [
  "/test"
],
"userName": [
  "user.4"
],
"display_name": "",
"scopes": "openid"
},
{
  "_rev": "1753454107",
  "tokenName": [
    "access_token"
  ],
  "expireTime": "Indefinitely",
  "scope": [
    "openid"
  ],
  "grant_type": [
    "password"
  ],
  "clientID": [
    "myClientID"
  ],
  "tokenType": [
    "Bearer"
  ],
  "redirectURI": [],
  "nonce": [],
  "realm": [
    "/test"
  ],
  "userName": [
    "user.4"
  ],
  "display_name": "",
  "scopes": "openid"
}
},
"resultCount": 2,
"pagedResultsCookie": null,
"totalPagedResultsPolicy": "NONE",
"totalPagedResults": -1,
"remainingPagedResults": -1
}
```

To list a specific user's tokens, perform an HTTP GET on `/frrest/oauth2/token/?_queryId=userName=string`, where *string* is the user, such as `user.4`. 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=user.4"
{
  "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": [
        "/test"
      ],
      "userName": [
        "user.4"
      ],
      "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": [
  "/test"
],
"userName": [
  "user.4"
],
"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=changeit" \  
--data "scope=cn" \  
--data "client_id=myClient" \  
--data "client_secret=forgerock" \  
"https://openam.example.com:8443/openam/oauth2/realms/root/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 labelled 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=changeit" \  
"
```

```
--data "scope=write" \  
--data "client_id=myClient" \  
--data "client_secret=forgerock" \  
"https://openam.example.com:8443/openam/oauth2/realms/root/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": "/",  
  "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.

(Introduced in AM 6.5.4) 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 than the resource owner's.

### `client_id`

Specifies the client that requested the token.

## Chapter 9

# 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
/oauth2/register	Register, read, and delete OAuth 2.0 clients (RFC7592 and >RFC7591)
/json/realms-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)

## /oauth2/register

Endpoint that allows OAuth 2.0/OpenID Connect clients to dynamically register as per RFC7591 and OpenID Connect Dynamic Client Registration 1.0 incorporating errata set 1. The endpoint also allows clients to read and update their metadata, as well as deprovision themselves as per RFC7592.

You must compose the path to the register endpoint, addressing the specific realm where the client is or should be registered/deprovisioned. For example, <https://openam.example.com:8443/openam/oauth2/realms/root/realms/subrealm1/register>.

By default, AM requires clients to present an access token to register themselves. This behavior is controlled by the Allow Open Dynamic Client Registration switch, under the Dynamic Registration tab of the OAuth 2.0 provider.

Read, update, and delete operations require an authorization bearer header that includes the `registration_access_token`, provided to the client during registration.

The endpoint supports the following methods:

- **POST**. Register clients

- **GET.** Read client information
- **PUT.** Update client information
- **DELETE.** Deprovision a client

For usage information and examples, see the following links:

- "Dynamic Client Registration"
- "Dynamic Client Registration Management"

## /json/realm-config/agents/OAuth2Client

AM-specific endpoint that allows AM and agent administrators to 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 navigate to API Explorer > /realm-config > /agents > /OAuth2Client.

The following example shows how to create a basic OAuth 2.0 client named `myClient` in a subrealm of the top-level realm named `subrealm1`. 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"
      },
      "redirectionUri":{
        "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/subrealm1/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 subrealm of the top-level realm named `subrealm1`. 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...3MTYx0A..*" \
  "https://openam.example.com:8443/openam/json/realms/root/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
  }
}
```

## /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, select the Help icon, and then navigate to API Explorer > /users > /{user} > /oauth2 > /resources > /sets.

The following example shows how to read an OAuth 2.0 resource and related policy in the top-level 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/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`

You must compose the path to the resource sets endpoint addressing the specific realm where the resource is registered. For example, <https://openam.example.com:8443/openam/oauth2/realms/root/realms/subrealm1/resources/sets>.

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 navigate to API Explorer > /users > /{user} > /oauth2 > /applications.

To call the endpoint, you must compose the path to the realm where the client is registered. For example, <https://openam.example.com:8443/openam/oauth2/realms/root/realms/subrealm1/applications>.

The following example shows how to list all the clients holding tokens granted in the top-level 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/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,
      "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 top-level 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/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,
  "scopes": {
    "write": "write"
  },
  "expiryDateTime": "2018-11-14T10:48:55.395Z",
  "logoUri": null
}
```

## Chapter 10

# Customizing OAuth 2.0

This chapter covers customizing AM's support for OAuth 2.0.

## 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.

## Designing an OAuth 2.0 Scope Validator Plugin

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.

The following plugin, taken from the `openam-scope-sample` example, sets whether `read` and `write` permissions were granted.

```
/*
 * The contents of this file are subject to the terms of the Common Development and
 * Distribution License (the License). You may not use this file except in compliance with the
 * License.
 *
 * You can obtain a copy of the License at legal/CDDLv1.0.txt. See the License for the
 * specific language governing permission and limitations under the License.
 *
 * When distributing Covered Software, include this CDDL Header Notice in each file and include
```

```

* the License file at legal/CDDLv1.0.txt. If applicable, add the following below the CDDL
* Header, with the fields enclosed by brackets [] replaced by your own identifying
* information: "Portions copyright [year] [name of copyright owner]".
*
* Copyright 2014-2019 ForgeRock AS. All Rights Reserved
*/
package org.forgerock.openam.examples;

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

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;

/**
 * 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 request) throws ServerException {
        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) throws ServerException {
        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;
    }

    @Override

```



```

public Set<String> validateBackChannelAuthorizationScope(
    ClientRegistration clientRegistration,
    Set<String> requestedScopes,
    OAuth2Request request) throws ServerException {

    if (requestedScopes == null || requestedScopes.isEmpty()) {
        return clientRegistration.getDefaultScopes();
    }

    Set<String> scopes = new HashSet<>(clientRegistration.getAllowedScopes());
    scopes.retainAll(requestedScopes);
    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(
    ClientRegistration clientRegistration,
    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
}

@Override
public void modifyAccessToken(AccessToken accessToken, OAuth2Request request) {
}
}
```

## Building the OAuth 2.0 Scope Validator Sample Plugin

For information on downloading and building AM sample source code, see [How do I access and build the sample code provided for OpenAM 12.x, 13.x and AM \(All versions\)?](#) in the *Knowledge Base*.

Get a local clone so that you can try the sample on your system. In the sources, you find the following files:

### **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 "Designing an OAuth 2.0 Scope Validator Plugin" for a listing.

After you successfully build the project, you find the **openam-scope-sample-6.5.5.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/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/tokeninfo\
?access_token=0d492486-11a7-4175-b116-2fc1cbff6d78
{
  "scope": [
    "read"
  ],
  "grant_type": "client_credentials",
  "realm": "/",
  "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.

## 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 both client-based and CTS-based access tokens, and are stored permanently in either the issued JWT or in the CTS respectively.

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 navigate to Realms > Top Level Realm > Scripts, and then select OAuth2 Access Token Modification Script.

For general information about scripting in AM, see "*About Scripting*".

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 then update the Custom Properties field.

#### **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:

```
[  
  "profile",  
  "friends"  
]
```

**scriptName**

The display name of the script. Always present.

The scripts also have access to the following APIs:

- "Global Scripting API Functionality"
- "AccessToken" interface in the *AM 6.5.5 Public API Javadoc*
- "AMIdentity" interface in the *AM 6.5.5 Public API Javadoc*
- "SSOToken" interface in the *AM 6.5.5 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 6.5.5 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".

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.

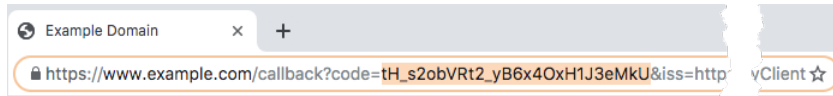


2. Log in as the `demo` user, with password `changeit`.

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_yB6x40xH1J3eMkU" \
  --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/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=sbQZuveFumUDV5R1vVB16QAGNB8" \
"https://openam.example.com:8443/openam/oauth2/realms/root/introspect"
{
  "active": true,
  "scope": "access",
  "client_id": "myClient",
  "user_id": "demo",
  "username": "demo",
  "token_type": "Bearer",
  "exp": 1556289970,
  "sub": "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.

## Chapter 11

# Reference

This reference section covers settings and other information relating to OAuth 2.0 support in AM.

## OAuth 2.0 Standards

AM implements the following RFCs, Internet-Drafts, and standards relating to OAuth 2.0:

### OAuth 2.0

RFC 6749: The OAuth 2.0 Authorization Framework

RFC 6750: The OAuth 2.0 Authorization Framework: Bearer Token Usage

RFC 7009: OAuth 2.0 Token Revocation

RFC 7515: JSON Web Signature (JWS)

RFC 7517: JSON Web Key (JWK)

RFC 7518: JSON Web Algorithms (JWA)

RFC 7519: JSON Web Token (JWT)

RFC 7522: Security Assertion Markup Language (SAML) 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants

RFC 7523: JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants

RFC 7591: OAuth 2.0 Dynamic Client Registration Protocol

RFC 7636: Proof Key for Code Exchange by OAuth Public Clients

RFC 7662: OAuth 2.0 Token Introspection

RFC 7800: Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)

Internet-Draft: OAuth 2.0 Device Flow for Browserless and Input Constrained Devices

Internet-Draft: OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens

RFC 7592: OAuth 2.0 Dynamic Client Registration Management Protocol

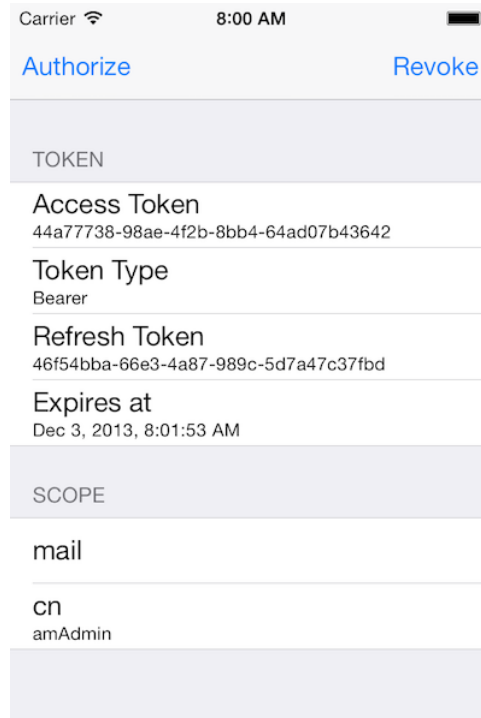
# OAuth 2.0 Sample Mobile Applications

Source code for sample mobile applications is available in sample repositories in the ForgeRock commons project. Get local clones of one or more of the following repositories so that you can try these sample applications on your system:

- AM OAuth2.0 Android sample app
- AM OAuth 2.0 iOS sample app

For example, if you have a Mac running OS X 10.8 or later with Xcode installed, try the AM OAuth 2.0 iOS Sample App.

## *OAuth 2.0 iOS Sample Application*



## OAuth2 Provider

**amster** service name: OAuth2Provider

## Global Attributes

The following settings appear on the **Global Attributes** tab:

### Token Blacklist Cache Size

Number of blacklisted tokens to cache in memory to speed up blacklist checks and reduce load on the CTS.

Default value: `10000`

**amster** attribute: `blacklistCacheSize`

### Blacklist Poll Interval (seconds)

How frequently to poll for token blacklist changes from other servers, in seconds.

How often each server will poll the CTS for token blacklist changes from other servers. This is used to maintain a highly compressed view of the overall current token blacklist improving performance. A lower number will reduce the delay for blacklisted tokens to propagate to all servers at the cost of increased CTS load. Set to 0 to disable this feature completely.

Default value: `60`

**amster** attribute: `blacklistPollInterval`

### Blacklist Purge Delay (minutes)

Length of time to blacklist tokens beyond their expiry time.

Allows additional time to account for clock skew to ensure that a token has expired before it is removed from the blacklist.

Default value: `1`

**amster** attribute: `blacklistPurgeDelay`

### Client-Based Grant Token Upgrade Compatibility Mode

Enable AM to consume and create client-based OAuth 2.0 tokens in two different formats simultaneously.

Enable this option when upgrading AM to allow the new instance to create and consume client-based OAuth 2.0 tokens in both the previous format, and the new format. Disable this option once all AM instances in the cluster have been upgraded.

Default value: `false`

**amster** attribute: `statelessGrantTokenUpgradeCompatibilityMode`

## CTS Storage Scheme

Storage scheme to be used when storing OAuth2 tokens to CTS.

In order to support rolling upgrades, this should be set to the latest storage scheme supported by all AM instances within your cluster. Select the latest storage scheme once all AM instances in the cluster have been upgraded.

### One-to-One Storage Scheme

Under this storage scheme, each OAuth2 token maps to an individual CTS entry.

*This storage scheme is deprecated.*

### Grant-Set Storage Scheme

Under this storage scheme, multiple authorization code, access token and refresh token for a given OAuth2 client and resource owner can be stored within a single CTS entry.

The Grant-Set storage scheme is more efficient than the One-to-One storage scheme so should be used once all servers have been upgraded to a version which supports this storage scheme

The possible values for this property are:

- `CTS_ONE_TO_ONE_MODEL`. One-to-One Storage Scheme
- `CTS_GRANT_SET_MODEL`. Grant-Set Storage Scheme

Default value: `CTS_ONE_TO_ONE_MODEL`

**amster** attribute: `storageScheme`

## Enforce JWT Unreasonable Lifetime

Enable the enforcement of JWT token unreasonable lifetime during validation.

The JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants specification (<https://www.rfc-editor.org/rfc/rfc7523.html#section-3>) states that an authorization server may reject JWTs with an "exp" claim value that is unreasonably far in the future and an "iat" claim value that is unreasonably far in the past. This enforcement may be disabled, but should only be done if the security implications have been evaluated.

Default value: `true`

**amster** attribute: `jwtTokenLifetimeValidationEnabled`

## JWT Unreasonable Lifetime (seconds)

Specify the lifetime (in seconds) of a JWT which should be considered unreasonable and rejected by validation.

The JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants specification (<https://www.rfc-editor.org/rfc/rfc7523.html#section-3>) states that an authorization server may reject JWTs with an "exp" claim value that is unreasonably far in the future and an "iat" claim value that is unreasonably far in the past. During token validation AM enforces that the token must expire within the specified duration and if the "iat" claim value is present, the token must not be older than the specified duration.

Default value: `86400`

**amster** attribute: `jwtTokenUnreasonableLifetime`

## Core

The following settings appear on the **Core** tab:

### Use Client-Based Access & Refresh Tokens

When enabled, AM issues access and refresh tokens that can be inspected by resource servers.

Default value: `false`

**amster** attribute: `statelessTokensEnabled`

### Authorization Code Lifetime (seconds)

The time an authorization code is valid for, in seconds.

Default value: `120`

**amster** attribute: `codeLifetime`

### Refresh Token Lifetime (seconds)

The time in seconds a refresh token is valid for. If this field is set to `-1`, the refresh token will never expire.

Default value: `604800`

**amster** attribute: `refreshTokenLifetime`

### Access Token Lifetime (seconds)

The time an access token is valid for, in seconds. Note that if you set the value to `0`, the access token will not be valid. A maximum lifetime of 600 seconds is recommended.

Default value: `3600`

**amster** attribute: `accessTokenLifetime`

## Issue Refresh Tokens

Whether to issue a refresh token when returning an access token.

Default value: `true`

**amster** attribute: `issueRefreshToken`

## Issue Refresh Tokens on Refreshing Access Tokens

Whether to issue a refresh token when refreshing an access token.

Default value: `true`

**amster** attribute: `issueRefreshTokenOnRefreshedToken`

## Use Policy Engine for Scope decisions

With this setting enabled, the policy engine is consulted for each scope value that is requested.

If a policy returns an action of GRANT=true, the scope is consented automatically, and the user is not consulted in a user-interaction flow. If a policy returns an action of GRANT=false, the scope is not added to any resulting token, and the user will not see it in a user-interaction flow. If no policy returns a value for the GRANT action, then if the grant type is user-facing (i.e. authorization or device code flows), the user is asked for consent (or saved consent is used), and if the grant type is not user-facing (password or client credentials), the scope is not added to any resulting token.

Default value: `false`

**amster** attribute: `usePolicyEngineForScope`

## OAuth2 Access Token Modification Script

The script that is executed when issuing an access token. The script can change the access token's internal data structure to include or exclude particular fields.

The possible values for this property are:

- `d22f9a0c-426a-4466-b95e-d0f125b0d5fa`. OAuth2 Access Token Modification Script
- `[Empty]`. --- Select a script ---

Default value: `d22f9a0c-426a-4466-b95e-d0f125b0d5fa`

**amster** attribute: `accessTokenModificationScript`

## Advanced

The following settings appear on the **Advanced** tab:



## Custom Login URL Template

Custom URL for handling login, to override the default AM login page.

Supports Freemarker syntax, with the following variables:

Variable	Description
<code>gotoUrl</code>	The URL to redirect to after login.
<code>acrValues</code>	The Authentication Context Class Reference (acr) values for the authorization request.
<code>realm</code>	The AM realm the authorization request was made on.
<code>module</code>	The name of the AM authentication module requested to perform resource owner authentication.
<code>service</code>	The name of the AM authentication chain requested to perform resource owner authentication.
<code>locale</code>	A space-separated list of locales, ordered by preference.

The following example template redirects users to a non-AM front end to handle login, which will then redirect back to the `/oauth2/authorize` endpoint with any required parameters:

```
http://mylogin.com/login?goto=${goto}<#if acrValues??>&acr_values=${acrValues}</#if><#if realm??>&realm=${realm}</#if><#if module??>&module=${module}</#if><#if service??>&service=${service}</#if><#if locale??>&locale=${locale}</#if>
```

**NOTE:** Default AM login page is constructed using "Base URL Source" service.

**amster** attribute: `customLoginUrlTemplate`

## Scope Implementation Class

The class that contains the required scope implementation, must implement the `org.forgerock.oauth2.core.ScopeValidator` interface.

Default value: `org.forgerock.openam.oauth2.OpenAMScopeValidator`

**amster** attribute: `scopeImplementationClass`

## Response Type Plugins

List of plugins that handle the valid `response_type` values.

OAuth 2.0 clients pass response types as parameters to the OAuth 2.0 Authorization endpoint (`/oauth2/authorize`) to indicate which grant type is requested from the provider. For example, the client passes `code` when requesting an authorization code, and `token` when requesting an access token.

Values in this list take the form `response-type|plugin-class-name`.

Default value:

```
code|org.forgerock.oauth2.core.AuthorizationCodeResponseTypeHandler
device_code|org.forgerock.oauth2.core.TokenResponseTypeHandler
token|org.forgerock.oauth2.core.TokenResponseTypeHandler
```

**amster** attribute: `responseTypeClasses`

### User Profile Attribute(s) the Resource Owner is Authenticated On

Names of profile attributes that resource owners use to log in. You can add others to the default, for example `mail`.

Default value: `uid`

**amster** attribute: `authenticationAttributes`

### User Display Name attribute

The profile attribute that contains the name to be displayed for the user on the consent page.

Default value: `cn`

**amster** attribute: `displayNameAttribute`

### Supported Scopes

The set of supported scopes, with translations.

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`

Locale strings are in the format: `language_country_variant`, for example `en`, `en_GB`, or `en_US_WIN`.

If the locale and pipe is omitted, the description is displayed to all users that have undefined locales.

If the description is also omitted, nothing is displayed on the consent page for the scope. For example specifying `read|` would allow the scope `read` to be used by the client, but would not display it to the user on the consent page when requested.

**amster** attribute: `supportedScopes`

### Subject Types supported

List of subject types supported. Valid values are:

- `public` - Each client receives the same subject (`sub`) value.
- `pairwise` - Each client receives a different subject (`sub`) value, to prevent correlation between clients.

Default value: `public`

`amster` attribute: `supportedSubjectTypes`

## Default Client Scopes

List of scopes a client will be granted if they request registration without specifying which scopes they want. Default scopes are NOT auto-granted to clients created through the AM console.

`amster` attribute: `defaultScopes`

## OAuth2 Token Signing Algorithm

Algorithm used to sign client-based OAuth 2.0 tokens in order to detect tampering.

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

- `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.

The possible values for this property are:

- `HS256`
- `HS384`
- `HS512`
- `RS256`
- `RS384`
- `RS512`
- `ES256`

- ES384
- ES512
- PS256
- PS384
- PS512

Default value: HS256

**amster** attribute: `tokenSigningAlgorithm`

### Client-Based Token Compression

Whether client-based access and refresh tokens should be compressed.

**amster** attribute: `tokenCompressionEnabled`

### Encrypt Client-Based Tokens

Whether client-based access and refresh tokens should be encrypted.

Enabling token encryption will disable token signing as encryption is performed using direct symmetric encryption.

Default value: `false`

**amster** attribute: `tokenEncryptionEnabled`

### Subject Identifier Hash Salt

If *pairwise* subject types are supported, it is *STRONGLY RECOMMENDED* to change this value. It is used in the salting of hashes for returning specific `sub` claims to individuals using the same `request_uri` or `sector_identifier_uri`.

For example, you might set this property to: *changeme*

**amster** attribute: `hashSalt`

### Code Verifier Parameter Required

If enabled, requests using the authorization code grant require a `code_challenge` attribute.

For more information, read the specification for this feature.

The possible values for this property are:

- `true`. All requests
- `public`. Requests from all public clients

- `passwordless`. Requests from all passwordless public clients
- `false`. No requests

Default value: `false`

**amster** attribute: `codeVerifierEnforced`

### Modified Timestamp Attribute Name

The identity Data Store attribute used to return modified timestamp values.

This attribute is paired together with the *Created Timestamp Attribute Name* attribute (`createdTimestampAttribute`). You can leave both attributes unset (default) or set them both. If you set only one attribute and leave the other blank, the access token fails with a 500 error.

For example, when you configure AM as an OpenID Connect Provider in a Mobile Connect application and use DS as an identity data store, the client accesses the `userinfo` endpoint to obtain the `updated_at` claim value in the ID token. The `updated_at` claim obtains its value from the `modifiedTimestampAttribute` attribute in the user profile. If the profile has never been modified the `updated_at` claim uses the `createdTimestampAttribute` attribute.

**amster** attribute: `modifiedTimestampAttribute`

### Created Timestamp Attribute Name

The identity Data Store attribute used to return created timestamp values.

**amster** attribute: `createdTimestampAttribute`

### Password Grant Authentication Service

The authentication service (chain or tree) that will be used to authenticate the username and password for the resource owner password credentials grant type.

The possible values for this property are:

- `[Empty]`
- `ldapService`
- `amsterService`
- `Example`
- `Agent`
- `RetryLimit`
- `PersistentCookie`
- `HmacOneTimePassword`

- `Facebook-ProvisionIDMAccount`
- `Google-AnonymousUser`
- `Google-DynamicAccountCreation`

**amster** attribute: `passwordGrantAuthService`

## Enable Auth Module Messages for Password Credentials Grant

If enabled, authentication module failure messages are used to create Resource Owner Password Credentials Grant failure messages. If disabled, a standard authentication failed message is used.

The Password Grant Type requires the `grant_type=password` parameter.

Default value: `false`

**amster** attribute: `moduleMessageEnabledInPasswordGrant`

## Grant Types

The set of Grant Types (OAuth2 Flows) that are permitted to be used by this client.

If no Grant Types (OAuth2 Flows) are configured nothing will be permitted.

Default value:

```
implicit
urn:ietf:params:oauth:grant-type:saml2-bearer
refresh_token
password
client_credentials
urn:ietf:params:oauth:grant-type:device_code
authorization_code
urn:openid:params:grant-type:ciba
urn:ietf:params:oauth:grant-type:uma-ticket
urn:ietf:params:oauth:grant-type:jwt-bearer
```

**amster** attribute: `grantTypes`

## Trusted TLS Client Certificate Header

HTTP Header to receive TLS client certificates when TLS is terminated at a proxy.

Leave blank if not terminating TLS at a proxy. Ensure that the proxy is configured to strip this header from incoming requests. Best practice is to use a random string.

**amster** attribute: `tlsClientCertificateTrustedHeader`

## Support TLS Certificate-Bound Access Tokens

Whether to bind access tokens to the client certificate when using TLS client certificate authentication.

Default value: `true`

**amster** attribute: `tlsCertificateBoundAccessTokensEnabled`

## Client Dynamic Registration

The following settings appear on the **Client Dynamic Registration** tab:

### Require Software Statement for Dynamic Client Registration

When enabled, a software statement JWT containing at least the `iss` (issuer) claim must be provided when registering an OAuth 2.0 client dynamically.

Default value: `false`

**amster** attribute: `dynamicClientRegistrationSoftwareStatementRequired`

### Required Software Statement Attested Attributes

The client attributes that are required to be present in the software statement JWT when registering an OAuth 2.0 client dynamically. Only applies if Require Software Statements for Dynamic Client Registration is enabled.

Leave blank to allow any attributes to be present.

Default value: `redirect_uris`

**amster** attribute: `requiredSoftwareStatementAttestedAttributes`

### Allow Open Dynamic Client Registration

Allow clients to register without an access token. If enabled, you should consider adding some form of rate limiting. For more information, see Client Registration in the OpenID Connect specification.

Default value: `false`

**amster** attribute: `allowDynamicRegistration`

### Generate Registration Access Tokens

Whether to generate Registration Access Tokens for clients that register by using open dynamic client registration. Such tokens allow the client to access the Client Configuration Endpoint as per the OpenID Connect specification. This setting has no effect if Allow Open Dynamic Client Registration is disabled.

Default value: `true`

**amster** attribute: `generateRegistrationAccessTokens`

## Scope to give access to dynamic client registration

Mandatory scope required when registering a new OAuth2 client.

Default value: `dynamic_client_registration`

**amster** attribute: `dynamicClientRegistrationScope`

## OpenID Connect

The following settings appear on the **OpenID Connect** tab:

### OIDC Claims Script

The script that is run when issuing an ID token or making a request to the *userinfo* endpoint during OpenID requests.

The script gathers the scopes and populates claims, and has access to the access token, the user's identity and, if available, the user's session.

The possible values for this property are:

- OIDC Claims Script

Default value: `OIDC Claims Script`

**amster** attribute: `oidcClaimsScript`

### ID Token Signing Algorithms supported

Algorithms supported to sign OpenID Connect `id_tokens`.

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

- `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.
- **PS256** - RSASSA-PSS using SHA-256.
- **PS384** - RSASSA-PSS using SHA-384.
- **PS512** - RSASSA-PSS using SHA-512.

Default value:

```
PS384
ES384
RS384
HS256
HS512
ES256
RS256
HS384
ES512
PS256
PS512
RS512
```

**amster** attribute: `supportedIDTokenSigningAlgorithms`

## ID Token Encryption Algorithms supported

Encryption algorithms supported to encrypt OpenID Connect ID tokens in order to hide its contents.

AM supports the following ID token encryption algorithms:

- **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.
- **A128KW** - AES Key Wrapping with 128-bit key derived from the client secret.
- **RSA1\_5** - RSA with PKCS#1 v1.5 padding.
- **A256KW** - AES Key Wrapping with 256-bit key derived from the client secret.
- **dir** - Direct encryption with AES using the hashed client secret.
- **A192KW** - AES Key Wrapping with 192-bit key derived from the client secret.

Default value:

```
RSA-OAEP
RSA-OAEP-256
A128KW
A256KW
```

```
RSA1_5
dir
A192KW
```

**amster** attribute: `supportedIDTokenEncryptionAlgorithms`

## ID Token Encryption Methods supported

Encryption methods supported to encrypt OpenID Connect ID tokens in order to hide its contents.

AM supports the following ID token encryption algorithms:

- `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:

```
A256GCM
A192GCM
A128GCM
A128CBC-HS256
A192CBC-HS384
A256CBC-HS512
```

**amster** attribute: `supportedIDTokenEncryptionMethods`

## Supported Claims

Set of claims supported by the OpenID Connect `/oauth2/userinfo` endpoint, with translations.

Claims may be entered as simple strings or pipe separated strings representing the internal claim name, locale, and localized description.

For example: `name|en|Your full name..`

Locale strings are in the format: `language + "_" + country + "_" + variant`, for example `en`, `en_GB`, or `en_US_WIN`. If the locale and pipe is omitted, the description is displayed to all users that have undefined locales.

If the description is also omitted, nothing is displayed on the consent page for the claim. For example specifying `family_name|` would allow the claim `family_name` to be used by the client, but would not display it to the user on the consent page when requested.

**amster** attribute: `supportedClaims`

## OpenID Connect JWT Token Lifetime (seconds)

The amount of time the JWT will be valid for, in seconds.

Default value: `3600`

**amster** attribute: `jwtTokenLifetime`

## Advanced OpenID Connect

The following settings appear on the **Advanced OpenID Connect** tab:

### Remote JSON Web Key URL

The Remote URL where the providers JSON Web Key can be retrieved.

If this setting is not configured, then AM provides a local URL to access the public key of the private key used to sign ID tokens.

**amster** attribute: `jkwsURI`

### Idtokeninfo Endpoint Requires Client Authentication

When enabled, the `/oauth2/idtokeninfo` endpoint requires client authentication if the signing algorithm is set to `HS256`, `HS384`, or `HS512`.

Default value: `true`

**amster** attribute: `idTokenInfoClientAuthenticationEnabled`

### Enable "claims\_parameter\_supported"

If enabled, clients will be able to request individual claims using the `claims` request parameter, as per section 5.5 of the OpenID Connect specification.

Default value: `false`

**amster** attribute: `claimsParameterSupported`

### OpenID Connect acr\_values to Auth Chain Mapping

Maps OpenID Connect ACR values to authentication chains. For more details, see the `acr_values` parameter in the OpenID Connect authentication request specification.

**amster** attribute: `loaMapping`

### Default ACR values

Default requested Authentication Context Class Reference values.

List of strings that specifies the default acr values that the OP is being requested to use for processing requests from this Client, with the values appearing in order of preference. The Authentication Context Class satisfied by the authentication performed is returned as the

acr Claim Value in the issued ID Token. The acr Claim is requested as a Voluntary Claim by this parameter. The `acr_values_supported` discovery element contains a list of the acr values supported by this server. Values specified in the `acr_values` request parameter or an individual acr Claim request override these default values.

**amster** attribute: `defaultACR`

### OpenID Connect `id_token amr` Values to Auth Module Mappings

Specify `amr` values to be returned in the OpenID Connect `id_token`. Once authentication has completed, the authentication modules that were used from the authentication service will be mapped to the `amr` values. If you do not require `amr` values, or are not providing OpenID Connect tokens, leave this field blank.

**amster** attribute: `amrMappings`

### Always Return Claims in ID Tokens

If enabled, include scope-derived claims in the `id_token`, even if an access token is also returned that could provide access to get the claims from the `userinfo` endpoint.

If not enabled, if an access token is requested the client must use it to access the `userinfo` endpoint for scope-derived claims, as they will not be included in the ID token.

Default value: `false`

**amster** attribute: `alwaysAddClaimsToToken`

### Store Ops Tokens

Whether AM will store the `ops` tokens corresponding to OpenID Connect sessions in the CTS store. Note that session management related endpoints will not work when this setting is disabled.

Default value: `true`

**amster** attribute: `storeOpsTokens`

### Request Parameter Signing Algorithms Supported

Algorithms supported to verify signature of Request parameterAM supports signing algorithms listed in JSON Web Algorithms (JWA): "alg" (Algorithm) Header Parameter Values for JWS:

- `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.

Default value:

```
PS384
ES384
RS384
HS256
HS512
ES256
RS256
HS384
ES512
PS256
PS512
RS512
```

**amster** attribute: `supportedRequestParameterSigningAlgorithms`

## Request Parameter Encryption Algorithms Supported

Encryption algorithms supported to decrypt Request parameter.

AM supports the following ID token encryption algorithms:

- **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.
- **A128KW** - AES Key Wrapping with 128-bit key derived from the client secret.
- **RSA1\_5** - RSA with PKCS#1 v1.5 padding.
- **A256KW** - AES Key Wrapping with 256-bit key derived from the client secret.
- **dir** - Direct encryption with AES using the hashed client secret.
- **A192KW** - AES Key Wrapping with 192-bit key derived from the client secret.

Default value:

```
RSA-OAEP
RSA-OAEP-256
A128KW
A256KW
RSA1_5
dir
A192KW
```

**amster** attribute: `supportedRequestParameterEncryptionAlgorithms`

## Request Parameter Encryption Methods Supported

Encryption methods supported to decrypt Request parameter.

AM supports the following Request parameter encryption algorithms:

- [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:

```
A256GCM
A192GCM
A128GCM
A128CBC-HS256
A192CBC-HS384
A256CBC-HS512
```

**amster** attribute: [supportedRequestParameterEncryptionEnc](#)

## Supported Token Endpoint JWS Signing Algorithms.

Supported JWS Signing Algorithms for 'private\_key\_jwt' JWT based authentication method.

Default value:

```
PS384
ES384
RS384
HS256
HS512
ES256
RS256
HS384
ES512
PS256
PS512
RS512
```

**amster** attribute: [supportedTokenEndpointAuthenticationSigningAlgorithms](#)

## Authorized OIDC SSO Clients

Clients authorized to use OpenID Connect ID tokens as SSO Tokens.

Allows clients to act with the full authority of the user. Grant this permission only to trusted clients.

**amster** attribute: [authorisedOpenIdConnectSSOClients](#)

## UserInfo Signing Algorithms Supported

Algorithms supported to verify signature of the UserInfo endpoint. AM supports signing algorithms listed in JSON Web Algorithms (JWA): "alg" (Algorithm) Header Parameter Values for JWS:

- **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.

Default value:

```
ES384
HS256
HS512
ES256
RS256
HS384
ES512
```

**amster** attribute: `supportedUserInfoSigningAlgorithms`

## UserInfo Encryption Algorithms Supported

Encryption algorithms supported by the UserInfo endpoint.

AM supports the following UserInfo endpoint encryption algorithms:

- **RSA-0AEP** - RSA with Optimal Asymmetric Encryption Padding (OAEP) with SHA-1 and MGF-1.
- **RSA-0AEP-256** - RSA with OAEP with SHA-256 and MGF-1.
- **A128KW** - AES Key Wrapping with 128-bit key derived from the client secret.
- **RSA1\_5** - RSA with PKCS#1 v1.5 padding.
- **A256KW** - AES Key Wrapping with 256-bit key derived from the client secret.
- **dir** - Direct encryption with AES using the hashed client secret.
- **A192KW** - AES Key Wrapping with 192-bit key derived from the client secret.

Default value:

```
RSA-OAEP
RSA-OAEP-256
A128KW
A256KW
RSA1_5
dir
A192KW
```

**amster** attribute: `supportedUserInfoEncryptionAlgorithms`

## UserInfo Encryption Methods Supported

Encryption methods supported by the UserInfo endpoint.

AM supports the following UserInfo endpoint 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:

```
A256GCM
A192GCM
A128GCM
A128CBC-HS256
A192CBC-HS384
A256CBC-HS512
```

**amster** attribute: `supportedUserInfoEncryptionEnc`

## Use Force Authentication for `prompt=login`

This setting is applied only when you've implemented modules or chains, and you've specified the `prompt=login` parameter. The default value is `false`.

When set to `false`, AM forces the end user to authenticate even if they already have a valid session. After re-authentication, AM creates a new session.

When set to `true`, AM forces the end user to authenticate even if they already have a valid session. But, after re-authentication, AM returns the same session ID. Setting this to `false`, to create new a session, is recommended to increase the level of security.

## Device Flow

The following settings appear on the **Device Flow** tab:



## Verification URL

The URL that the user will be instructed to visit to complete their OAuth 2.0 login and consent when using the device code flow.

**amster** attribute: `verificationUrl`

## Device Completion URL

The URL that the user will be sent to on completion of their OAuth 2.0 login and consent when using the device code flow.

**amster** attribute: `completionUrl`

## Device Code Lifetime (seconds)

The lifetime of the device code, in seconds.

Default value: `300`

**amster** attribute: `deviceCodeLifetime`

## Device Polling Interval

The polling frequency for devices waiting for tokens when using the device code flow.

Default value: `5`

**amster** attribute: `devicePollInterval`

## Consent

The following settings appear on the **Consent** tab:

### Saved Consent Attribute Name

Name of a multi-valued attribute on resource owner profiles where AM can save authorization consent decisions.

When the resource owner chooses to save the decision to authorize access for a client application, then AM updates the resource owner's profile to avoid having to prompt the resource owner to grant authorization when the client issues subsequent authorization requests.

**amster** attribute: `savedConsentAttribute`

### Allow Clients to Skip Consent

If enabled, clients may be configured so that the resource owner will not be asked for consent during authorization flows.

Default value: `false`

**amster** attribute: `clientsCanSkipConsent`

### Enable Remote Consent

Default value: `false`

**amster** attribute: `enableRemoteConsent`

### Remote Consent Service ID

The ID of an existing remote consent service agent.

The possible values for this property are:

- `[Empty]`

**amster** attribute: `remoteConsentServiceId`

### Remote Consent Service Request Signing Algorithms Supported

Algorithms supported to sign `consent_request` JWTs for Remote Consent Services.

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

- `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.

Default value:

```
PS384
ES384
RS384
HS256
HS512
ES256
RS256
HS384
ES512
PS256
```

```
PS512
RS512
```

**amster** attribute: `supportedRcsRequestSigningAlgorithms`

## Remote Consent Service Request Encryption Algorithms Supported

Encryption algorithms supported to encrypt Remote Consent Service requests.

AM supports the following encryption algorithms:

- `RSA1_5` - RSA with PKCS#1 v1.5 padding.
- `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.
- `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.
- `dir` - Direct encryption with AES using the hashed client secret.

Default value:

```
RSA-OAEP
RSA-OAEP-256
A128KW
RSA1_5
A256KW
dir
A192KW
```

**amster** attribute: `supportedRcsRequestEncryptionAlgorithms`

## Remote Consent Service Request Encryption Methods Supported

Encryption methods supported to encrypt Remote Consent Service requests.

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:

```
A256GCM
```

```
A192GCM
A128GCM
A128CBC-HS256
A192CBC-HS384
A256CBC-HS512
```

**amster** attribute: `supportedRcsRequestEncryptionMethods`

## Remote Consent Service Response Signing Algorithms Supported

Algorithms supported to verify signed consent\_response JWT from Remote Consent Services.

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

- `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.

Default value:

```
PS384
ES384
RS384
HS256
HS512
ES256
RS256
HS384
ES512
PS256
PS512
RS512
```

**amster** attribute: `supportedRcsResponseSigningAlgorithms`

## Remote Consent Service Response Encryption Algorithms Supported

Encryption algorithms supported to decrypt Remote Consent Service responses.

AM supports the following encryption algorithms:

- `RSA1_5` - RSA with PKCS#1 v1.5 padding.

- **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.
- **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.
- **dir** - Direct encryption with AES using the hashed client secret.

Default value:

```
RSA-OAEP
RSA-OAEP-256
A128KW
A256KW
RSA1_5
dir
A192KW
```

**amster** attribute: `supportedRcsResponseEncryptionAlgorithms`

## Remote Consent Service Response Encryption Methods Supported

Encryption methods supported to decrypt Remote Consent Service responses.

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:

```
A256GCM
A192GCM
A128GCM
A128CBC-HS256
A192CBC-HS384
A256CBC-HS512
```

**amster** attribute: `supportedRcsResponseEncryptionMethods`

## CIBA

The following settings appear on the **CIBA** tab:

### Back Channel Authentication ID Lifetime (seconds)

The time back channel authentication request id is valid for, in seconds.

Default value: 600

**amster** attribute: `cibaAuthReqIdLifetime`

### Polling Wait Interval (seconds)

The minimum amount of time in seconds that the Client should wait between polling requests to the token endpoint

Default value: 2

**amster** attribute: `cibaMinimumPollingInterval`

### Signing Algorithms Supported

Algorithms supported to sign the CIBA request parameter.

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.
- `PS256` - RSASSA-PSS using SHA-256.

Default value:

```
ES256  
PS256
```

**amster** attribute: `supportedCibaSigningAlgorithms`

## Remote Consent Service

**amster** service name: `RemoteConsentService`

### Realm Defaults

The following settings appear on the **Realm Defaults** tab:

#### Client Name

The name used to identify this OAuth 2.0 remote consent service when referenced in other services.

**amster** attribute: `clientId`

### Signing Key Alias

The alias of the key in the default keystore to use for signing.

**amster** attribute: `signingKeyAlias`

### Encryption Key Alias

The alias of the key in the default keystore to use for encryption.

**amster** attribute: `encryptionKeyAlias`

### Authorization Server `jwt_uri`

The `jwt_uri` for retrieving the authorization server signing and encryption keys.

**amster** attribute: `jwtUriAS`

### JWK Store Cache Timeout (in minutes)

The cache timeout for the JWK store of the authorization server, in minutes.

Default value: `60`

**amster** attribute: `jwtStoreCacheTimeout`

### JWK Store Cache Miss Cache Time (in minutes)

The length of time a cache miss is cached, in minutes.

Default value: `1`

**amster** attribute: `jwtStoreCacheMissCacheTime`

### Consent Response Time Limit (in minutes)

The time limit set on the consent response JWT before it expires, in minutes.

Default value: `2`

**amster** attribute: `consentResponseTimeLimit`

## OAuth 2.0 and OpenID Connect 1.0 Client Settings

To register an OAuth 2.0 client with AM as the OAuth 2.0 authorization server, or register an OpenID Connect 1.0 client through the AM console, then create an OAuth 2.0 client profile. After creating the

client profile, you can further configure the properties in the AM console by navigating to Realms > *Realm Name* > Applications > OAuth 2.0 > *Client Name*.

## Core

The following properties appear on the Core tab:

### 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.

### 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. The redirection URI must NOT contain a fragment (`#`).

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. 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 a special scope, `am-introspect-all-tokens`. As administrator, add this scope to the OAuth 2.0 client profile to allow the client to introspect access tokens issued to other clients in the same realm. This scope cannot be added during dynamic client registration.

## Default Scope(s)

Specify scopes in `scope` or `scope|locale|localized description` format. These scopes are set automatically when tokens are issued.

The `openid` scope is required. 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.

## 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

The following properties appear on the Advanced tab:

### 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.

### 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](#)

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".

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](#).

By default, the following response types are available:

- [code](#). Specifies that the client application requests an authorization code grant.
- [token](#). Specifies that the client application requests an implicit grant type and requests a token from the API.
- [id\\_token](#). Specifies that the client application requests an ID token.
- [code token](#). Specifies that the client application requests an access token, access token type, and an authorization code.
- [token id\\_token](#). Specifies that the client application requests an access token, access token type, and an ID token.
- [code id\\_token](#). Specifies that the client application requests an authorization code and an ID token.
- [code token id\\_token](#). Specifies that the client application requests an authorization code, access token, access token type, and an ID token.

## 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 "*Authenticating OAuth 2.0 Clients*", 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

The following properties appear on the OpenID Connect tab:

### 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 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)

Specify 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`

## Signing and Encryption

### Note

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.

The following properties appear on the Signing and Encryption tab:

## Json Web Key URI

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

## 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 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`

## 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` authentication method 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-----  
MIIDETCCAfmGAWIBA...  
-----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 *Setup and Maintenance Guide*.

To export the certificate from the new key pair in PEM format, run a command similar to the following:

```
$ keytool \  
-list \  
-alias myAlias \  
-rfc \  
-storetype JCEKS \  
-keystore myKeystore.jceks \  
-keypass myKeypass \  
-storepass myStorepass  
  
Alias name: myAlias  
Creation date: Oct 27, 2014  
Entry type: PrivateKeyEntry  
Certificate chain length: 1  
Certificate[1]:  
-----BEGIN CERTIFICATE-----  
MIIDETCCAfmGAWIBA...  
-----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=my0auth2Client`.

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 labelled `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".

## 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".

## 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".

Default: [A128CBC-HS256](#)

## UMA

The following properties appear on the UMA tab:

## 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.

## OAuth 2.0 Remote Consent Agent Settings

To register an OAuth 2.0 remote consent service with AM as the OAuth 2.0 authorization server, create a Remote Consent Agent profile. After creating the profile, you can further configure the properties in the AM console by navigating to Realms > *Realm Name* > Applications > Remote Consent > *Agent Name*.

The following properties appear on the agent profile page:

### Note

The properties value examples below are applicable to the example remote consent service provided with AM. Alter the values as required by your remote consent service.

### 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 default secret ID mappings used to sign remote consent requests:

Secret ID	Default Alias	Algorithms <sup>a</sup>
am.applications.agents.remote.consent.request.signing.es256	ES256	ES256
am.applications.agents.remote.consent.request.signing.es384	ES384	ES384
am.applications.agents.remote.consent.request.signing.es512	ES512	ES512
am.applications.agents.remote.consent.request.signing.rsa	RSAtsigningkey	RS256 RS384 RS512 PS256 PS384 PS512

<sup>a</sup> 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 *Setup and Maintenance 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-0AEP` - RSA with Optimal Asymmetric Encryption Padding (OAEP) with SHA-1 and MGF-1.
- `RSA-0AEP-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-0AEP-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 default secret ID mapping used to decrypt remote consent responses:

Secret ID	Default Alias	Algorithms <sup>a</sup>
am.services.oauth2.remote.consent.response.decrypt	demo	RSA-OAEP-256

<sup>a</sup> 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 *Setup and Maintenance Guide*.

Default value: `RSA-0AEP-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 `JWKS` is selected, you must 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 AM example remote consent service provides an `/oauth2/consent/jwk_uri` path to provide the 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-0AEP-256",
      "n": "AL4kjjz74rDo3VQ3Wx...nhch4qJRGt2QnCF7M0",
      "e": "AQAB"
    },
    {
      "kty": "RSA",
      "kid": "wUy3ifIIaL...eM1rP1QM=",
      "use": "sig",
      "alg": "RS256",
      "n": "ANdIhk0ZeSHagT9Ze...ci0ACVuGUoNTzztLCUK",
      "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

# Appendix A. About the REST API

This appendix shows how to use the RESTful interfaces for direct integration between web client applications and ForgeRock Access Management.

## Introducing REST

Representational State Transfer (REST) is an architectural style that sets certain constraints for designing and building large-scale distributed hypermedia systems.

As an architectural style, REST has very broad applications. The designs of both HTTP 1.1 and URIs follow RESTful principles. The World Wide Web is no doubt the largest and best known REST application. Many other web services also follow the REST architectural style. Examples include OAuth 2.0, OpenID Connect 1.0, and User-Managed Access (UMA).

The ForgeRock Common REST (CREST) API applies RESTful principles to define common verbs for HTTP-based APIs that access web resources and collections of web resources.

Interface Stability: Evolving

Most native AM REST APIs use the CREST verbs. (In contrast, OAuth 2.0, OpenID Connect 1.0 and UMA APIs follow their respective standards.)

## About ForgeRock Common REST

ForgeRock® Common REST is a common REST API framework. It works across the ForgeRock platform to provide common ways to access web resources and collections of resources. Adapt the examples in this section to your resources and deployment.

### Note

This section describes the full Common REST framework. Some platform component products do not implement all Common REST behaviors exactly as described in this section. For details, refer to the product-specific examples and reference information in other sections of this documentation set.

## Common REST Resources

Servers generally return JSON-format resources, though resource formats can depend on the implementation.

Resources in collections can be found by their unique identifiers (IDs). IDs are exposed in the resource URIs. For example, if a server has a user collection under `/users`, then you can access a user at `/users/user-id`. The ID is also the value of the `_id` field of the resource.

Resources are versioned using revision numbers. A revision is specified in the resource's `_rev` field. Revisions make it possible to figure out whether to apply changes without resource locking and without distributed transactions.

## Common REST Verbs

The Common REST APIs use the following verbs, sometimes referred to collectively as CRUDPAQ. For details and HTTP-based examples of each, follow the links to the sections for each verb.

### Create

Add a new resource.

This verb maps to HTTP PUT or HTTP POST.

For details, see "Create".

### Read

Retrieve a single resource.

This verb maps to HTTP GET.

For details, see "Read".

### Update

Replace an existing resource.

This verb maps to HTTP PUT.

For details, see "Update".

## Delete

Remove an existing resource.

This verb maps to HTTP DELETE.

For details, see "Delete".

## Patch

Modify part of an existing resource.

This verb maps to HTTP PATCH.

For details, see "Patch".

## Action

Perform a predefined action.

This verb maps to HTTP POST.

For details, see "Action".

## Query

Search a collection of resources.

This verb maps to HTTP GET.

For details, see "Query".

## Common REST Parameters

Common REST reserved query string parameter names start with an underscore, `_`.

Reserved query string parameters include, but are not limited to, the following names:

```
_action  
_api  
_crestapi  
_fields  
_mimeType  
_pageSize  
_pagedResultsCookie  
_pagedResultsOffset  
_prettyPrint  
_queryExpression
```

`_queryFilter`  
`_queryId`  
`_sortKeys`  
`_totalPagedResultsPolicy`

#### Note

Some parameter values are not safe for URLs, so URL-encode parameter values as necessary.

Continue reading for details about how to use each parameter.

## Common REST Extension Points

The *action* verb is the main vehicle for extensions. For example, to create a new user with HTTP POST rather than HTTP PUT, you might use `/users?_action=create`. A server can define additional actions. For example, `/tasks/1?_action=cancel`.

A server can define *stored queries* to call by ID. For example, `/groups?_queryId=hasDeletedMembers`. Stored queries can call for additional parameters. The parameters are also passed in the query string. Which parameters are valid depends on the stored query.

## Common REST API Documentation

Common REST APIs often depend at least in part on runtime configuration. Many Common REST endpoints therefore serve *API descriptors* at runtime. An API descriptor documents the actual API as it is configured.

Use the following query string parameters to retrieve API descriptors:

### `_api`

Serves an API descriptor that complies with the OpenAPI specification.

This API descriptor represents the API accessible over HTTP. It is suitable for use with popular tools such as Swagger UI.

### `_crestapi`

Serves a native Common REST API descriptor.

This API descriptor provides a compact representation that is not dependent on the transport protocol. It requires a client that understands Common REST, as it omits many Common REST defaults.

#### Note

Consider limiting access to API descriptors in production environments in order to avoid unnecessary traffic.

To provide documentation in production environments, see "To Publish OpenAPI Documentation" instead.

## To Publish OpenAPI Documentation

In production systems, developers expect stable, well-documented APIs. Rather than retrieving API descriptors at runtime through Common REST, prepare final versions, and publish them alongside the software in production.

Use the OpenAPI-compliant descriptors to provide API reference documentation for your developers as described in the following steps:

1. Configure the software to produce production-ready APIs.

In other words, the software should be configured as in production so that the APIs are identical to what developers see in production.

2. Retrieve the OpenAPI-compliant descriptor.

The following command saves the descriptor to a file, `myapi.json`:

```
$ curl -o myapi.json endpoint?_api
```

3. (Optional) If necessary, edit the descriptor.

For example, you might want to add security definitions to describe how the API is protected.

If you make any changes, then also consider using a source control system to manage your versions of the API descriptor.

4. Publish the descriptor using a tool such as Swagger UI.

You can customize Swagger UI for your organization as described in the documentation for the tool.

## Create

There are two ways to create a resource, either with an HTTP POST or with an HTTP PUT.

To create a resource using POST, perform an HTTP POST with the query string parameter `action=create` and the JSON resource as a payload. Accept a JSON response. The server creates the identifier if not specified:

```
POST /users?_action=create HTTP/1.1
Host: example.com
Accept: application/json
Content-Length: ...
Content-Type: application/json
{ JSON resource }
```

To create a resource using PUT, perform an HTTP PUT including the case-sensitive identifier for the resource in the URL path, and the JSON resource as a payload. Use the `If-None-Match: *` header. Accept a JSON response:

```
PUT /users/some-id HTTP/1.1
Host: example.com
Accept: application/json
Content-Length: ...
Content-Type: application/json
If-None-Match: *
{ JSON resource }
```

The `_id` and content of the resource depend on the server implementation. The server is not required to use the `_id` that the client provides. The server response to the create request indicates the resource location as the value of the `Location` header.

If you include the `If-None-Match` header, its value must be `*`. In this case, the request creates the object if it does not exist, and fails if the object does exist. If you include the `If-None-Match` header with any value other than `*`, the server returns an HTTP 400 Bad Request error. For example, creating an object with `If-None-Match: revision` returns a bad request error. If you do not include `If-None-Match: *`, the request creates the object if it does not exist, and *updates* the object if it does exist.

## Parameters

You can use the following parameters:

`_prettyPrint=true`

Format the body of the response.

`_fields=field[,field...]`

Return only the specified fields in the body of the response.

The `field` values are JSON pointers. For example if the resource is `{"parent":{"child":"value"}}`, `parent/child` refers to the `"child":"value"`.

If the `field` is left blank, the server returns all default values.



## Read

To retrieve a single resource, perform an HTTP GET on the resource by its case-sensitive identifier (`_id`) and accept a JSON response:

```
GET /users/some-id HTTP/1.1
Host: example.com
Accept: application/json
```

### Parameters

You can use the following parameters:

`_prettyPrint=true`

Format the body of the response.

`_fields=field[,field...]`

Return only the specified fields in the body of the response.

The `field` values are JSON pointers. For example if the resource is `{"parent":{"child":"value"}}`, `parent/child` refers to the `"child":"value"`.

If the `field` is left blank, the server returns all default values.

`_mimeType=mime-type`

Some resources have fields whose values are multi-media resources such as a profile photo for example.

By specifying both a single `field` and also the `mime-type` for the response content, you can read a single field value that is a multi-media resource.

In this case, the content type of the field value returned matches the `mime-type` that you specify, and the body of the response is the multi-media resource.

The `Accept` header is not used in this case. For example, `Accept: image/png` does not work. Use the `_mimeType` query string parameter instead.

## Update

To update a resource, perform an HTTP PUT including the case-sensitive identifier (`_id`) as the final element of the path to the resource, and the JSON resource as the payload. Use the `If-Match: _rev` header to check that you are actually updating the version you modified. Use `If-Match: *` if the version does not matter. Accept a JSON response:

```
PUT /users/some-id HTTP/1.1
Host: example.com
Accept: application/json
Content-Length: ...
Content-Type: application/json
If-Match: _rev
{ JSON resource }
```

When updating a resource, include all the attributes to be retained. Omitting an attribute in the resource amounts to deleting the attribute unless it is not under the control of your application. Attributes not under the control of your application include private and read-only attributes. In addition, virtual attributes and relationship references might not be under the control of your application.

## Parameters

You can use the following parameters:

`_prettyPrint=true`

Format the body of the response.

`_fields=field[,field...]`

Return only the specified fields in the body of the response.

The `field` values are JSON pointers. For example if the resource is `{"parent":{"child":"value"}}`, `parent/child` refers to the `"child":"value"`.

If the `field` is left blank, the server returns all default values.

## Delete

To delete a single resource, perform an HTTP DELETE by its case-sensitive identifier (`_id`) and accept a JSON response:

```
DELETE /users/some-id HTTP/1.1
Host: example.com
Accept: application/json
```

## Parameters

You can use the following parameters:

`_prettyPrint=true`

Format the body of the response.

### `_fields=field[,field...]`

Return only the specified fields in the body of the response.

The `field` values are JSON pointers. For example if the resource is `{"parent":{"child":"value"}}`, `parent/child` refers to the `"child":"value"`.

If the `field` is left blank, the server returns all default values.

## Patch

To patch a resource, send an HTTP PATCH request with the following parameters:

- `operation`
- `field`
- `value`
- `from` (optional with copy and move operations)

You can include these parameters in the payload for a PATCH request, or in a JSON PATCH file. If successful, you'll see a JSON response similar to:

```
PATCH /users/some-id HTTP/1.1
Host: example.com
Accept: application/json
Content-Length: ...
Content-Type: application/json
If-Match: _rev
{ JSON array of patch operations }
```

PATCH operations apply to three types of targets:

- **single-valued**, such as an object, string, boolean, or number.
- **list semantics array**, where the elements are ordered, and duplicates are allowed.
- **set semantics array**, where the elements are not ordered, and duplicates are not allowed.

ForgeRock PATCH supports several different `operations`. The following sections show each of these operations, along with options for the `field` and `value`:

### Patch Operation: Add

The `add` operation ensures that the target field contains the value provided, creating parent fields as necessary.

If the target field is single-valued, then the value you include in the PATCH replaces the value of the target. Examples of a single-valued field include: object, string, boolean, or number.

An **add** operation has different results on two standard types of arrays:

- **List semantic arrays:** you can run any of these **add** operations on that type of array:
  - If you **add** an array of values, the PATCH operation appends it to the existing list of values.
  - If you **add** a single value, specify an ordinal element in the target array, or use the **{-}** special index to add that value to the end of the list.
- **Set semantic arrays:** The value included in the patch is merged with the existing set of values. Any duplicates within the array are removed.

As an example, start with the following list semantic array resource:

```
{
  "fruits" : [ "orange", "apple" ]
}
```

The following add operation includes the pineapple to the end of the list of fruits, as indicated by the **-** at the end of the **fruits** array.

```
{
  "operation" : "add",
  "field" : "/fruits/-",
  "value" : "pineapple"
}
```

The following is the resulting resource:

```
{
  "fruits" : [ "orange", "apple", "pineapple" ]
}
```

Note that you can add only one array element one at a time, as per the corresponding JSON Patch specification. If you add an array of elements, for example:

```
{
  "operation" : "add",
  "field" : "/fruits/-",
  "value" : ["pineapple", "mango"]
}
```

The resulting resource would have the following invalid JSON structure:

```
{
  "fruits" : [ "orange", "apple", ["pineapple", "mango"] ]
}
```

## Patch Operation: Copy

The copy operation takes one or more existing values from the source field. It then adds those same values on the target field. Once the values are known, it is equivalent to performing an **add** operation on the target field.

The following `copy` operation takes the value from a field named `mail`, and then runs a `replace` operation on the target field, `another_mail`.

```
[
  {
    "operation": "copy",
    "from": "mail",
    "field": "another_mail"
  }
]
```

If the source field value and the target field value are configured as arrays, the result depends on whether the array has list semantics or set semantics, as described in "Patch Operation: Add".

## Patch Operation: Increment

The `increment` operation changes the value or values of the target field by the amount you specify. The value that you include must be one number, and may be positive or negative. The value of the target field must accept numbers. The following `increment` operation adds `1000` to the target value of `/user/payment`.

```
[
  {
    "operation": "increment",
    "field": "/user/payment",
    "value": "1000"
  }
]
```

Since the `value` of the `increment` is a single number, arrays do not apply.

## Patch Operation: Move

The move operation removes existing values on the source field. It then adds those same values on the target field. It is equivalent to performing a `remove` operation on the source, followed by an `add` operation with the same values, on the target.

The following `move` operation is equivalent to a `remove` operation on the source field, `surname`, followed by a `replace` operation on the target field value, `lastName`. If the target field does not exist, it is created.

```
[
  {
    "operation": "move",
    "from": "surname",
    "field": "lastName"
  }
]
```

To apply a `move` operation on an array, you need a compatible single-value, list semantic array, or set semantic array on both the source and the target. For details, see the criteria described in "Patch Operation: Add".

## Patch Operation: Remove

The **remove** operation ensures that the target field no longer contains the value provided. If the remove operation does not include a value, the operation removes the field. The following **remove** deletes the value of the **phoneNumber**, along with the field.

```
[
  {
    "operation" : "remove",
    "field" : "phoneNumber"
  }
]
```

If the object has more than one **phoneNumber**, those values are stored as an array.

A **remove** operation has different results on two standard types of arrays:

- **List semantic arrays:** A **remove** operation deletes the specified element in the array. For example, the following operation removes the first phone number, based on its array index (zero-based):

```
[
  {
    "operation" : "remove",
    "field" : "/phoneNumber/0"
  }
]
```

- **Set semantic arrays:** The list of values included in a patch are removed from the existing array.

## Patch Operation: Replace

The **replace** operation removes any existing value(s) of the targeted field, and replaces them with the provided value(s). It is essentially equivalent to a **remove** followed by a **add** operation. If the arrays are used, the criteria is based on "Patch Operation: Add". However, indexed updates are not allowed, even when the target is an array.

The following **replace** operation removes the existing **telephoneNumber** value for the user, and then adds the new value of **+1 408 555 9999**.

```
[
  {
    "operation" : "replace",
    "field" : "/telephoneNumber",
    "value" : "+1 408 555 9999"
  }
]
```

A PATCH replace operation on a list semantic array works in the same fashion as a PATCH remove operation. The following example demonstrates how the effect of both operations. Start with the following resource:

```
{
  "fruits" : [ "apple", "orange", "kiwi", "lime" ],
}
```

Apply the following operations on that resource:

```
[
  {
    "operation" : "remove",
    "field" : "/fruits/0",
    "value" : ""
  },
  {
    "operation" : "replace",
    "field" : "/fruits/1",
    "value" : "pineapple"
  }
]
```

The PATCH operations are applied sequentially. The `remove` operation removes the first member of that resource, based on its array index, (`fruits/0`), with the following result:

```
[
  {
    "fruits" : [ "orange", "kiwi", "lime" ],
  }
]
```

The second PATCH operation, a `replace`, is applied on the second member (`fruits/1`) of the intermediate resource, with the following result:

```
[
  {
    "fruits" : [ "orange", "pineapple", "lime" ],
  }
]
```

## Patch Operation: Transform

The `transform` operation changes the value of a field based on a script or some other data transformation command. The following `transform` operation takes the value from the field named `/objects`, and applies the `something.js` script as shown:

```
[
  {
    "operation" : "transform",
    "field" : "/objects",
    "value" : {
      "script" : {
        "type" : "text/javascript",
        "file" : "something.js"
      }
    }
  }
]
```

## Patch Operation Limitations

Some HTTP client libraries do not support the HTTP PATCH operation. Make sure that the library you use supports HTTP PATCH before using this REST operation.

For example, the Java Development Kit HTTP client does not support PATCH as a valid HTTP method. Instead, the method `HttpURLConnection.setRequestMethod("PATCH")` throws `ProtocolException`.

### Parameters

You can use the following parameters. Other parameters might depend on the specific action implementation:

`_prettyPrint=true`

Format the body of the response.

`_fields=field[,field...]`

Return only the specified fields in the body of the response.

The `field` values are JSON pointers. For example if the resource is `{"parent":{"child":"value"}}`, `parent/child` refers to the `"child":"value"`.

If the `field` is left blank, the server returns all default values.

### Action

Actions are a means of extending Common REST APIs and are defined by the resource provider, so the actions you can use depend on the implementation.

The standard action indicated by `_action=create` is described in "Create".

### Parameters

You can use the following parameters. Other parameters might depend on the specific action implementation:

`_prettyPrint=true`

Format the body of the response.

`_fields=field[,field...]`

Return only the specified fields in the body of the response.

The `field` values are JSON pointers. For example if the resource is `{"parent":{"child":"value"}}`, `parent/child` refers to the `"child":"value"`.

If the `field` is left blank, the server returns all default values.



## Query

To query a resource collection (or resource container if you prefer to think of it that way), perform an HTTP GET and accept a JSON response, including at least a `_queryExpression`, `_queryFilter`, or `_queryId` parameter. These parameters cannot be used together:

```
GET /users?_queryFilter=true HTTP/1.1
Host: example.com
Accept: application/json
```

The server returns the result as a JSON object including a "results" array and other fields related to the query string parameters that you specify.

## Parameters

You can use the following parameters:

`\_queryFilter=filter-expression`

Query filters request that the server return entries that match the filter expression. You must URL-escape the filter expression.

The string representation is summarized as follows. Continue reading for additional explanation:

```
Expr           = OrExpr
OrExpr         = AndExpr ( 'or' AndExpr ) *
AndExpr        = NotExpr ( 'and' NotExpr ) *
NotExpr        = '!' PrimaryExpr | PrimaryExpr
PrimaryExpr    = '(' Expr ')' | ComparisonExpr | PresenceExpr | LiteralExpr
ComparisonExpr = Pointer OpName JsonValue
PresenceExpr   = Pointer 'pr'
LiteralExpr    = 'true' | 'false'
Pointer        = JSON pointer
OpName         = 'eq' | # equal to
                'co' | # contains
                'sw' | # starts with
                'lt' | # less than
                'le' | # less than or equal to
                'gt' | # greater than
                'ge' | # greater than or equal to
                STRING # extended operator
JsonValue      = NUMBER | BOOLEAN | ''' UTF8STRING '''
STRING         = ASCII string not containing white-space
UTF8STRING     = UTF-8 string possibly containing white-space
```

*JsonValue* components of filter expressions follow RFC 7159: *The JavaScript Object Notation (JSON) Data Interchange Format*. In particular, as described in section 7 of the RFC, the escape character in strings is the backslash character. For example, to match the identifier `test\`, use `_id eq 'test\\'`. In the JSON resource, the `\` is escaped the same way: `"_id": "test\\"`.

When using a query filter in a URL, be aware that the filter expression is part of a query string parameter. A query string parameter must be URL encoded as described in RFC 3986: *Uniform Resource Identifier (URI): Generic Syntax* For example, white space, double quotes ("), parentheses, and exclamation characters need URL encoding in HTTP query strings. The following rules apply to URL query components:

```

query      = *( pchar / "/" / "?" )
pchar     = unreserved / pct-encoded / sub-delims / ":" / "@"
unreserved = ALPHA / DIGIT / "-" / "." / "_" / "~"
pct-encoded = "%" HEXDIG HEXDIG
sub-delims = "!" / "$" / "&" / "'" / "(" / ")"
           / "*" / "+" / "," / ";" / "="

```

**ALPHA**, **DIGIT**, and **HEXDIG** are core rules of RFC 5234: *Augmented BNF for Syntax Specifications*:

```

ALPHA     = %x41-5A / %x61-7A ; A-Z / a-z
DIGIT     = %x30-39          ; 0-9
HEXDIG    = DIGIT / "A" / "B" / "C" / "D" / "E" / "F"

```

As a result, a backslash escape character in a *JsonValue* component is percent-encoded in the URL query string parameter as %5C. To encode the query filter expression `_id eq 'test\\'`, use `_id +eq+'test%5C%5C'`, for example.

A simple filter expression can represent a comparison, presence, or a literal value.

For comparison expressions use *json-pointer comparator json-value*, where the *comparator* is one of the following:

```

eq (equals)
co (contains)
sw (starts with)
lt (less than)
le (less than or equal to)
gt (greater than)
ge (greater than or equal to)

```

For presence, use *json-pointer pr* to match resources where:

- The JSON pointer is present.
- The value it points to is not `null`.

Literal values include `true` (match anything) and `false` (match nothing).

Complex expressions employ `and`, `or`, and `!` (not), with parentheses, (*expression*), to group expressions.

### `_queryId=identifier`

Specify a query by its identifier.

Specific queries can take their own query string parameter arguments, which depend on the implementation.

#### `_pagedResultsCookie=string`

The string is an opaque cookie used by the server to keep track of the position in the search results. The server returns the cookie in the JSON response as the value of `pagedResultsCookie`.

In the request `_pageSize` must also be set and non-zero. You receive the cookie value from the provider on the first request, and then supply the cookie value in subsequent requests until the server returns a `null` cookie, meaning that the final page of results has been returned.

The `_pagedResultsCookie` parameter is supported when used with the `_queryFilter` parameter. The `pagedResultsCookie` parameter is not guaranteed to work when used with the `_queryExpression` and `_queryId` parameters.

The `_pagedResultsCookie` and `_pagedResultsOffset` parameters are mutually exclusive, and not to be used together.

#### `_pagedResultsOffset=integer`

When `_pageSize` is non-zero, use this as an index in the result set indicating the first page to return.

The `_pagedResultsCookie` and `_pagedResultsOffset` parameters are mutually exclusive, and not to be used together.

#### `_pageSize=integer`

Return query results in pages of this size. After the initial request, use `_pagedResultsCookie` or `_pageResultsOffset` to page through the results.

#### `_totalPagedResultsPolicy=string`

When a `_pageSize` is specified, and non-zero, the server calculates the "totalPagedResults", in accordance with the `totalPagedResultsPolicy`, and provides the value as part of the response. The "totalPagedResults" is either an estimate of the total number of paged results (`_totalPagedResultsPolicy=ESTIMATE`), or the exact total result count (`_totalPagedResultsPolicy=EXACT`). If no count policy is specified in the query, or if `_totalPagedResultsPolicy=NONE`, result counting is disabled, and the server returns value of -1 for "totalPagedResults".

#### `_sortKeys=[+/-]field[, [+/-]field...]`

Sort the resources returned based on the specified field(s), either in `+` (ascending, default) order, or in `-` (descending) order.

Because ascending order is the default, including the `+` character in the query is unnecessary. If you do include the `+`, it must be URL-encoded as `%2B`, for example:

```
http://localhost:8080/api/users?_prettyPrint=true&_queryFilter=true&_sortKeys=%2Bname/givenName
```

The `_sortKeys` parameter is not supported for predefined queries (`_queryId`).

#### `_prettyPrint=true`

Format the body of the response.

#### `_fields=field[,field...]`

Return only the specified fields in each element of the "results" array in the response.

The `field` values are JSON pointers. For example if the resource is `{"parent":{"child":"value"}}`, `parent/child` refers to the `"child":"value"`.

If the `field` is left blank, the server returns all default values.

## HTTP Status Codes

When working with a Common REST API over HTTP, client applications should expect at least the following HTTP status codes. Not all servers necessarily return all status codes identified here:

### **200 OK**

The request was successful and a resource returned, depending on the request.

### **201 Created**

The request succeeded and the resource was created.

### **204 No Content**

The action request succeeded, and there was no content to return.

### **304 Not Modified**

The read request included an `If-None-Match` header, and the value of the header matched the revision value of the resource.

### **400 Bad Request**

The request was malformed.

### **401 Unauthorized**

The request requires user authentication.

### **403 Forbidden**

Access was forbidden during an operation on a resource.

### **404 Not Found**

The specified resource could not be found, perhaps because it does not exist.

#### **405 Method Not Allowed**

The HTTP method is not allowed for the requested resource.

#### **406 Not Acceptable**

The request contains parameters that are not acceptable, such as a resource or protocol version that is not available.

#### **409 Conflict**

The request would have resulted in a conflict with the current state of the resource.

#### **410 Gone**

The requested resource is no longer available, and will not become available again. This can happen when resources expire for example.

#### **412 Precondition Failed**

The resource's current version does not match the version provided.

#### **415 Unsupported Media Type**

The request is in a format not supported by the requested resource for the requested method.

#### **428 Precondition Required**

The resource requires a version, but no version was supplied in the request.

#### **500 Internal Server Error**

The server encountered an unexpected condition that prevented it from fulfilling the request.

#### **501 Not Implemented**

The resource does not support the functionality required to fulfill the request.

#### **503 Service Unavailable**

The requested resource was temporarily unavailable. The service may have been disabled, for example.

## **Cross-Site Request Forgery (CSRF) Protection**

AM includes a global filter to harden AM's protection against CSRF attacks. The filter applies to all REST endpoints under `json/` and requires that all requests other than GET, HEAD, or OPTIONS include, at least, one of the following headers:

- `X-Requested-With`

This header is often sent by Javascript frameworks, and the XUI already sends it on all requests.

- **Accept-API-Version**

This header specifies which version of the REST API to use. Use this header in your requests to ensure future changes to the API do not affect your clients.

For more information about API versioning, see "REST API Versioning".

Failure to include at least one of the headers would cause the REST call to fail with a **403 Forbidden** error, even if the SSO token is valid.

To disable the filter, navigate to Configure > Global Services > REST APIs > and turn off Enable CSRF Protection.

The `json/` endpoint is not vulnerable to CSRF attacks when the filter is disabled, since it requires the "Content-Type: `application/json`" header, which currently triggers the same protection in browsers. This may change in the future, so it is recommended to enable the CSRF filter.

## REST API Versioning

In OpenAM 12.0.0 and later, REST API features are assigned version numbers.

Providing version numbers in the REST API helps ensure compatibility between releases. The version number of a feature increases when AM introduces a non-backwards-compatible change that affects clients making use of the feature.

AM provides versions for the following aspects of the REST API.

### **resource**

Any changes to the structure or syntax of a returned response will incur a *resource* version change. For example changing `errorMessage` to `message` in a JSON response.

### **protocol**

Any changes to the methods used to make REST API calls will incur a *protocol* version change. For example changing `_action` to `$action` in the required parameters of an API feature.

To ensure your clients are always compatible with a newer version of AM, you should always include resource versions in your REST calls.

## Supported REST API Versions

For information about the supported protocol and resource versions available in AM, see the API Explorer in the *Development Guide* available in the AM console.

The *AM Release Notes* section, "Changes and Deprecated Functionality" in the *Release Notes* describes the differences between API versions.

## Specifying an Explicit REST API Version

You can specify which version of the REST API to use by adding an `Accept-API-Version` header to the request. The following example requests *resource* version 2.0 and *protocol* version 1.0:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "X-OpenAM-Username: demo" \
--header "X-OpenAM-Password: changeit" \
--header "Accept-API-Version: resource=2.0, protocol=1.0" \
'https://openam.example.com:8443/openam/json/realms/root/authenticate'
```

You can configure the default behavior AM will take when a REST call does not specify explicit version information. For more information, see "Configuring the Default REST API Version for a Deployment".

## Configuring the Default REST API Version for a Deployment

You can configure the default behavior AM will take when a REST call does not specify explicit version information using either of the following procedures:

- "Configure Versioning Behavior by using the AM Console"
- "Configure Versioning Behavior by Using the ssoadm Command"

The available options for default behavior are as follows:

### **Latest**

The latest available supported version of the API is used.

This is the preset default for new installations of AM.

### **Oldest**

The oldest available supported version of the API is used.

This is the preset default for upgraded AM instances.

#### **Note**

The oldest supported version may not be the first that was released, as APIs versions become deprecated or unsupported. See "Deprecated Functionality" in the *Release Notes*.

### **None**

No version will be used. When a REST client application calls a REST API without specifying the version, AM returns an error and the request fails.

## Configure Versioning Behavior by using the AM Console

1. Log in as AM administrator, `amadmin`.
2. Click Configure > Global Services, and then click REST APIs.
3. In Default Version, select the required response to a REST API request that does not specify an explicit version: `Latest`, `Oldest`, or `None`.
4. (Optional) Optionally, enable `Warning Header` to include warning messages in the headers of responses to requests.
5. Save your work.

## Configure Versioning Behavior by Using the `ssoadm` Command

- Use the `ssoadm set-attr-defs` command with the `openam-rest-apis-default-version` attribute set to either `Latest`, `Oldest` or `None`, as in the following example:

```
$ ssh openam.example.com
$ cd /path/to/openam-tools/admin/openam/bin
$ ./ssoadm \
  set-attr-defs \
  --adminid amadmin \
  --password-file /tmp/pwd.txt \
  --servicename RestApisService \
  --schematype Global \
  --attributevalues openam-rest-apis-default-version=None
Schema attribute defaults were set.
```

## REST API Versioning Messages

AM provides REST API version messages in the JSON response to a REST API call. You can also configure AM to return version messages in the response headers.

Messages include:

- Details of the REST API versions used to service a REST API call.
- Warning messages if REST API version information is not specified or is incorrect in a REST API call.

The `resource` and `protocol` version used to service a REST API call are returned in the `Content-API-Version` header, as shown below:



```
$ curl \
-i \
--request POST \
--header "Content-Type: application/json" \
--header "X-OpenAM-Username: demo" \
--header "X-OpenAM-Password: changeit" \
--header "Accept-API-Version: resource=2.0, protocol=1.0" \
'https://openam.example.com:8443/openam/json/realms/root/authenticate'
HTTP/1.1 200 OK
Content-API-Version: protocol=1.0,resource=2.0
Server: Restlet-Framework/2.1.7
Content-Type: application/json;charset=UTF-8

{
  "tokenId":"AQIC5wM...TU30Q*",
  "successUrl":"/openam/console"
}
```

If the default REST API version behavior is set to **None**, and a REST API call does not include the **Accept-API-Version** header, or does not specify a **resource** version, then a **400 Bad Request** status code is returned, as shown below:

```
$ curl \
--header "Content-Type: application/json" \
--header "Accept-API-Version: protocol=1.0" \
https://openam.example.com:8443/openam/json/realms/root/serverinfo/*
{
  "code":400,
  "reason":"Bad Request",
  "message":"No requested version specified and behavior set to NONE."
}
```

If a REST API call does include the **Accept-API-Version** header, but the specified **resource** or **protocol** version does not exist in AM, then a **404 Not Found** status code is returned, as shown below:

```
$ curl \
--header "Content-Type: application/json" \
--header "Accept-API-Version: protocol=1.0, resource=999.0" \
https://openam.example.com:8443/openam/json/realms/root/serverinfo/*
{
  "code":404,
  "reason":"Not Found",
  "message":"Accept-API-Version: Requested version \"999.0\" does not match any routes."
}
```

### Tip

For more information on setting the default REST API version behavior, see "Specifying an Explicit REST API Version".

## Specifying Realms in REST API Calls

This section describes how to work with realms when making REST API calls to AM.

Realms can be specified in the following ways when making a REST API call to AM:

### DNS Alias

When making a REST API call, the DNS alias of a realm can be specified in the subdomain and domain name components of the REST endpoint.

To list all users in the top-level realm use the DNS alias of the AM instance, for example, the REST endpoint would be:

```
https://openam.example.com:8443/openam/json/users?_queryId=*
```

To list all users in a realm with DNS alias `suppliers.example.com` the REST endpoint would be:

```
https://suppliers.example.com:8443/openam/json/users?_queryId=*
```

### Path

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`.

To authenticate a user in the top-level realm, use the `root` keyword. For example:

```
https://openam.example.com:8443/openam/json/realms/root/authenticate
```

To authenticate a user in a subrealm named `customers` within the top-level realm, the REST endpoint would be:

```
https://openam.example.com:8443/openam/json/realms/root/realms/customers/authenticate
```

If realms are specified using both the DNS alias and path methods, the path is used to determine the realm.

For example, the following REST endpoint returns users in a subrealm of the top-level realm named `europe`, not the realm with DNS alias `suppliers.example.com`:

```
https://suppliers.example.com:8443/openam/json/realms/root/realms/europe/users?_queryId=*
```

## Authentication and Logout using REST

You can use REST-like APIs under `/json/authenticate` and `/json/sessions` for authentication and for logout.

The `/json/authenticate` endpoint does not support the CRUDPAQ verbs and therefore does not technically satisfy REST architectural requirements. The term *REST-like* describes this endpoint better than *REST*.

After a successful authentication, AM returns a `tokenId` object that applications can present as a cookie value for other operations that require authentication. This object is a session in the *Authentication and Single Sign-On Guide* token—a representation of the exchange of information and credentials between AM and the user or identity.

The type of `tokenId` returned varies depending on where AM stores the sessions for the realm to which the user authenticates:

- If CTS-based sessions are enabled, the `tokenId` object is a reference to the session state stored in the CTS token store.
- If client-based sessions are enabled, the `tokenId` object is the session state for that particular user or identity.

Developers should be aware that the size of the `tokenId` for client-based sessions—2000 bytes or greater—is considerably longer than for CTS-based sessions—approximately 100 bytes. For more information about session tokens, see "Session Cookies" in the *Authentication and Single Sign-On Guide*.

## Authenticating to AM using REST

To log in to AM using REST, make an HTTP POST request to the `json/authenticate` 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`.

AM uses the default authentication service configured for the realm. You can override the default by specifying authentication services and other options in the REST request.

AM provides both simple authentication methods, such as providing user name and password, and complex authentication journeys that may involve a tree with inner tree evaluation and/or multi-factor authentication.

For authentication journeys where providing a user name and password is enough, you can log in to AM using a `curl` command similar to the following:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "X-OpenAM-Username: demo" \
--header "X-OpenAM-Password: changeit" \
--header "Accept-API-Version: resource=2.0, protocol=1.0" \
'https://openam.example.com:8443/openam/json/realms/root/authenticate'
{
  "tokenId": "AQIC5w...NTcy*",
  "successUrl": "/openam/console",
  "realm": "/"
}
```

The user name and password are sent in headers. This zero page login mechanism works only for name/password authentication.

Note that the POST body is empty; otherwise, AM interprets the body as a continuation of an existing authentication attempt, one that uses a supported callback mechanism. AM implements callback mechanisms to support complex authentication journeys, such as those where the user needs to be redirected to a third party or interact with a device as part of multi-factor authentication.

When a client makes a call to the `/json/authenticate` endpoint appending a valid SSO token, AM returns the `tokenId` field **empty** when `HttpOnly` cookies are enabled. For example:

```
{
  "tokenId": "",
  "successUrl": "/openam/console",
  "realm": "/"
}
```

### Tip

#### About Success and Failure URLs

On authentication success, AM returns an SSO token and a success URL. By default, users are redirected to `/openam/console`.

No failure URL is configured by default. When configured, on authentication failure, AM returns HTTP status code 401 Unauthorized and the failure URL:

```
{
  "code": 401,
  "reason": "Unauthorized",
  "message": "Login failure",
  "failureUrl": "http://www.example.com/401.html"
}
```

For more information about configuring successful or failed authentication, see "Configuring Success and Failure Redirection URLs" in the *Authentication and Single Sign-On Guide*.

## Using UTF-8 User Names

To use UTF-8 user names and passwords in calls to the `/json/authenticate` endpoint, base64-encode the string, and then wrap the string as described in RFC 2047:

```
encoded-word = "=?" charset "?" encoding "?" encoded-text "=?"
```

For example, to authenticate using a UTF-8 username, such as `dēmjø`, perform the following steps:

1. Encode the string in base64 format: `yZfDq8mxw7g=`.
2. Wrap the base64-encoded string as per RFC 2047: `=?UTF-8?B?yZfDq8mxw7g=?=`.
3. Use the result in the `X-OpenAM-Username` header passed to the authentication endpoint as follows:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "X-OpenAM-Username: =?UTF-8?B?yZfDq8mxw7g=?=" \
--header "X-OpenAM-Password: changeit" \
--header "Accept-API-Version: resource=2.0, protocol=1.0" \
'https://openam.example.com:8443/openam/json/realms/root/authenticate'
{
  "tokenId": "AQIC5w...NTcy*",
  "successUrl": "/openam/console",
  "realm": "/"
}
```

## Authenticating to Specific Authentication Services

You can provide AM with additional information about how you are authenticating. For example, you can specify the authentication tree you want to use, or request from AM a list of the authentication services that would satisfy a particular authentication condition.

The following example shows how to specify the `ldapService` chain by using the `authIndexType` and `authIndexValue` query string parameters:

```
$ curl \
--request POST \
--header "X-OpenAM-Username: demo" \
--header "X-OpenAM-Password: changeit" \
--header 'Accept-API-Version: resource=2.0, protocol=1.0' \
'https://openam.example.com:8443/openam/json/realms/root/authenticate?
authIndexType=service&authIndexValue=ldapService'
```

You can exchange the `ldapService` chain with any other chain or tree.

For more information about using the `authIndexType` parameter to authenticate to specific services, see "Authenticate Endpoint Parameters".

## Authenticate Endpoint Parameters

To authenticate to AM using REST, make an HTTP POST request to the `/json/authenticate` 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`.

The following list describes the `/json/authenticate` endpoint supported parameters:

### `authIndexType`

Specifies the type of authentication the user will perform. Always use in conjunction with the `authIndexValue` parameter to provide additional information about the way the user is authenticating.

If not specified, AM authenticates the user against the default authentication service configured for the realm.

The `authIndexType` parameter supports the following types:

- `composite_advice`

Specifies that the value of the `authIndexValue` parameter is a URL-encoded composite advice string.

Use `composite_advice` when you want to give AM hints of which authentication services to use when logging in a user. For example, use an authentication service that provides an authentication level of 10 or higher:

```
$ curl -get \
--request POST \
--header "Content-Type: application/json" \
--header 'Accept-API-Version: resource=2.0, protocol=1.0' \
--data-urlencode 'authIndexType=composite_advice' \
--data-urlencode 'authIndexValue=<Advices>
  <AttributeValuePair>
    <Attribute name="AuthLevelConditionAdvice"/>
    <Value>10</Value>
  </AttributeValuePair>
</Advices>' \
'https://openam.example.com:8443/openam/json/realms/root/authenticate'
```

Note that the previous `curl` command URL-encodes the XML values, and the `-G` parameter appends them as query string parameters to the URL.

Possible options for advices are:

- **TransactionConditionAdvice**. Requires the unique ID of a transaction token. For example:

```
<Advices>
  <AttributeValuePair>
    <Attribute name="TransactionConditionAdvice"/>
    <Value>9dae2c80-fe7a-4a36-b57b-4fb1271b0687</Value>
  </AttributeValuePair>
</Advices>
```

For more information, see *"Implementing Transactional Authorization"* in the *Authorization Guide*.

- **AuthenticateToServiceConditionAdvice**. Requires the name of an authentication chain or tree. For example:

```
<Advices>
  <AttributeValuePair>
    <Attribute name="AuthenticateToServiceConditionAdvice"/>
    <Value>myExampleTree</Value>
  </AttributeValuePair>
</Advices>
```

- **AuthSchemeConditionAdvice**. Requires the name of an authentication module. For example:

```
<Advices>
  <AttributeValuePair>
    <Attribute name="AuthSchemeConditionAdvice"/>
    <Value>DataStoreModule</Value>
  </AttributeValuePair>
</Advices>
```

- **AuthenticateToRealmConditionAdvice**. Requires the name of a realm. For example:

```
<Advices>
  <AttributeValuePair>
    <Attribute name="AuthenticateToRealmConditionAdvice"/>
    <Value>myRealm</Value>
  </AttributeValuePair>
</Advices>
```

- **AuthLevelConditionAdvice**. Requires an authentication level. For example:

```
<Advices>
  <AttributeValuePair>
    <Attribute name="AuthLevelConditionAdvice"/>
    <Value>10</Value>
  </AttributeValuePair>
</Advices>
```

- **AuthenticateToTreeConditionAdvice**. Requires the name of an authentication tree. For example:

```
<Advices>
  <AttributeValuePair>
    <Attribute name="AuthenticateToTreeConditionAdvice"/>
    <Value>PersistentCookieTree</Value>
  </AttributeValuePair>
</Advices>
```

You can specify multiple advice conditions and combine them. For example:

```
<Advices>
  <AttributeValuePair>
    <Attribute name="AuthenticateToServiceConditionAdvice"/>
    <Value>ldapService</Value>
  </AttributeValuePair>
  <AttributeValuePair>
    <Attribute name="AuthenticateToServiceConditionAdvice"/>
    <Value>Example</Value>
  </AttributeValuePair>
  <AttributeValuePair>
    <Attribute name="AuthLevelConditionAdvice"/>
    <Value>10</Value>
  </AttributeValuePair>
</Advices>
```

- level

Specifies that the value of the `authIndexValue` parameter is the minimum authentication level an authentication service must satisfy to log in the user.

For example, to log into AM using an authentication service that provides a minimum authentication level of 10, you could use the following:

```
$ curl \
  --request POST \
  --header 'Accept-API-Version: resource=2.0, protocol=1.0' \
  'https://openam.example.com:8443/openam/json/realms/root/authenticate?
  authIndexType=level&authIndexValue=10'
```

- module

Specifies that the value of the `authIndexValue` parameter is the name of the authentication module AM must use to log in the user.

For example, to log into AM using the built-in `DataStore` authentication module, you could use the following:

```
$ curl \
--request POST \
--header 'Accept-API-Version: resource=2.0, protocol=1.0' \
'https://openam.example.com:8443/openam/json/realms/root/authenticate?
authIndexType=module&authIndexValue=DataStore'
```

- resource

Specifies that the value of the `authIndexValue` parameter is a URL protected by an AM policy.

For example, to log into AM using a policy matching the `http://www.example.com` resource, you could use the following:

```
$ curl \
--request POST \
--header 'Accept-API-Version: resource=2.0, protocol=1.0' \
'https://openam.example.com:8443/openam/json/realms/root/authenticate?
authIndexType=resource&authIndexValue=http%3A%2F%2Fwww.example.com'
```

Note that the resource must be URL-encoded. Authentication will fail if no policy matches the resource.

- service

Specifies that the value of the `authIndexValue` parameter is the name of an authentication tree or authentication chain AM must use to log in the user.

For example, to log in to AM using the built-in `ldapService` authentication chain, you could use the following:

```
$ curl \
--request POST \
--header 'Accept-API-Version: resource=2.0, protocol=1.0' \
'https://openam.example.com:8443/openam/json/realms/root/authenticate?
authIndexType=service&authIndexValue=ldapService'
```

- user

Specifies that the value of the `authIndexValue` parameter is a valid user ID. AM will then authenticate the user against the chain configured in the User Authentication Configuration field of that user's profile.

For example, for the user `demo` to log into AM using the chain specified in their user profile, you could use the following:



```
$ curl \
--request POST \
--header 'Accept-API-Version: resource=2.0, protocol=1.0' \
'https://openam.example.com:8443/openam/json/realms/root/authenticate?
authIndexType=user&authIndexValue=demo'
```

Authentication will fail if the User Authentication Configuration field is empty for the user.

If several authentication services that satisfy the authentication requirements are available, AM presents them as a choice callback to the user. Return the required callbacks to AM to authenticate.

Required: No.

#### **authIndexValue**

Specifies the value of the **authIndexType** parameter.

Required: Yes, when using the **authIndexType** parameter.

#### **noSession**

When set to **true**, specifies that AM should not return a session when authenticating a user. For example:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "Accept-API-Version: resource=2.0, protocol=1.0" \
--header "X-OpenAM-Username: demo" \
--header "X-OpenAM-Password: changeit" \
'https://openam.example.com:8443/openam/json/realms/root/authenticate?noSession=true'
{
  "message": "Authentication Successful",
  "successUrl": "/openam/console",
  "realm": "/"
}
```

Required: No.

## Returning Callback Information to AM

The `/json/authenticate` endpoint supports callback mechanisms to perform complex authentication journeys. Whenever AM needs to return or request information, it will return a JSON object with the authentication step, the authentication identifier, and the related callbacks.

The following types of callbacks are available:

- *Read-only callbacks.* AM uses read-only callbacks to provide information to the user, such as text messages or the amount of time that the user needs to wait before continuing their authentication journey.

- *Interactive callbacks.* AM uses interactive callbacks ask the user for information. For example, to request their user name and password, or to request that the user chooses between different options.
- *Backchannel callbacks.* AM uses backchannel callbacks when it needs to access additional information from the user's request. For example, when it requires a particular header or a certificate.

Read-only and interactive callbacks have an array of **output** elements suitable for displaying to the end user. The JSON returned in interactive callbacks also contains an array of **input** elements, which must be completed and returned to AM. For example:

```
"output": [
  {
    "name": "prompt",
    "value": " User Name: "
  }
],
"input": [
  {
    "name": "IDToken1",
    "value": ""
  }
]
```

The value of some interactive callbacks can be returned as headers, such as the **X-OpenAM-Username** and **X-OpenAM-Password** headers, but most of them must be returned in JSON as a response to the request.

Depending on how complex the authentication journey is, AM may return several callbacks sequentially. Each must be completed and returned to AM until authentication is successful.

The following example shows a request for authentication, and AM's response of the **NameCallback** and **PasswordCallback** callbacks:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "Accept-API-Version: resource=2.0, protocol=1.0" \
'https://openam.example.com:8443/openam/json/realms/root/authenticate'
```

```
{
  "authId": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJvdiIjOiJ... ", ❶
  "template": "", ❷
  "stage": "DataStore1", ❸
  "callbacks": [
    {
      "type": "NameCallback", ❹
      "output": [ ❺
        {
          "name": "prompt",
          "value": " User Name: "
        }
      ],
      "input": [ ❻
        {
          "name": "IDToken1",
```

```

        "value": ""
    }
  ]
},
{
  "type": "PasswordCallback", ❹
  "output": [ ❺
    {
      "name": "prompt",
      "value": " Password: "
    }
  ],
  "input": [ ❻
    {
      "name": "IDToken2",
      "value": ""
    }
  ]
}
]
}
}

```

### Key:

- ❶ The JWT that uniquely identifies the authentication context to AM.
- ❷ A template to customize the look of the authentication module, if exists. For more information, see [How do I customize the Login page?](#) in the *ForgeRock Knowledge Base*.
- ❸ The authentication module stage where the authentication journey is at the moment.
- ❹ The type of callback. It must be one the "Supported Callbacks".
- ❺ The information AM offers about this callback. Usually, this information would be displayed to the user in the UI.
- ❻ The information AM is requesting. The user must fill the `"value": ""` object with the required information.

To respond to a callback, send back the whole JSON object with the missing values filled. The following example shows how to respond to the `NameCallback` and `PasswordCallback` callbacks, with the `demo` and `changeit` values filled:

```

$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "Accept-API-Version: resource=2.0, protocol=1.0" \
--data '{
  "authId": ""eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJvdGsiOiJ...",
  "template": "",
  "stage": "DataStore1",
  "callbacks": [
    {
      "type": "NameCallback",
      "output": [
        {
          "name": "prompt",
          "value": " User Name: "
        }
      ]
    }
  ]
}

```

```

    ],
    "input": [
      {
        "name": "IDToken1",
        "value": "demo"
      }
    ]
  },
  {
    "type": "PasswordCallback",
    "output": [
      {
        "name": "prompt",
        "value": " Password: "
      }
    ],
    "input": [
      {
        "name": "IDToken2",
        "value": "changeit"
      }
    ]
  }
]
}
} \
'https://openam.example.com:8443/openam/json/realms/root/authenticate'
{
  "tokenId": "AQIC5wM2...U3MTE4NA..*",
  "successUrl": "/openam/console",
  "realm": "/"
}

```

On complex authentication journeys, AM may send several callbacks sequentially. Each must be completed and returned to AM until authentication is successful.

## Supported Callbacks

The following types of callbacks are available:

- Interactive Callbacks
- Read-only Callbacks
- Backchannel Callbacks

### *Interactive Callbacks*

AM returns the following callbacks to request information from the user:

#### **ChoiceCallback**

Used to display a list of choices and retrieve the selected choice. To indicate that the user selected the first choice, return a value of `0` to AM. For the second choice, return a value of `1`, and so forth. For example:

```
"callbacks":[
  {
    "type":"ChoiceCallback",
    "output":[
      {
        "name":"prompt",
        "value":"Choose one"
      },
      {
        "name":"choices",
        "value":[
          "Choice A",
          "Choice B",
          "Choice C"
        ]
      },
      {
        "name":"defaultChoice",
        "value":2
      }
    ],
    "input":[
      {
        "name":"IDToken1",
        "value":0
      }
    ]
  }
]
```

Class to import: `javax.security.auth.callback.ChoiceCallback`

## ConfirmationCallback

Used to ask for a boolean-style confirmation, such as yes/no or true/false, and retrieve the response. Also can present a "Cancel" option. To indicate that the user selected the first choice, return a value of `0` to AM. For the second choice, return a value of `1`, and so forth. For example:

```
"callbacks":[
  {
    "type":"ConfirmationCallback",
    "output":[
      {
        "name":"prompt",
        "value":""
      },
      {
        "name":"messageType",
        "value":0
      },
      {
        "name":"options",
        "value":[
          "Submit",
          "Start Over",
          "Cancel"
        ]
      }
    ]
  }
]
```

```
    },
    {
      "name": "optionType",
      "value": -1
    },
    {
      "name": "defaultOption",
      "value": 1
    }
  ],
  "input": [
    {
      "name": "IDToken2",
      "value": 0
    }
  ]
}
]
```

Class to import: `javax.security.auth.callback.ConfirmationCallback`

### NameCallback

Used to retrieve a data string which can be entered by the user. Usually used for collecting user names. For example:

```
"callbacks": [
  {
    "type": "NameCallback",
    "output": [
      {
        "name": "prompt",
        "value": "User Name"
      }
    ],
    "input": [
      {
        "name": "IDToken1",
        "value": ""
      }
    ]
  }
]
```

Class to import: `javax.security.auth.callback.NameCallback`

### PasswordCallback

Used to retrieve a password value. For example:

```
"callbacks":[
  {
    "type":"PasswordCallback",
    "output":[
      {
        "name":"prompt",
        "value":"Password"
      }
    ],
    "input":[
      {
        "name":"IDToken1",
        "value":""
      }
    ]
  }
]
```

Class to import: `javax.security.auth.callback.PasswordCallback`

### **TextInputCallback**

Used to retrieve text input from the end user. For example

```
"callbacks":[
  {
    "type":"TextInputCallback",
    "output":[
      {
        "name":"prompt",
        "value":"User Name"
      }
    ],
    "input":[
      {
        "name":"IDToken1",
        "value":""
      }
    ]
  }
]
```

Class to import: `javax.security.auth.callback.TextInputCallback`

### *Read-only Callbacks*

### **HiddenValueCallback**

Used to return form values that are not visually rendered to the end user. For example:

```
"callbacks":[
  {
    "type":"HiddenValueCallback",
    "output":[
      {
        "name":"value",
        "value":"6186c911-b3be-4dbc-8192-bdf251392072"
      },
      {
        "name":"id",
        "value":"jwt"
      }
    ],
    "input":[
      {
        "name":"IDToken1",
        "value":"jwt"
      }
    ]
  }
]
```

Class to import: `com.sun.identity.authentication.callbacks.HiddenValueCallback`

## MetadataCallback

Used to inject key-value meta data into the authentication process. For example:

```
"callbacks":[
  {
    "type":"MetadataCallback",
    "output":[
      {
        "name":"data",
        "value":{"
          "myParameter": "MyValue"
        }
      }
    ]
  }
]
```

Class to import: `com.sun.identity.authentication.spi.MetadataCallback`

## PollingWaitCallback

Tells the user the amount of time to wait before responding to the callback.



```
"callbacks":[
  {
    "type":"PollingWaitCallback",
    "output":[
      {
        "name":"waitTime",
        "value":"8000"
      },
      {
        "name":"message",
        "value":"Waiting for response..."
      }
    ]
  }
]
```

Class to import: `org.forgerock.openam.authentication.callbacks.PollingWaitCallback`

## RedirectCallback

Used to redirect the user's browser or user-agent.

```
"callbacks":[
  {
    "type":"RedirectCallback",
    "output":[
      {
        "name":"redirectUrl",
        "value":"https://accounts.google.com/o/oauth2/v2/auth?nonce..."
      },
      {
        "name":"redirectMethod",
        "value":"GET"
      },
      {
        "name":"trackingCookie",
        "value":true
      }
    ]
  }
]
```

Class to import: `com.sun.identity.authentication.spi.RedirectCallback`

## TextOutputCallback

Used to display a message to the end user.

```
"callbacks": [
  {
    "type": "TextOutputCallback",
    "output": [
      {
        "name": "message",
        "value": "Default message"
      },
      {
        "name": "messageType",
        "value": "0"
      }
    ]
  }
]
```

Class to import: `javax.security.auth.callback.TextOutputCallback`

## Backchannel Callbacks

AM uses backchannel callbacks when it needs to recover additional information from the user's request. For example, when it requires a particular header or a certificate.

### HttpCallback

Used to access user credentials sent in the Authorization header. For example:

```
Authorization: Basic bXlDbGllbnQ6Zm9yZ2Vybn2Nr
```

Class to import: `com.sun.identity.authentication.spi.HttpCallback`

### LanguageCallback

Used to retrieve the locale for localizing text presented to the end user. The locale is sent in the request as a header.

Class to import: `javax.security.auth.callback.LanguageCallback`

### ScriptTextOutputCallback

Used to insert a script into the page presented to the end user. The script can, for example, collect data about the user's environment.

Class to import: `com.sun.identity.authentication.callbacks.ScriptTextOutputCallback`

### X509CertificateCallback

Used to retrieve the content of an x.509 certificate, for example, from a header.

Class to import: `com.sun.identity.authentication.spi.X509CertificateCallback`

## Logging out of AM Using REST

Authenticated users can log out with the token cookie value and an HTTP POST to `/json/sessions/?_action=logout`:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "Cache-Control: no-cache" \
--header "iPlanetDirectoryPro: AQIC5wM2...U3MTE4NA..*" \
--header "Accept-API-Version: resource=3.1, protocol=1.0" \
https://openam.example.com:8443/openam/json/realms/root/sessions/?_action=logout
{
  "result": "Successfully logged out"
}
```

## Invalidating All Sessions for a Given User

To log out all sessions for a given user, first obtain a list of session handles of their active sessions, by performing an HTTP GET to the `/json/sessions/` endpoint, using the SSO token of an administrative user, such as `amAdmin` as the value of the `iPlanetDirectoryPro` header. You must also specify a `queryFilter` parameter.

The `queryFilter` parameter requires the name of the user, and the realm to search. For example, to obtain a list of session handles for a user named `demo` in the top-level realm, the query filter value would be:

```
username eq "demo" and realm eq "/"
```

### Note

The query filter value must be URL encoded when sent over HTTP.

For more information on query filter parameters, see "Query" in the *Authentication and Single Sign-On Guide*.

In the following example, there are two active sessions:

```
$ curl \
--request GET \
--header "Content-Type: application/json" \
--header "Cache-Control: no-cache" \
--header "iPlanetDirectoryPro: AQICS...NzEz*" \
--header "Accept-API-Version: resource=3.1, protocol=1.0" \
https://openam.example.com:8443/openam/json/realms/root/sessions?_queryFilter=username%20eq%20demo%22%20and%20realm%20eq%20%22%2F%22
{
  "result": [
    {
      "username": "demo",
      "universalId": "id=demo,ou=user,dc=openam,dc=forgerock,dc=org",
      "realm": "/",
      "sessionHandle": "shandle:SJ80.*AA...JT.*",
    }
  ]
}
```

```

    "latestAccessTime": "2018-10-23T09:37:54.387Z",
    "maxIdleExpirationTime": "2018-10-23T10:07:54Z",
    "maxSessionExpirationTime": "2018-10-23T11:37:54Z"
  },
  {
    "username": "demo",
    "universalId": "id=demo,ou=user,dc=openam,dc=forgerock,dc=org",
    "realm": "/",
    "sessionHandle": "shandle:H4CV.*DV...FM.*",
    "latestAccessTime": "2018-10-23T09:37:43.780Z",
    "maxIdleExpirationTime": "2018-10-23T10:07:43Z",
    "maxSessionExpirationTime": "2018-10-23T11:37:43Z"
  }
],
"resultCount": 2,
"pagedResultsCookie": null,
"totalPagedResultsPolicy": "NONE",
"totalPagedResults": -1,
"remainingPagedResults": -1
}

```

To log out all sessions for the specific user, perform an HTTP POST to the `/json/sessions/` endpoint, using the SSO token of an administrative user, such as `amAdmin` as the value of the `iPlanetDirectoryPro` header. You must also specify the `logoutByHandle` action, and include an array of the session handles to invalidate in the POST body, in a property named `sessionHandles`, as shown below:

```

$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "Cache-Control: no-cache" \
--header "iPlanetDirectoryPro: AQIC5w...NTcy*" \
--header "Accept-API-Version: resource=3.1, protocol=1.0" \
--data '{
  "sessionHandles": [
    "shandle:SJ80.*AA...JT.*",
    "shandle:H4CV.*DV...FM.*"
  ]
}' \
https://openam.example.com:8443/openam/json/realms/root/sessions/?_action=logoutByHandle
{
  "result": {
    "shandle:SJ80.*AA...JT.*": true,
    "shandle:H4CV.*DV...FM.*": true
  }
}

```

## Load Balancer and Proxy Layer Requirements

When authentication depends on the client IP address and AM lies behind a load balancer or proxy layer, configure the load balancer or proxy to send the address by using the `X-Forwarded-For` header, and configure AM to consume and forward the header as necessary. For details, see "Handling HTTP Request Headers" in the *Installation Guide*.

## Windows Desktop SSO Requirements

When authenticating with Windows Desktop SSO, add an **Authorization** header containing the string **Basic**, followed by a base64-encoded string of the username, a colon character, and the password. In the following example, the credentials **demo:changeit** are base64-encoded into the string **ZGVtbzpjajGFuZ2VpdA==**:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "Accept-API-Version: resource=2.0, protocol=1.0" \
--header "X-OpenAM-Username: demo" \
--header "X-OpenAM-Password: changeit" \
--header "Authorization: Basic ZGVtbzpjajGFuZ2VpdA==" \
'https://openam.example.com:8443/openam/json/realms/root/authenticate'
{
  "tokenId": "AQIC5w...NTcy*",
  "successUrl": "/openam/console",
  "realm": "/"
}
```

## Using the Session Token After Authentication

The following is a common scenario when accessing AM by using REST API calls:

- First, call the `/json/authenticate` endpoint to log a user in to AM. This REST API call returns a **tokenId** value, which is used in subsequent REST API calls to identify the user:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "X-OpenAM-Username: demo" \
--header "X-OpenAM-Password: changeit" \
--header "Accept-API-Version: resource=2.0, protocol=1.0" \
'https://openam.example.com:8443/openam/json/realms/root/authenticate'
{
  "tokenId": "AQIC5wM...TU30Q*",
  "successUrl": "/openam/console",
  "realm": "/"
}
```

The returned **tokenId** is known as a session token (also referred to as an SSO token). REST API calls made after successful authentication to AM must present the session token in the HTTP header as proof of authentication.

- Next, call one or more additional REST APIs on behalf of the logged-in user. Each REST API call passes the user's **tokenId** back to AM in the HTTP header as proof of previous authentication.

The following is a *partial* example of a **curl** command that inserts the token ID returned from a prior successful AM authentication attempt into the HTTP header:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "iPlanetDirectoryPro: AQIC5w...NTcy*" \
--header "Accept-API-Version: resource=2.0, protocol=1.0" \
--data '{
...

```

Observe that the session token is inserted into a header field named `iPlanetDirectoryPro`. This header field name must correspond to the name of the AM session cookie—by default, `iPlanetDirectoryPro`. You can find the cookie name in the AM console by navigating to `Deployment > Servers > Server Name > Security > Cookie`, in the `Cookie Name` field of the AM console.

Once a user has authenticated, it is *not* necessary to insert login credentials in the HTTP header in subsequent REST API calls. Note the absence of `X-OpenAM-Username` and `X-OpenAM-Password` headers in the preceding example.

Users are required to have appropriate privileges in order to access AM functionality using the REST API. For example, users who lack administrative privileges cannot create AM realms. For more information on the AM privilege model, see "Delegating Realm Administration Privileges" in the *Setup and Maintenance Guide*.

- Finally, call the REST API to log the user out of AM as described in "Authentication and Logout using REST". As with other REST API calls made after a user has authenticated, the REST API call to log out of AM requires the user's `tokenID` in the HTTP header.

## Server Information

You can retrieve AM server information by using HTTP GET on `/json/serverinfo/*` as follows:

```
$ curl \
--request GET \
--header "Content-Type: application/json" \
--header "Accept-API-Version: resource=1.1, protocol=1.0" \
https://openam.example.com:8443/openam/json/serverinfo/*
{
  "domains": [
    ".example.com"
  ],
  "protectedUserAttributes": [],
  "cookieName": "iPlanetDirectoryPro",
  "secureCookie": false,
  "forgotPassword": "false",
  "forgotUsername": "false",
  "kbaEnabled": "false",
  "selfRegistration": "false",
  "lang": "en-US",
  "successfulUserRegistrationDestination": "default",
  "socialImplementations": [
    {
      "iconPath": "XUI/images/logos/facebook.png",
      "authnChain": "FacebookSocialAuthenticationService",

```

```
        "displayName": "Facebook",
        "valid": true
    }
],
"referralsEnabled": "false",
"zeroPageLogin": {
    "enabled": false,
    "referrerWhitelist": [
        ""
    ],
    "allowedWithoutReferer": true
},
"realm": "/",
"xuiUserSessionValidationEnabled": true,
"FQDN": "openam.example.com"
}
```

## Token Encoding

Valid tokens in AM require configuration either in percent encoding or in *C66Encode* format. C66Encode format is encouraged. It is the default token format for AM, and is used in this section. The following is an example token that has not been encoded:

```
AQIC5wM2LY4SfzczntBbXvEA0uECbqMY3J4NW3byH6xwGkGE=@AAJTSQACMDE=#
```

This token includes reserved characters such as `+`, `/`, and `=` (The `@`, `#`, and `*` are not reserved characters per se, but substitutions are still required). To c66encode this token, you would substitute certain characters for others, as follows:

- `+` is replaced with `-`
- `/` is replaced with `_`
- `=` is replaced with `.`
- `@` is replaced with `*`
- `#` is replaced with `*`
- `*` (first instance) is replaced with `@`
- `*` (subsequent instances) is replaced with `#`

In this case, the translated token would appear as shown here:

```
AQIC5wM2LY4SfzczntBbXvEA0uECbqMY3J4NW3byH6xwGkGE.*AAJTSQACMDE.*
```

## Logging

AM supports two Audit Logging Services: a new common REST-based Audit Logging Service, and the legacy Logging Service, which is based on a Java SDK and is available in AM versions prior to OpenAM 13. The legacy Logging Service is deprecated.

Both audit facilities log AM REST API calls.

## Common Audit Logging of REST API Calls

AM logs information about all REST API calls to the `access` topic. For more information about AM audit topics, see "Audit Log Topics" in the *Setup and Maintenance Guide*.

Locate specific REST endpoints in the `http.path` log file property.

## Legacy Logging of REST API Calls

AM logs information about REST API calls to two files:

- **amRest.access**. Records accesses to a CREST endpoint, regardless of whether the request successfully reached the endpoint through policy authorization.

An `amRest.access` example is as follows:

```
$ cat openam/openam/log/amRest.access
#Version: 1.0
#Fields: time Data LoginID ContextID IPAddr LogLevel Domain LoggedBy MessageID ModuleName
NameID HostName
"2011-09-14 16:38:17" /home/user/openam/openam/log/ "cn=dsameuser,ou=DSAME Users,o=openam"
aa307b2dcb721d4201 "Not Available" INFO o=openam "cn=dsameuser,ou=DSAME Users,o=openam"
LOG-1 amRest.access "Not Available" 192.168.56.2
"2011-09-14 16:38:17" "Hello World" id=bjensen,ou=user,o=openam 8a4025a2b3af291d01 "Not Available"
INFO o=openam id=amadmin,ou=user,o=openam "Not Available" amRest.access "Not Available"
192.168.56.2
```

- **amRest.authz**. Records all CREST authorization results regardless of success. If a request has an entry in the `amRest.access` log, but no corresponding entry in `amRest.authz`, then that endpoint was not protected by an authorization filter and therefore the request was granted access to the resource.

The `amRest.authz` file contains the `Data` field, which specifies the authorization decision, resource, and type of action performed on that resource. The `Data` field has the following syntax:



```
("GRANT"|"DENY") > "RESOURCE | ACTION"
```

where

```
"GRANT > " is prepended to the entry if the request was allowed
"DENY > " is prepended to the entry if the request was not allowed
"RESOURCE" is "ResourceLocation | ResourceParameter"
  where
    "ResourceLocation" is the endpoint location (e.g., subrealm/applicationtypes)
    "ResourceParameter" is the ID of the resource being touched
    (e.g., myApplicationType) if applicable. Otherwise, this field is empty
    if touching the resource itself, such as in a query.
```

```
"ACTION" is "ActionType | ActionParameter"
```

where

```
"ActionType" is "CREATE||READ||UPDATE||DELETE||PATCH||ACTION||QUERY"
"ActionParameter" is one of the following depending on the ActionType:
  For CREATE: the new resource ID
  For READ: empty
  For UPDATE: the revision of the resource to update
  For DELETE: the revision of the resource to delete
  For PATCH: the revision of the resource to patch
  For ACTION: the actual action performed (e.g., "forgotPassword")
  For QUERY: the query ID if any
```

```
$ cat openam/openam/log/amRest.authz
```

```
#Version: 1.0
#Fields: time Data ContextID LoginID IPAddr LogLevel Domain MessageID LoggedBy NameID
ModuleName HostName
"2014-09-16 14:17:28" /var/root/openam/openam/log/ 7d3af9e799b6393301
"cn=dsameuser,ou=DSAME Users,dc=openam,dc=forgerock,dc=org" "Not Available" INFO
dc=openam,dc=forgerock,dc=org LOG-1 "cn=dsameuser,ou=DSAME Users,dc=openam,dc=forgerock,dc=org"
"Not Available" amRest.authz 10.0.1.5
"2014-09-16 15:56:12" "GRANT > sessions|ACTION|logout|AdminOnlyFilter" d3977a55a2ee18c201
id=amadmin,ou=user,dc=openam,dc=forgerock,dc=org "Not Available" INFO dc=openam,dc=forgerock,dc=org
OAuth2Provider-2 "cn=dsameuser,ou=DSAME Users,dc=openam,dc=forgerock,dc=org" "Not Available"
amRest.authz 127.0.0.1
"2014-09-16 15:56:40" "GRANT > sessions|ACTION|logout|AdminOnlyFilter" eedbc205bf51780001
id=amadmin,ou=user,dc=openam,dc=forgerock,dc=org "Not Available" INFO dc=openam,dc=forgerock,dc=org
OAuth2Provider-2 "cn=dsameuser,ou=DSAME Users,dc=openam,dc=forgerock,dc=org" "Not Available"
amRest.authz 127.0.0.1
```

AM also provides additional information in its debug notifications for accesses to any endpoint, depending on the message type (error, warning or message) including realm, user, and result of the operation.

## Reference

This reference section covers return codes and system settings relating to REST API support in AM.

### REST APIs

**amster** service name: `RestApis`

The following settings are available in this service:

### Default Resource Version

The API resource version to use when the REST request does not specify an explicit version. Choose from:

- **Latest**. If an explicit version is not specified, the latest resource version of an API is used.
- **Oldest**. If an explicit version is not specified, the oldest supported resource version of an API is used. Note that since APIs may be deprecated and fall out of support, the oldest *supported* version may not be the first version.
- **None**. If an explicit version is not specified, the request will not be handled and an error status is returned.

The possible values for this property are:

- **Latest**
- **Oldest**
- **None**

Default value: **Latest**

**amster** attribute: **defaultVersion**

### Warning Header

Whether to include a warning header in the response to a request which fails to include the **Accept-API-Version** header.

Default value: **false**

**amster** attribute: **warningHeader**

### API Descriptions

Whether API Explorer and API Docs are enabled in AM and how the documentation for them is generated. Dynamic generation includes descriptions from any custom services and authentication modules you may have added. Static generation only includes services and authentication modules that were present when AM was built. Note that dynamic documentation generation may not work in some application containers.

The possible values for this property are:

- **DYNAMIC**. Enabled with Dynamic Documentation
- **STATIC**. Enabled with Static Documentation

- `DISABLED`

Default value: `STATIC`

**amster** attribute: `descriptionsState`

## Default Protocol Version

The API protocol version to use when a REST request does not specify an explicit version. Choose from:

- `Oldest`. If an explicit version is not specified, the oldest protocol version is used.
- `Latest`. If an explicit version is not specified, the latest protocol version is used.
- `None`. If an explicit version is not specified, the request will not be handled and an error status is returned.

The possible values for this property are:

- `Oldest`
- `Latest`
- `None`

Default value: `Latest`

**amster** attribute: `defaultProtocolVersion`

## Enable CSRF Protection

If enabled, all non-read/query requests will require the X-Requested-With header to be present.

Requiring a non-standard header ensures requests can only be made via methods (XHR) that have stricter same-origin policy protections in Web browsers, preventing Cross-Site Request Forgery (CSRF) attacks. Without this filter, cross-origin requests are prevented by the use of the application/json Content-Type header, which is less robust.

Default value: `true`

**amster** attribute: `csrfFilterEnabled`

## Appendix B. About Scripting

You can use scripts for client-side and server-side authentication, policy conditions, and handling OpenID Connect claims.

### The Scripting Environment

AM supports scripts written in either JavaScript, or Groovy <sup>1</sup>, and the same variables and bindings are delivered to scripts of either language.

+ *How to determine the JavaScript Engine Version?*

You can use a script to check the version of the JavaScript engine AM is using. You could temporarily add the following script to a Scripted Decision node, for example, to output the engine version to the debug log:

```
var rhino = JavaImporter(  
    org.mozilla.javascript.Context  
)  
  
var currentContext = rhino.Context.getCurrentContext()  
var rhinoVersion = currentContext.getImplementationVersion()  
  
logger.error("JS Script Engine: " + rhinoVersion)  
  
outcome = "true"
```

<sup>1</sup>Scripts used for client-side authentication must be in written in JavaScript.

### Note

Ensure the following are listed in the Java class whitelist property of the scripting engine.

- `org.mozilla.javascript.Context`
- `org.forgerock.openam.scripting.timeouts.*`

To view the Java class whitelist, go to [Configure > Global Services > Scripting > Secondary Configurations](#). Select the script type, and on the Secondary Configurations tab, click `engineConfiguration`.

For information on the capabilities of the JavaScript engine AM uses, see [Mozilla Rhino](#).

### + *How to determine the Groovy Engine Version?*

You can use a script to check the version of the Groovy scripting engine AM is using. You could temporarily add the following script to a Scripted Decision node, for example, to output the engine version to the debug log:

```
logger.error("Groovy Script Engine: " + GroovySystem.version)
outcome = "true"
```

### Note

Ensure the following are listed in the Java class whitelist property of the scripting engine.

- `groovy.lang.GroovySystem`

To view the Java class whitelist, go to [Configure > Global Services > Scripting > Secondary Configurations](#). Select the script type, and on the Secondary Configurations tab, click `engineConfiguration`.

For information on the capabilities of the Groovy engine AM uses, see [Apache Groovy](#).

To access the functionality AM provides, import the required Java class or package, as follows:

#### *JavaScript*

```
var fr = JavaImporter(
    org.forgerock.openam.auth.node.api,
    javax.security.auth.callback.NameCallback
);
with (fr) {
    ...
}
```

#### *Groovy*

```
import org.forgerock.openam.auth.node.api.*;
import javax.security.auth.callback.NameCallback;
```

You may need to whitelist the classes you use in scripts. See "Security".

You can use scripts to modify default AM behavior in the following situations, also known as *contexts*:

### **Client-side Authentication**

Scripts that are executed on the client during authentication. Client-side scripts must be in JavaScript.

### **Server-side Authentication**

Scripts are included in an authentication module within a chain and are executed on the server during authentication.

### **Authentication Trees**

Scripts are included in an authentication node within a tree and are executed on the server during authentication.

### **Policy Condition**

Scripts used as conditions within policies.

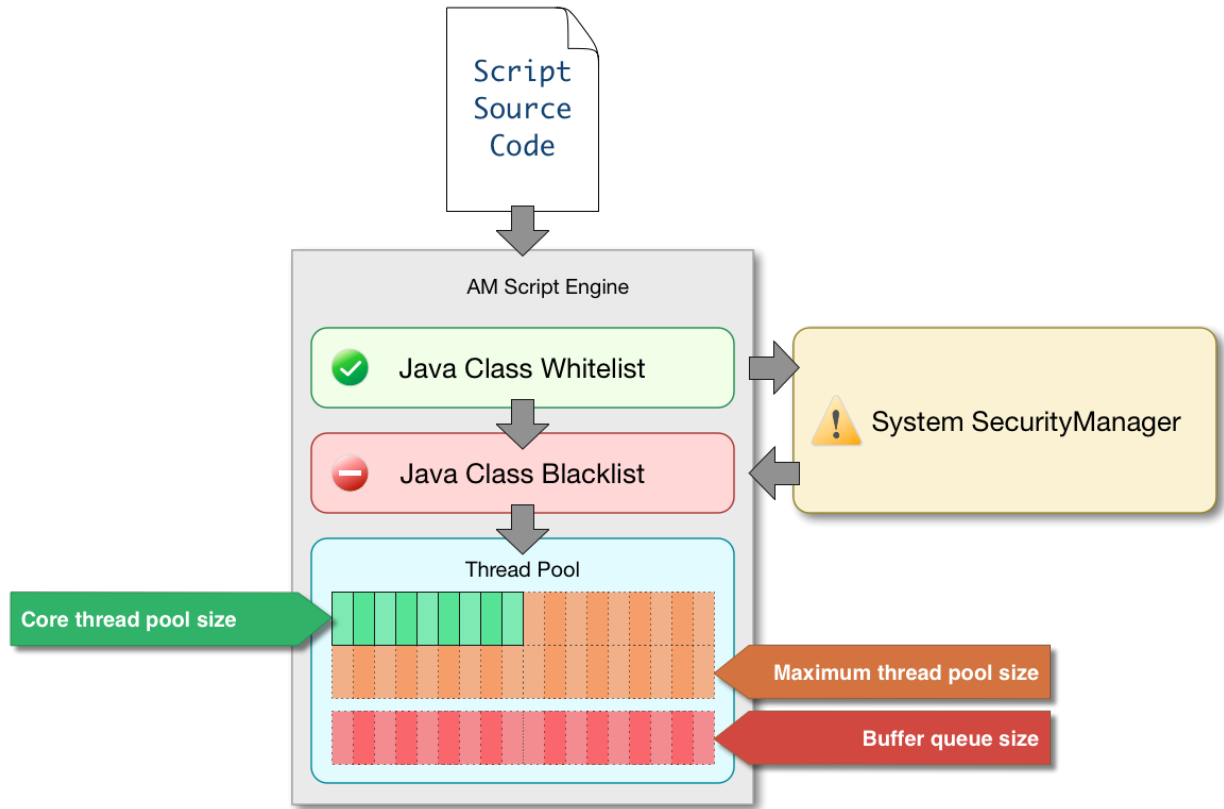
### **OIDC Claims**

Scripts that gather and populate the claims in a request when issuing an ID token or making a request to the `userinfo` endpoint.

For information on the global API, available to all script types, see "Global Scripting API Functionality".

AM implements a configurable scripting engine for each of the context types that are executed on the server.

The scripting engines in AM have two main components: security settings, and the thread pool.



## Security

AM scripting engines provide security features for ensuring that malicious Java classes are not directly called. The engines validate scripts by checking all directly-called Java classes against a configurable blacklist and whitelist, and, optionally, against the JVM SecurityManager, if it is configured.

Whitelists and blacklists contain class names that are allowed or denied execution respectively. Specify classes in whitelists and blacklists by name or by using regular expressions.

Classes called by the script are checked against the whitelist first, and must match at least one pattern in the list. The blacklist is applied after the whitelist, and classes matching any pattern are disallowed.

You can also configure the scripting engine to make an additional call to the JVM security manager for each class that is accessed. The security manager throws an exception if a class being called is not allowed to execute.

For more information on configuring script engine security, see "Scripting".

### Important Points About Script Engine Security

The following points should be considered when configuring the security settings within each script engine:

#### The scripting engine only validates directly accessible classes.

The security settings only apply to classes that the script *directly* accesses. If the script calls `Foo.a()` and then that method calls `Bar.b()`, the scripting engine will be unable to prevent it. You must consider the whole chain of accessible classes.

#### Note

Access includes actions such as:

- Importing or loading a class.
- Accessing any instance of that class. For example, passed as a parameter to the script.
- Calling a static method on that class.
- Calling a method on an instance of that class.
- Accessing a method or field that returns an instance of that class.

#### Potentially dangerous Java classes are blacklisted by default.

All Java reflection classes (`java.lang.Class`, `java.lang.reflect.*`) are blacklisted by default to avoid bypassing the security settings.

The `java.security.AccessController` class is also blacklisted by default to prevent access to the `doPrivileged()` methods.

#### Caution

You should not remove potentially dangerous Java classes from the blacklist.

#### The whitelists and blacklists match class or package names only.

The whitelist and blacklist patterns apply only to the exact class or package names involved. The script engine does not know anything about inheritance, so it is best to whitelist known, specific classes.

## Thread Pools

Each script is executed in an individual thread. Each scripting engine starts with an initial number of threads available for executing scripts. If no threads are available for execution, AM creates a new thread to execute the script, until the configured maximum number of threads is reached.



If the maximum number of threads is reached, pending script executions are queued in a number of buffer threads, until a thread becomes available for execution. If a created thread has completed script execution and has remained idle for a configured amount of time, AM terminates the thread, shrinking the pool.

For more information on configuring script engine thread pools, see "Scripting".

## Global Scripting API Functionality

This section covers functionality available to each of the server-side script types.

Global API functionality includes:

- Accessing HTTP Services
- Debug Logging

### Accessing HTTP Services

AM passes an HTTP client object, `httpClient`, to server-side scripts. Server-side scripts can call HTTP services with the `httpClient.send` method. The method returns an `HttpClientResponse` object.

Configure the parameters for the HTTP client object by using the `org.forgerock.http.protocol` package. This package contains the `Request` class, which has methods for setting the URI and type of request.

The following example, taken from the default server-side Scripted authentication module script, uses these methods to call an online API to determine the longitude and latitude of a user based on their postal address:

```
function getLongitudeLatitudeFromUserPostalAddress() {
    var request = new org.forgerock.http.protocol.Request();

    request.setUri("http://maps.googleapis.com/maps/api/geocode/json?address=" +
encodeURIComponent(userPostalAddress));
    request.setMethod("GET");

    var response = httpClient.send(request).get();
    logResponse(response);

    var geocode = JSON.parse(response.getEntity());
    var i;

    for (i = 0; i < geocode.results.length; i++) {
        var result = geocode.results[i];
        latitude = result.geometry.location.lat;
        longitude = result.geometry.location.lng;

        logger.message("latitude:" + latitude + " longitude:" + longitude);
    }
}
```

HTTP client requests are synchronous and blocking until they return. You can, however, set a global timeout for server-side scripts. For details, see "Scripted Authentication Module Properties" in the *Authentication and Single Sign-On Guide*.

Server-side scripts can access response data by using the methods listed in the table below.

### HTTP Client Response Methods

Method	Parameters	Return Type	Description
<code>HttpClientResponse.getCookies</code>	<code>Void</code>	<code>Map&lt;String, String&gt;</code>	Get the cookies for the returned response, if any exist.
<code>HttpClientResponse.getEntity</code>	<code>Void</code>	<code>String</code>	Get the entity of the returned response.
<code>HttpClientResponse.getHeaders</code>	<code>Void</code>	<code>Map&lt;String, String&gt;</code>	Get the headers for the returned response, if any exist.
<code>HttpClientResponse.getReasonPhrase</code>	<code>Void</code>	<code>String</code>	Get the reason phrase of the returned response.
<code>HttpClientResponse.getStatusCode</code>	<code>Void</code>	<code>Integer</code>	Get the status code of the returned response.
<code>HttpClientResponse.hasCookies</code>	<code>Void</code>	<code>Boolean</code>	Indicate whether the returned response had any cookies.
<code>HttpClientResponse.hasHeaders</code>	<code>Void</code>	<code>Boolean</code>	Indicate whether the returned response had any headers.

## Debug Logging

Server-side scripts can write messages to AM debug logs by using the `logger` object.

AM does not log debug messages from scripts by default. You can configure AM to log such messages by setting the debug log level for the `amScript` service. For details, see "Debug Logging By Service" in the *Setup and Maintenance Guide*.

The following table lists the `logger` methods.

### Logger Methods

Method	Parameters	Return Type	Description
<code>logger.error</code>	<code>Error Message</code> (type: <code>String</code> )	<code>Void</code>	Write <code>Error Message</code> to AM debug logs if ERROR level logging is enabled.

Method	Parameters	Return Type	Description
<code>logger.errorEnabled</code>	<code>Void</code>	<code>Boolean</code>	Return <code>true</code> when ERROR level debug messages are enabled.
<code>logger.message</code>	<code>Message</code> (type: <code>String</code> )	<code>Void</code>	Write <code>Message</code> to AM debug logs if MESSAGE level logging is enabled.
<code>logger.messageEnabled</code>	<code>Void</code>	<code>Boolean</code>	Return <code>true</code> when MESSAGE level debug messages are enabled.
<code>logger.warning</code>	<code>Warning Message</code> (type: <code>String</code> )	<code>Void</code>	Write <code>Warning Message</code> to AM debug logs if WARNING level logging is enabled.
<code>logger.warningEnabled</code>	<code>Void</code>	<code>Boolean</code>	Return <code>true</code> when WARNING level debug messages are enabled.

## Managing Scripts

This section shows you how to manage scripts used for client-side and server-side scripted authentication, custom policy conditions, and handling OpenID Connect claims using the AM console, the `ssoadm` command, and the REST API.

### Managing Scripts With the AM Console

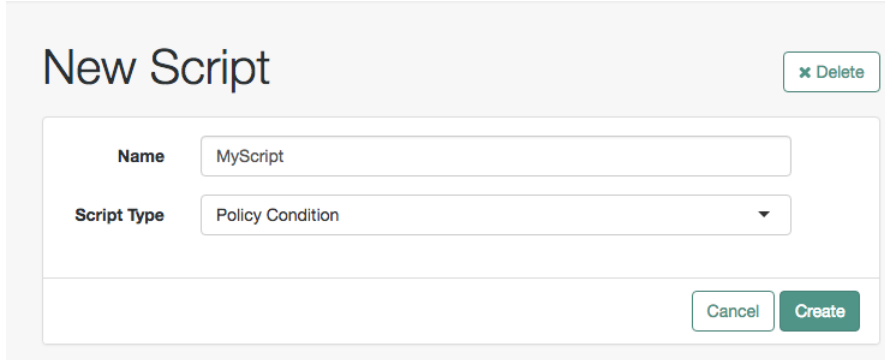
The following procedures describe how to create, modify, and delete scripts using the AM console:

- "To Create Scripts by Using the AM Console"
- "To Modify Scripts by Using the AM Console"
- "To Delete Scripts by Using the AM Console"

#### To Create Scripts by Using the AM Console

1. Log in to the AM console as an AM administrator, for example, `amadmin`.
2. Navigate to Realms > *Realm Name* > Scripts.
3. Click New Script.

The New Script page appears:




New Script ✕ Delete

Name

Script Type

4. Specify a name for the script.
5. Select the type of script from the Script Type drop-down list.
6. Click Create.

The *Script Name* page appears:



SCRIPT

## MyScript

✕ Delete

**Name**

**Description**

**Script Type** Policy Condition ⚙️ Change

**Language**  JavaScript  Groovy

**Script**

```

1  /**
2  * This is a Policy Condition example script. It demon
3  * use that information in external HTTP calls and mak
4  */
5
6  var userAddress, userIP, resourceHost;
7
8  if (validateAndInitializeParameters()) {
9
10     var countryFromUserAddress = getCountryFromUserAdd
11     logger.message("Country retrieved from user's addr
12     var countryFromUserIP = getCountryFromUserIP();
13     logger.message("Country retrieved from user's IP:
14     var countryFromResourceURI = getCountryFromResourc
15     logger.message("Country retrieved from resource UR
16
17     if (countryFromUserAddress === countryFromUserIP &
18         logger.message("Authorization Succeeded");
19         responseAttributes.put("countryOfOrigin", {cou
20         authorized = true;
21     } else {

```

Upload
Validate
🗒️ Edit Fullscreen

Save Changes

7. Enter values on the *Script Name* page as follows:

- a. Enter a description of the script.
- b. Choose the script language, either JavaScript or Groovy. Note that not every script type supports both languages.
- c. Enter the source code in the Script field.

On supported browsers, you can click Upload, navigate to the script file, and then click Open to upload the contents to the Script field.

- d. Click Validate to check for compilation errors in the script.

Correct any compilation errors, and revalidate the script until all errors have been fixed.

- e. Save your changes.

### *To Modify Scripts by Using the AM Console*

1. Log in to the AM console as an AM administrator, for example, `amadmin`.
2. Navigate to Realms > *Realm Name* > Scripts.
3. Select the script you want to modify from the list of scripts.

The *Script Name* page appears.

4. Modify values on the *Script Name* page as needed. Note that if you change the Script Type, existing code in the script is replaced.
5. If you modified the code in the script, click Validate to check for compilation errors.

Correct any compilation errors, and revalidate the script until all errors have been fixed.

6. Save your changes.

### *To Delete Scripts by Using the AM Console*

1. Log in to the AM console as an AM administrator, for example, `amadmin`.
2. Navigate to Realms > *Realm Name* > Scripts.
3. Choose one or more scripts to delete by activating the checkboxes in the relevant rows. Note that you can only delete user-created scripts—you cannot delete the global sample scripts provided with AM.
4. Click Delete.

## Managing Scripts With the REST API

This section shows you how to manage scripts used for client-side and server-side scripted authentication, custom policy conditions, and handling OpenID Connect claims by using the REST API.

AM provides the `scripts` REST endpoint for the following:

- "Querying Scripts"
- "Reading a Script"

- "Validating a Script"
- "Creating a Script"
- "Updating a Script"
- "Deleting a Script"

User-created scripts are realm-specific, hence the URI for the scripts' API can contain a realm component, such as `/json{/realm}/scripts`. If the realm is not specified in the URI, the top level realm is used.

#### Tip

AM includes some global example scripts that can be used in any realm.

Scripts are represented in JSON and take the following form. Scripts are built from standard JSON objects and values (strings, numbers, objects, sets, arrays, `true`, `false`, and `null`). Each script has a system-generated *universally unique identifier* (UUID), which must be used when modifying existing scripts. Renaming a script will not affect the UUID:

```
{
  "_id": "7e3d7067-d50f-4674-8c76-a3e13a810c33",
  "name": "Scripted Module - Server Side",
  "description": "Default global script for server side Scripted Authentication Module",
  "script": "dmFyIFNUNUQVJUX1RJ...",
  "language": "JAVASCRIPT",
  "context": "AUTHENTICATION_SERVER_SIDE",
  "createdBy": "id=dsameuser,ou=user,dc=openam,dc=forgerock,dc=org",
  "creationDate": 1433147666269,
  "lastModifiedBy": "id=dsameuser,ou=user,dc=openam,dc=forgerock,dc=org",
  "lastModifiedDate": 1433147666269
}
```

The values for the fields shown in the example above are explained below:

#### `_id`

The UUID that AM generates for the script.

#### `name`

The name provided for the script.

#### `description`

An optional text string to help identify the script.

#### `script`

The source code of the script. The source code is in UTF-8 format and encoded into Base64.

For example, a script such as the following:

```
var a = 123;  
var b = 456;
```

When encoded into Base64 becomes:

```
dmFyIGEGPSAxMjM7IA0KdmFyIGIyPSA0NTY7
```

### Language

The language the script is written in - [JAVASCRIPT](#) or [GROOVY](#).

### Language Support per Context

Script Context	Supported Languages
<a href="#">POLICY_CONDITION</a>	<a href="#">JAVASCRIPT</a> , <a href="#">GROOVY</a>
<a href="#">AUTHENTICATION_SERVER_SIDE</a>	<a href="#">JAVASCRIPT</a> , <a href="#">GROOVY</a>
<a href="#">AUTHENTICATION_CLIENT_SIDE</a>	<a href="#">JAVASCRIPT</a>
<a href="#">OIDC_CLAIMS</a>	<a href="#">JAVASCRIPT</a> , <a href="#">GROOVY</a>
<a href="#">AUTHENTICATION_TREE_DECISION_NODE</a>	<a href="#">JAVASCRIPT</a> , <a href="#">GROOVY</a>

### context

The context type of the script.

Supported values are:

#### [POLICY\\_CONDITION](#)

Policy Condition

#### [AUTHENTICATION\\_SERVER\\_SIDE](#)

Server-side Authentication

#### [AUTHENTICATION\\_CLIENT\\_SIDE](#)

Client-side Authentication

#### Note

Client-side scripts must be written in JavaScript.

#### [OIDC\\_CLAIMS](#)

OIDC Claims



## AUTHENTICATION\_TREE\_DECISION\_NODE

Authentication scripts used by Scripted Tree Decision authentication nodes.

### createdBy

A string containing the universal identifier DN of the subject that created the script.

### creationDate

An integer containing the creation date and time, in ISO 8601 format.

### lastModifiedBy

A string containing the universal identifier DN of the subject that most recently updated the resource type.

If the script has not been modified since it was created, this property will have the same value as `createdBy`.

### lastModifiedDate

A string containing the last modified date and time, in ISO 8601 format.

If the script has not been modified since it was created, this property will have the same value as `creationDate`.

## Querying Scripts

To list all the scripts in a realm, as well as any global scripts, perform an HTTP GET to the `/json/{realm}/scripts` endpoint with a `_queryFilter` parameter set to `true`.

### Note

If the realm is not specified in the URL, AM returns scripts in the top level realm, as well as any global scripts.

The `iPlanetDirectoryPro` header is required and should contain the SSO token of an administrative user, such as `amAdmin`, who has access to perform the operation.

```
$ curl \
--header "iPlanetDirectoryPro: AQIC5..." \
--header "Accept-API-Version: resource=1.1" \
https://openam.example.com:8443/openam/json/realm/root/realm/myrealm/scripts?_queryFilter=true
{
  "result": [
    {
      "_id": "9de3eb62-f131-4fac-a294-7bd170fd4acb",
      "name": "Scripted Policy Condition",
      "description": "Default global script for Scripted Policy Conditions",
```

```

"script": "LyoqCiAqIFRoaxMg...",
"language": "JAVASCRIPT",
"context": "POLICY_CONDITION",
"createdBy": "id=dsameuser,ou=user,dc=openam,dc=forgerock,dc=org",
"creationDate": 1433147666269,
"lastModifiedBy": "id=dsameuser,ou=user,dc=openam,dc=forgerock,dc=org",
"lastModifiedDate": 1433147666269
},
{
  "_id": "7e3d7067-d50f-4674-8c76-a3e13a810c33",
  "name": "Scripted Module - Server Side",
  "description": "Default global script for server side Scripted Authentication Module",
  "script": "dmFyIFNUQVJUX1RJ...",
  "language": "JAVASCRIPT",
  "context": "AUTHENTICATION_SERVER_SIDE",
  "createdBy": "id=dsameuser,ou=user,dc=openam,dc=forgerock,dc=org",
  "creationDate": 1433147666269,
  "lastModifiedBy": "id=dsameuser,ou=user,dc=openam,dc=forgerock,dc=org",
  "lastModifiedDate": 1433147666269
}
],
"resultCount": 2,
"pagedResultsCookie": null,
"remainingPagedResults": -1
}

```

### Supported `_queryFilter` Fields and Operators

Field	Supported Operators
<code>_id</code>	Equals (eq), Contains (co), Starts with (sw)
<code>name</code>	Equals (eq), Contains (co), Starts with (sw)
<code>description</code>	Equals (eq), Contains (co), Starts with (sw)
<code>script</code>	Equals (eq), Contains (co), Starts with (sw)
<code>language</code>	Equals (eq), Contains (co), Starts with (sw)
<code>context</code>	Equals (eq), Contains (co), Starts with (sw)

## Reading a Script

To read an individual script in a realm, perform an HTTP GET using the `/json{/realm}/scripts` endpoint, specifying the UUID in the URL.

### Tip

To read a script in the top-level realm, or to read a built-in global script, do not specify a realm in the URL.

The `iPlanetDirectoryPro` header is required and should contain the SSO token of an administrative user, such as `amAdmin`, who has access to perform the operation.

```
$ curl \
--header "iPlanetDirectoryPro: AQIC5..." \
--header "Accept-API-Version: resource=1.1" \
https://openam.example.com:8443/openam/json/realms/root/realms/myrealm/scripts/9de3eb62-f131-4fac-a294-7bd170fd4acb
{
  "_id": "9de3eb62-f131-4fac-a294-7bd170fd4acb",
  "name": "Scripted Policy Condition",
  "description": "Default global script for Scripted Policy Conditions",
  "script": "Ly0qCiAqIFRoaxMg...",
  "language": "JAVASCRIPT",
  "context": "POLICY_CONDITION",
  "createdBy": "id=dsameuser,ou=user,dc=openam,dc=forgerock,dc=org",
  "creationDate": 1433147666269,
  "lastModifiedBy": "id=dsameuser,ou=user,dc=openam,dc=forgerock,dc=org",
  "lastModifiedDate": 1433147666269
}
```

## Validating a Script

To validate a script, perform an HTTP POST using the `/json{/realm}/scripts` endpoint, with an `_action` parameter set to `validate`. Include a JSON representation of the script and the script language, `JAVASCRIPT` or `GR00VY`, in the POST data.

The value for `script` must be in UTF-8 format and then encoded into Base64.

The `iPlanetDirectoryPro` header is required and should contain the SSO token of an administrative user, such as `amAdmin`, who has access to perform the operation.

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "iPlanetDirectoryPro: AQIC5..." \
--header "Accept-API-Version: resource=1.1" \
--data '{
  "script": "dmFyIGEGPSAxMjM7dmFyIGIyPSA0NTY7Cg==",
  "language": "JAVASCRIPT"
}' \
https://openam.example.com:8443/openam/json/realms/root/realms/myrealm/scripts/?_action=validate
{
  "success": true
}
```

If the script is valid the JSON response contains a `success` key with a value of `true`.

If the script is invalid the JSON response contains a `success` key with a value of `false`, and an indication of the problem and where it occurs, as shown below:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "iPlanetDirectoryPro: AQIC5..." \
--header "Accept-API-Version: resource=1.1" \
--data '{
  "script": "dmFyIGEGPSAxMjM7dmFyIGIyPSA0NTY7ID1WQUxJREFUSU90IFNIT1VMRCBGQULMPQo=",
  "language": "JAVASCRIPT"
}' \
https://openam.example.com:8443/openam/json/realms/root/realms/myrealm/scripts/?_action=validate
{
  "success": false,
  "errors": [
    {
      "line": 1,
      "column": 27,
      "message": "syntax error"
    }
  ]
}
```

## Creating a Script

To create a script in a realm, perform an HTTP POST using the `/json{/realm}/scripts` endpoint, with an `_action` parameter set to `create`. Include a JSON representation of the script in the POST data.

The value for `script` must be in UTF-8 format and then encoded into Base64.

### Note

If the realm is not specified in the URL, AM creates the script in the top level realm.

The `iPlanetDirectoryPro` header is required and should contain the SSO token of an administrative user, such as `amAdmin`, who has access to perform the operation.

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "iPlanetDirectoryPro: AQIC5..." \
--header "Accept-API-Version: resource=1.1" \
--data '{
  "name": "MyJavaScript",
  "script": "dmFyIGEGPSAxMjM7CnZhciBiID0gNDU2Ow==",
  "language": "JAVASCRIPT",
  "context": "POLICY_CONDITION",
  "description": "An example script"
}' \
https://openam.example.com:8443/openam/json/realms/root/realms/myrealm/scripts/?_action=create
{
  "_id": "0168d494-015a-420f-ae5a-6a2a5c1126af",
  "name": "MyJavaScript",
  "description": "An example script",
  "script": "dmFyIGEGPSAxMjM7CnZhciBiID0gNDU2Ow==",
  "language": "JAVASCRIPT",
  "context": "POLICY_CONDITION",
  "createdBy": "id=amadmin,ou=user,dc=openam,dc=forgerock,dc=org",
  "creationDate": 1436807766258,
  "lastModifiedBy": "id=amadmin,ou=user,dc=openam,dc=forgerock,dc=org",
  "lastModifiedDate": 1436807766258
}
```

## Updating a Script

To update an individual script in a realm, perform an HTTP PUT using the `/json{/realm}/scripts` endpoint, specifying the UUID in both the URL and the PUT body. Include a JSON representation of the updated script in the PUT data, alongside the UUID.

### Note

If the realm is not specified in the URL, AM uses the top level realm.

The `iPlanetDirectoryPro` header is required and should contain the SSO token of an administrative user, such as `amAdmin`, who has access to perform the operation.

```
$ curl \
--header "iPlanetDirectoryPro: AQIC5..." \
--header "Content-Type: application/json" \
--header "Accept-API-Version: resource=1.1" \
--request PUT \
--data '{
  "name": "MyUpdatedJavaScript",
  "script": "dmFyIGEGPSAXMjM7CnZhciBiID0gNDU2Ow==",
  "language": "JAVASCRIPT",
  "context": "POLICY_CONDITION",
  "description": "An updated example script configuration"
}' \
https://openam.example.com:8443/openam/json/realms/root/realms/myrealm/scripts/0168d494-015a-420f-ae5a-6a2a5c1126af
{
  "_id": "0168d494-015a-420f-ae5a-6a2a5c1126af",
  "name": "MyUpdatedJavaScript",
  "description": "An updated example script configuration",
  "script": "dmFyIGEGPSAXMjM7CnZhciBiID0gNDU2Ow==",
  "language": "JAVASCRIPT",
  "context": "POLICY_CONDITION",
  "createdBy": "id=amadmin,ou=user,dc=openam,dc=forgerock,dc=org",
  "creationDate": 1436807766258,
  "lastModifiedBy": "id=amadmin,ou=user,dc=openam,dc=forgerock,dc=org",
  "lastModifiedDate": 1436808364681
}
```

## Deleting a Script

To delete an individual script in a realm, perform an HTTP DELETE using the `/json{/realm}/scripts` endpoint, specifying the UUID in the URL.

### Note

If the realm is not specified in the URL, AM uses the top level realm.

The `iPlanetDirectoryPro` header is required and should contain the SSO token of an administrative user, such as `amAdmin`, who has access to perform the operation.

```
$ curl \
--request DELETE \
--header "iPlanetDirectoryPro: AQIC5..." \
--header "Accept-API-Version: resource=1.1" \
https://openam.example.com:8443/openam/json/realms/root/realms/myrealm/scripts/0168d494-015a-420f-ae5a-6a2a5c1126af
{}
```

## Managing Scripts With the `ssoadm` Command

Use the `ssoadm` command's `create-sub-cfg`, `get-sub-cfg`, and `delete-sub-cfg` subcommands to manage AM scripts.

Create an AM script as follows:

1. Create a script configuration file, for example `/path/to/myScriptConfigurationFile.txt`, containing the following:

```
script-file=/path/to/myScriptFile.js
language=JAVASCRIPT ❶
name=My New Script
context=AUTHENTICATION_SERVER_SIDE ❷
```

- ❶ Possible values for the `language` property are:

- JAVASCRIPT
- GROOVY

- ❷ Possible values for the `context` property are:

- POLICY\_CONDITION
- AUTHENTICATION\_SERVER\_SIDE
- AUTHENTICATION\_CLIENT\_SIDE
- OIDC\_CLAIMS
- AUTHENTICATION\_TREE\_DECISION\_NODE

2. Run the `ssoadm create-sub-cfg` command. The `--datafile` argument references the script configuration file you created in the previous step:

```
$ ssoadm \
  create-sub-cfg \
  --realm /myRealm \
  --adminid amadmin \
  --password-file /tmp/pwd.txt \
  --servicename ScriptingService \
  --subconfigname scriptConfigurations/scriptConfiguration \
  --subconfigid myScriptID \
  --datafile /path/to/myScriptConfigurationFile.txt
Sub Configuration scriptConfigurations/scriptConfiguration was added to realm /myRealm
```

To list the properties of a script, run the `ssoadm get-sub-cfg` command:

```
$ ssoadm \  
  get-sub-cfg \  
    --realm /myRealm \  
    --adminid amadmin \  
    --password-file /tmp/pwd.txt \  
    --servicename ScriptingService \  
    --subconfigname scriptConfigurations/myScriptID  
createdBy=  
lastModifiedDate=  
lastModifiedBy=  
name=My New Script  
context=AUTHENTICATION_SERVER_SIDE  
description=  
language=JAVASCRIPT  
creationDate=  
script=...Script output follows...
```

To delete a script, run the **ssoadm delete-sub-cfg** command:

```
$ ssoadm \  
  delete-sub-cfg \  
    --realm /myRealm \  
    --adminid amadmin \  
    --password-file /tmp/pwd.txt \  
    --servicename ScriptingService \  
    --subconfigname scriptConfigurations/myScriptID  
Sub Configuration scriptConfigurations/myScriptID was deleted from realm /myRealm
```

## Scripting

**amster** service name: `Scripting`

### Configuration

The following settings appear on the **Configuration** tab:

#### Default Script Type

The default script context type when creating a new script.

The possible values for this property are:

- `POLICY_CONDITION`. Policy Condition
- `AUTHENTICATION_SERVER_SIDE`. Server-side Authentication
- `AUTHENTICATION_CLIENT_SIDE`. Client-side Authentication
- `OIDC_CLAIMS`. OIDC Claims
- `AUTHENTICATION_TREE_DECISION_NODE`. Decision node script for authentication trees



- `OAuth2_ACCESS_TOKEN_MODIFICATION`. OAuth2 Access Token Modification

Default value: `POLICY_CONDITION`

**amster** attribute: `defaultContext`

## Secondary Configurations

This service has the following Secondary Configurations.

## Engine Configuration

The following properties are available for Scripting Service secondary configuration instances:

### Engine Configuration

Configure script engine parameters for running a particular script type in AM.

**ssoadm** attribute: `engineConfiguration`

To access a secondary configuration instance using the **ssoadm** command, use: `--subconfigname [primary configuration]/[secondary configuration]` For example:

```
$ ssoadm set-sub-cfg \  
--adminid amAdmin \  
--password-file admin_pwd_file \  
--servicename ScriptingService \  
--subconfigname OIDC_CLAIMS/engineConfiguration \  
--operation set \  
--attributevalues maxThreads=300 queueSize=-1
```

#### Note

Supports server-side scripts only. AM cannot configure engine settings for client-side scripts.

The configurable engine settings are as follows:

### Server-side Script Timeout

The maximum execution time any individual script should take on the server (in seconds). AM terminates scripts which take longer to run than this value.

**ssoadm** attribute: `serverTimeout`

### Core thread pool size

The initial number of threads in the thread pool from which scripts operate. AM will ensure the pool contains at least this many threads.

**ssoadm** attribute: `coreThreads`

### Maximum thread pool size

The maximum number of threads in the thread pool from which scripts operate. If no free thread is available in the pool, AM creates new threads in the pool for script execution up to the configured maximum. It is recommended to set the maximum number of threads to 300.

**ssoadm** attribute: `maxThreads`

### Thread pool queue size

Specifies the number of threads to use for buffering script execution requests when the maximum thread pool size is reached.

For short, CPU-bound scripts, consider a small pool size and larger queue length. For I/O-bound scripts, for example, REST calls, consider a larger maximum pool size and a smaller queue.

Not hot-swappable: restart server for changes to take effect.

**ssoadm** attribute: `queueSize`

### Thread idle timeout (seconds)

Length of time (in seconds) for a thread to be idle before AM terminates created threads. If the current pool size contains the number of threads set in `Core thread pool size` idle threads will not be terminated, to maintain the initial pool size.

**ssoadm** attribute: `idleTimeout`

### Java class whitelist

Specifies the list of class-name patterns allowed to be invoked by the script. Every class accessed by the script must match at least one of these patterns.

You can specify the class name as-is or use a regular expression.

**ssoadm** attribute: `whiteList`

### Java class blacklist

Specifies the list of class-name patterns that are NOT allowed to be invoked by the script. The blacklist is applied AFTER the whitelist to exclude those classes - access to a class specified in both the whitelist and the blacklist will be denied.

You can specify the class name to exclude as-is or use a regular expression.

**ssoadm** attribute: `blackList`

## Use system SecurityManager

If enabled, AM will make a call to `System.getSecurityManager().checkPackageAccess(...)` for each class that is accessed. The method throws `SecurityException` if the calling thread is not allowed to access the package.

### Note

This feature only takes effect if the security manager is enabled for the JVM.

**ssoadm** attribute: `useSecurityManager`

## Scripting languages

Select the languages available for scripts on the chosen type. Either `GROOVY` or `JAVASCRIPT`.

**ssoadm** attribute: `languages`

## Default Script

The source code that is presented as the default when creating a new script of this type.

**ssoadm** attribute: `defaultScript`

## Appendix C. Getting Support

ForgeRock provides support services, professional services, training through ForgeRock University, and partner services to assist you in setting up and maintaining your deployments. For a general overview of these services, see <https://www.forgerock.com>.

ForgeRock has staff members around the globe who support our international customers and partners. For details on ForgeRock's support offering, including support plans and service level agreements (SLAs), visit <https://www.forgerock.com/support>.

ForgeRock publishes comprehensive documentation online:

- The ForgeRock Knowledge Base offers a large and increasing number of up-to-date, practical articles that help you deploy and manage ForgeRock software.

While many articles are visible to community members, ForgeRock customers have access to much more, including advanced information for customers using ForgeRock software in a mission-critical capacity.

- ForgeRock product documentation, such as this document, aims to be technically accurate and complete with respect to the software documented. It is visible to everyone and covers all product features and examples of how to use them.

# 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	<p>In general terms, a service exposing protected resources.</p> <p>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.</p>
Application type	<p>Application types act as templates for creating policy applications.</p> <p>Application types define a preset list of actions and functional logic, such as policy lookup and resource comparator logic.</p>

---

	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 <i>reference</i> 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

---

	<p>request. For browser-based clients, AM sets a cookie in the browser that contains the session information.</p> <p>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.</p>
Conditions	<p>Defined as part of policies, these determine the circumstances under which which a policy applies.</p> <p>Environmental conditions reflect circumstances like the client IP address, time of day, how the subject authenticated, or the authentication level achieved.</p> <p>Subject conditions reflect characteristics of the subject like whether the subject authenticated, the identity of the subject, or claims in the subject's JWT.</p>
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 <i>reference</i> 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	<p>Group of AM servers configured the same way, accessed through a load balancer layer. The load balancer handles failover to provide service-level availability.</p> <p>The load balancer can also be used to protect AM services.</p>
Standard metadata	Standard federation configuration information that you can share with other access management software.
Stateless Service	<p>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.</p> <p>All AM services are stateless unless otherwise specified. See also <a href="#">Client-based sessions</a> and <a href="#">CTS-based sessions</a>.</p>
Subject	<p>Entity that requests access to a resource</p> <p>When an identity successfully authenticates, AM associates the identity with the <a href="#">Principal</a> that distinguishes it from other identities. An identity can be associated with multiple principals.</p>
Identity store	Data storage service holding principals' profiles; underlying storage can be an LDAP directory service or a custom <a href="#">IdRepo</a> implementation.
Web Agent	Native library installed in a web server that acts as a policy enforcement point with policies based on web page URLs.