FORGEROCK®

OAuth 2.0 Guide
/ ForgeRock Access Management 6

Latest update: 6.0.0.7

Copyright © 2011-2018 ForgeRock AS.

Abstract

Guide showing you how to use ForgeRock® Access Management with OAuth 2.0.

Table of Contents

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. The chapter begins by showing where AM fits into the OAuth 2.0 authorization framework, and then shows how to configure the functionality.

RFC 6749, *The OAuth 2.0 Authorization Framework*, provides a standard way for *resource owners* to grant *client* applications access to the owners' web-based resources. The canonical example involves a user (resource owner) granting access to a printing service (client) to print photos that the user has stored on a photo-sharing server.

The section describes how AM supports the OAuth 2.0 authorization framework in terms of the roles that AM plays.[1] The following sequence diagram indicates the primary roles AM can play in the OAuth 2.0 protocol flow.

OAuth 2.0 Protocol Flow

OAuth 2.0 Protocol Flow

---
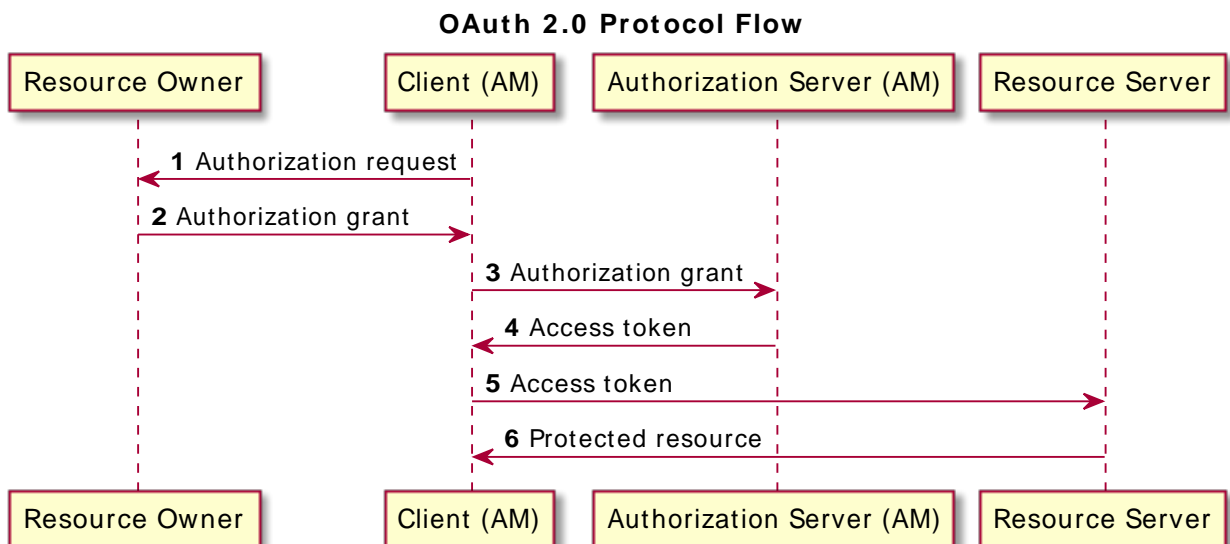
[1]Read RFC 6749 to understand the authorization framework itself.

1.1. OAuth 2.0 Authorization Server

AM can function as an OAuth 2.0 *authorization server*. In this role, AM authenticates resource owners and obtains their authorization in order to return access tokens to clients.

When using AM as authorization server, you can register clients in the AM console alongside agent profiles under the OAuth 2.0 Client tab. Clients can also register clients with AM dynamically. AM supports both confidential and public clients.

AM supports the four main grants for obtaining authorization described in RFC 6749: the authorization code grant, the implicit grant, the resource owner password credentials grant, and the client credentials grant.

Regardless of the grant type used, the request for an access token to the authorization server must contain a list of the required scopes. You can configure AM to grant scopes statically or dynamically:

- Statically. 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*.

See RFC 6749 for details on the authorization grant process, and for details on how clients should make authorization requests and handle authorization responses. AM also supports the *SAML v2.0 Bearer Assertion Profiles for OAuth 2.0*, described in the Internet-Draft.

1.1.1. OAuth 2.0 Authorization Grant

The authorization code grant starts with the client, such as a web-based service, redirecting the resource owner's user-agent to the AM authorization service. After authenticating the resource owner and obtaining the resource owner's authorization, AM redirects the resource owner's user-agent back to the client with an authorization code that the client uses to request the access token. The following sequence diagram outlines a successful process from initial client redirection through to the client accessing the protected resource.

⬥ FORGEROCK®

OAuth 2.0 Authorization Code Grant Process

OAuth 2.0 Authorization Code Grant
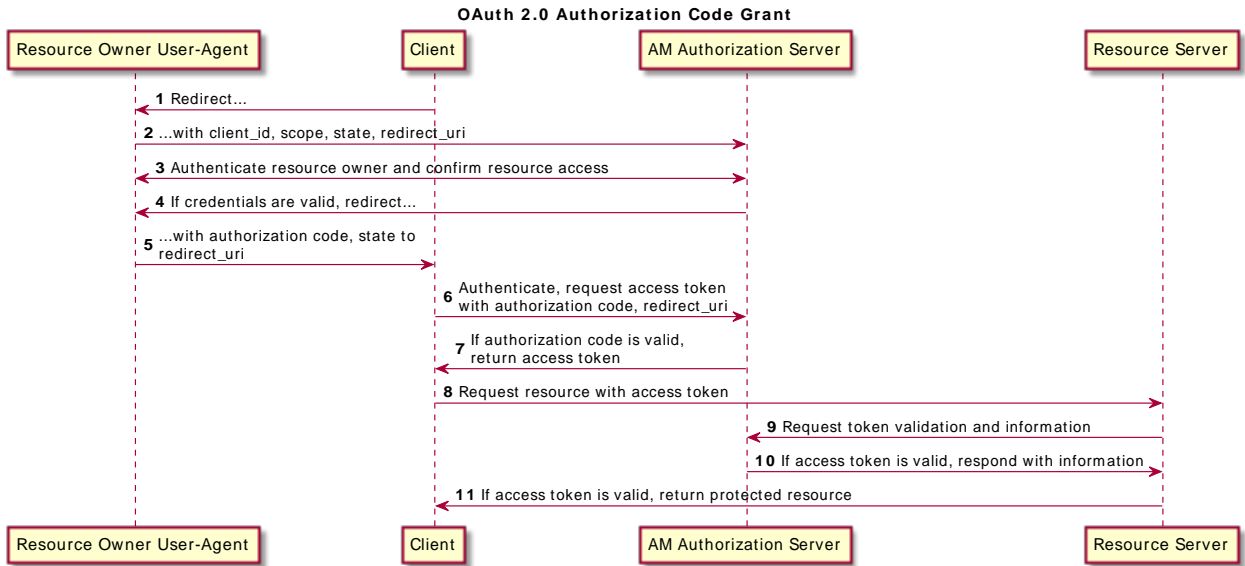
1.1.2. OAuth 2.0 Implicit Grant

The implicit grant is designed for clients implemented to run inside the resource-owner user agent. Instead of providing an authorization code that the client must use to retrieve an access token, AM returns the access token directly in the fragment portion of the redirect URI. The following sequence diagram outlines the successful process.

FORGEROCK

OAuth 2.0 Implicit Grant Process

OAuth 2.0 Implicit Grant

| Resource Owner User-Agent | Client | AM Authorization Server | Web-Hosted Client Resource | Resource Server |

1 Redirect...

2 ...with client_id, scope, state, redirect_uri

3 Authenticate resource owner and confirm resource access

4 Redirect with redirect_uri, access token in URI fragment...

5 ...to request redirect_uri without the fragment

6 Return web page with embedded script to extract access token

7 Execute script to retrieve access token

8 Return access token

9 Request resource with access token

10 Request access token validation and information

11 If access token is valid, respond with information

12 If access token is valid, return protected resource

1.1.3. OAuth 2.0 Resource Owner Password Credentials Grant

The Resource Owner Password Credentials grant type lets the client use the resource owner's user name and password to get an access token directly. Although this grant might seem to conflict with an original OAuth goal of not having to share resource owner credentials with the client, it can makes sense in a secure context where other authorization grant types are not available, such as a client that is part of a device operating system using the resource owner credentials once and thereafter using refresh tokens to continue accessing resources. The following sequence diagram shows the successful process.

OAuth 2.0 Resource Owner Password Credentials Grant Process

OAuth 2.0 Resource Owner Password Credentials Grant

Resource Owner	Client	AM Authorization Server	Resource Server

1 Provide user name, password

2 Authenticate

3 If credentials are valid, return access token

4 Request resource with access token

5 Request access token validation and information

6 If access token is valid, respond with information

7 If access token is valid, return protected resource

Resource Owner	Client	AM Authorization Server	Resource Server

1.1.4. OAuth 2.0 Client Credentials Grant

The client credentials grant uses client credentials as an authorization grant. This grant makes sense when the client is also the resource owner, for example. The following sequence diagram shows the successful process.

OAuth 2.0 Client Credentials Grant Process

OAuth 2.0 Client Credentials Grant

Client ResourceOwner	AM Authorization Server	Resource Server

1 Authenticate, request access token from token endpoint

2 If credentials are valid, return access token

3 Request resource with access token

4 Request access token validation and information

5 If access token is valid, respond with information

6 If access token is valid, return protected resource

Client ResourceOwner	AM Authorization Server	Resource Server

1.1.5. OAuth 2.0 Device Flow

The OAuth 2.0 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 on a PC or smartphone, and entering a code displayed on the client device.

The sequence diagram below demonstrates the OAuth 2.0 Device Flow:

OAuth 2.0 Device Flow
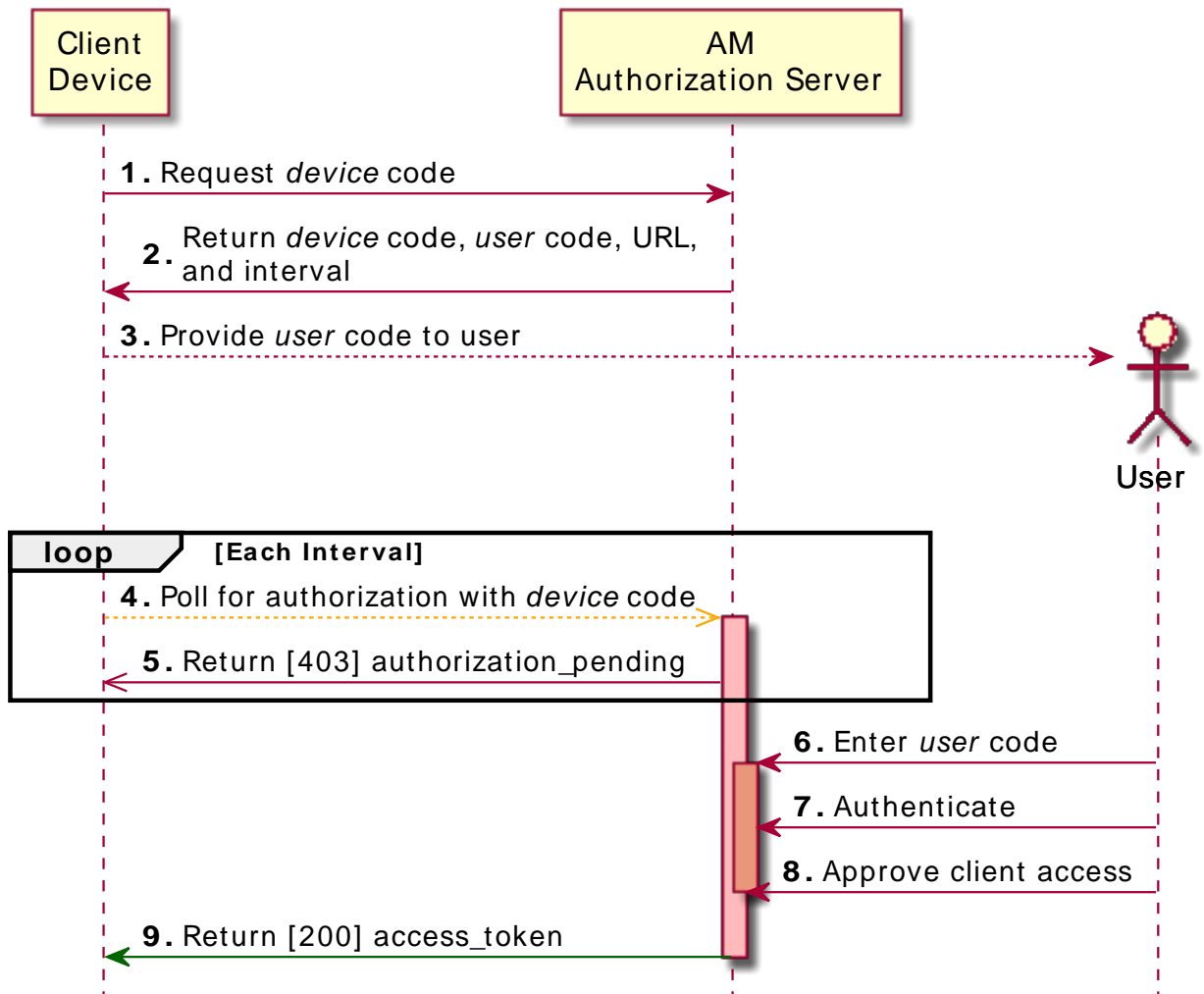
The steps in the diagram are described below:

FORGEROCK

1. The client device requests a device code from AM by using a REST call.

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 will ask the user to authenticate.

8. Upon authentication the user can authorize the client device. The AM consent page also displays the requested scopes, and their values:

OAuth 2.0 Consent Page

9. 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.

For more information, see "OAuth 2.0 Device Flow Endpoints".

1.1.6. 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`.

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.
>
> Should 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.
>
> Should 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.

`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`.

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.

Both AM and the remote consent service make the required public keys available from a `jwk_uri` URI, enabling the signing and encryption between the two servers.

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

For information on configuring a remote consent service, see"Configuring Remote Consent Services".

1.1.7. JWT Bearer Profile

The Internet-Draft, *JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants* describes a means to use a JWT for client authentication or to use a JWT to request an access token. When clients are also resource owners, the profile allows clients to issue JWTs to obtain access tokens rather than use the resource owner password credentials grant.

AM implements both features of the profile. Both involve HTTP POST requests to the access token endpoint.

When the client bearing the JWT uses it for authentication, then in the POST data the client sets `client_assertion_type` to `urn:ietf:params:oauth:client-assertion-type:jwt-bearer` and `client_assertion` to the JWT string.

JWT Bearer Client Authentication

JWT Bearer Client Authentication
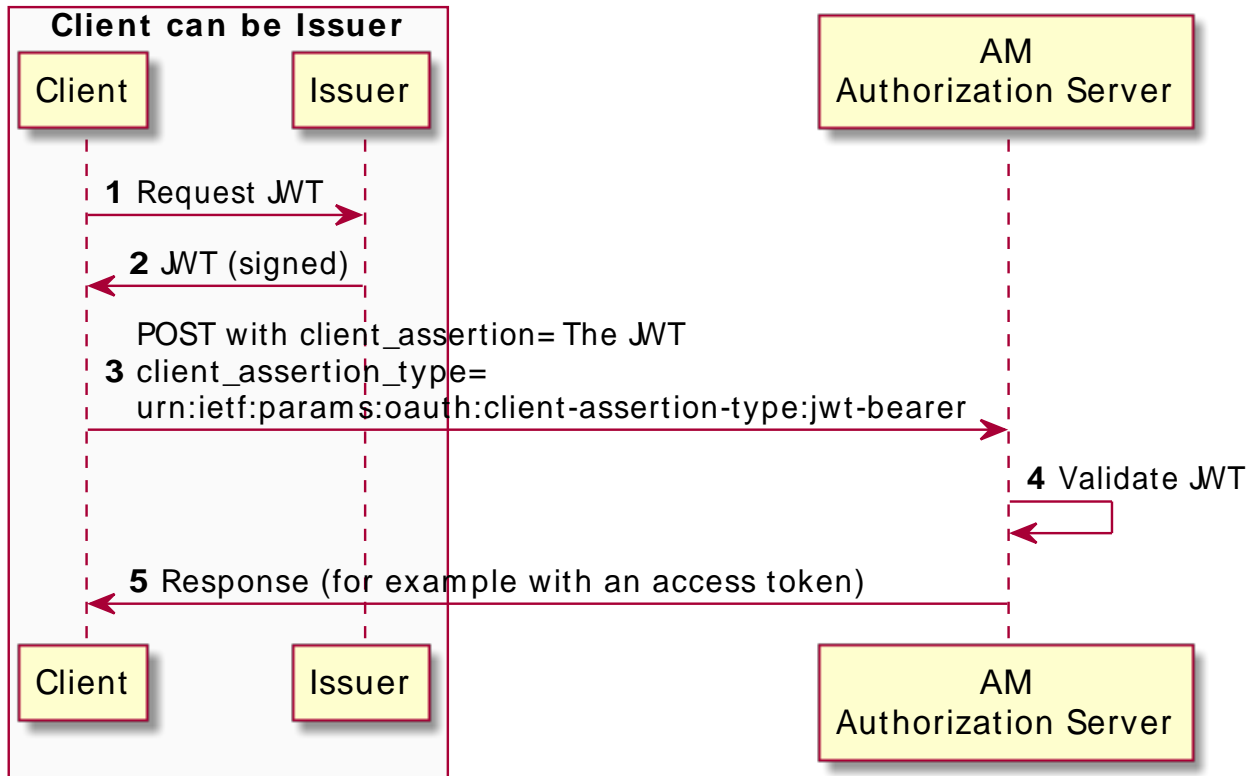
The HTTP POST to AM looks something like the following, where the assertion value is the JWT:

```
POST /openam/oauth2/realms/root/access_token HTTP/1.1
Host: openam.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&
code=362ad374-735c-4f69-aa8e-bf384f8602de&
client_assertion_type=urn%3Aietf%3Aparams%3Aoauth%3A
 client-assertion-type%3Ajwt-bearer&
client_assertion=eyAiYWxnIjogIlJTMjU2IiB9.eyAic3ViIjogImp3...
```

In the above profile, AM must be able to validate the JWT with the following claims:

• "iss" (issuer) whose value identifies the JWT issuer.

- "sub" (subject) whose value identifies the principal who is the subject of the JWT.

  For client authentication, the "sub" value must be the same as the value of the "client_id".

- "aud" (audience) whose value identifies the authorization server that is the intended audience of the JWT.

  When the JWT is used for authentication, this is the AM access token endpoint.

- "exp" (expiration) whose value specifies the time of expiration.

> **Important**
>
> Providing a JWT with an expiry time greater than 30 minutes causes AM to return a `JWT expiration time is unreasonable` error message.

Also for validation, the issuer must digitally sign the JWT or apply a keyed message digest. When the issuer is also the client, the client can sign the JWT by using a private key, and include the public key in its profile registered with AM.

A sample Java-based client 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*.

1.1.8. SAML v2.0 Bearer Assertion Profiles

The Internet-Draft, *SAML v2.0 Bearer Assertion Profiles for OAuth 2.0*, describes a means to use SAML v2.0 assertions to request access tokens and to authenticate OAuth 2.0 clients.

At present AM implements the profile to request access tokens.

In both profiles, the issuer must sign the assertion. The client communicates the assertion over a channel protected with transport layer security by performing an HTTP POST to the AM's access token endpoint. AM as OAuth 2.0 authorization server uses the issuer ID to validate the signature on the assertion.

In the profile to request an access token, the OAuth 2.0 client bears a SAML v2.0 assertion that was issued to the resource owner on successful authentication. A valid assertion in this case is equivalent to an authorization grant by the resource owner to the client. OAuth 2.0 clients must make it clear to the resource owner that by authenticating to the identity provider who issues the assertion, they are granting the client permission to access the protected resources.

SAML v2.0 Bearer Assertion Authorization Grant

SAML v2.0 Bearer Assertion Authorization Grant
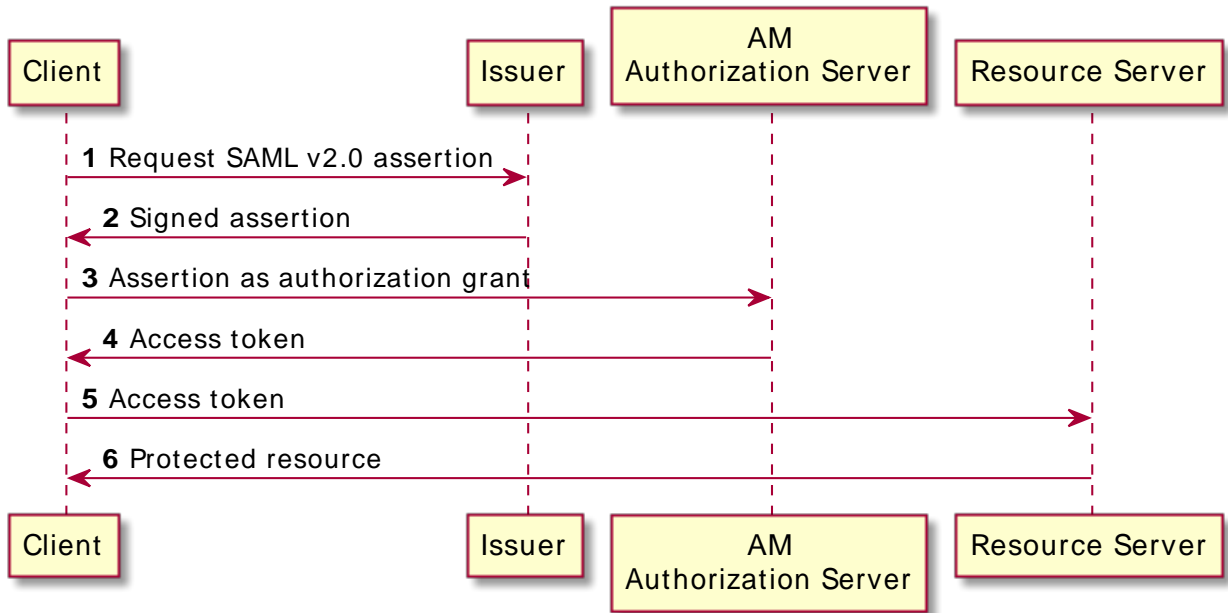
The HTTP POST to AM to request an access token looks something like this:

```
POST /openam/oauth2/realms/root/access_token HTTP/1.1
    Host: openam.example.com
    Content-Type: application/x-www-form-urlencoded

    grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Asaml2-bearer&
    assertion=PHNhbWx0l...[base64url encoded assertion]...ZT4&
    client_id=[ID registered with OpenAM]
```

1.2. OAuth 2.0 Client and Resource Server Solution

AM can function as an OAuth 2.0 client for installations where the web 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

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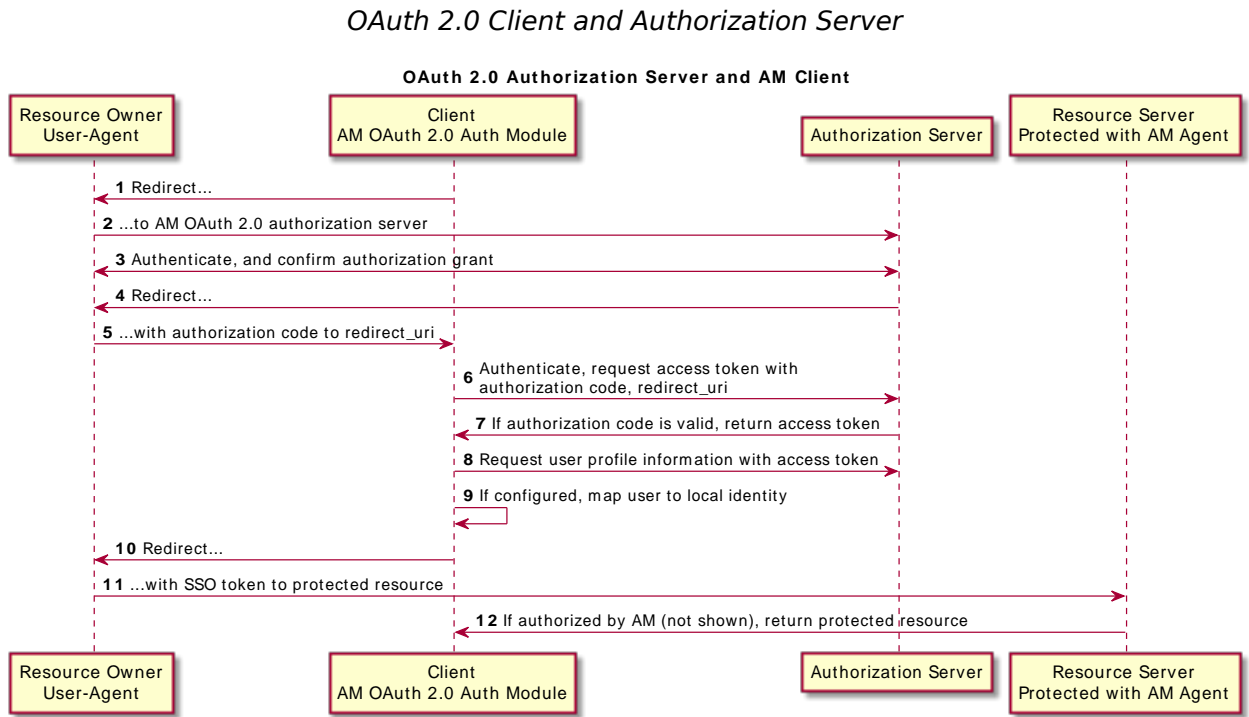
As the OAuth 2.0 client functionality is implemented as an AM authentication module, you do not need to deploy your own resource server implementation when using AM as an OAuth 2.0 client. Instead, use web or Java agents or IG to protect resources.

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

1.3. 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/tokeninfo` endpoint to determine whether the access token is still valid, and to retrieve the scopes associated with the access token.

The default AM implementation of OAuth 2.0 scopes assumes that the space-separated (%20 when URL-encoded) list of scopes in an access token request correspond to names of attributes in the resource owner's profile.

To take a concrete example, consider an access token request where `scope=mail%20cn` and where the resource owner is the default AM demo user. (The demo user has no email address by default, but you can add one, such as `demo@example.com` to the demo user's profile.) When the resource server performs an HTTP GET on the token information endpoint, `/oauth2/tokeninfo?access_token=token-id`, AM populates the `mail` and `cn` scopes with the email address (`demo@example.com`) and common name (`demo`) from the demo user's profile. The result is something like the following token information response:

```
{
    "mail": "demo@example.com",
    "scope": [
        "mail",
        "cn"
    ],
    "cn": "demo",
    "realm": "/",
    "token_type": "Bearer",
    "expires_in": 577,
    "client_id": "MyClientID",
    "access_token": "f9063e26-3a29-41ec-86de-1d0d68aa85e9"
}
```

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.

1.4. 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.

1.5. OAuth 2.0 JSON Web Token Proof-of-Possession

AM supports associating a confirmation key with an access token to support proof-of-possession interactions as per the Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs) internet-draft. This allows the presenter of a bearer token to prove that it was originally issued the access token.

AM supports confirmation keys for both stateful and stateless OAuth 2.0 tokens. For more information about stateless OAuth 2.0 tokens, see "Stateless OAuth 2.0 Access and Refresh Tokens".

To implement proof-of-possession for tokens, the client should include a JSON web key (JWK) when making a request to an authorization server, such as AM, for an OAuth 2.0 access token. The JWK consists of the public key of a keypair generated by the client.

When the issued access token is presented 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.

FORGEROCK

OAuth 2.0 Proof-of-Possession
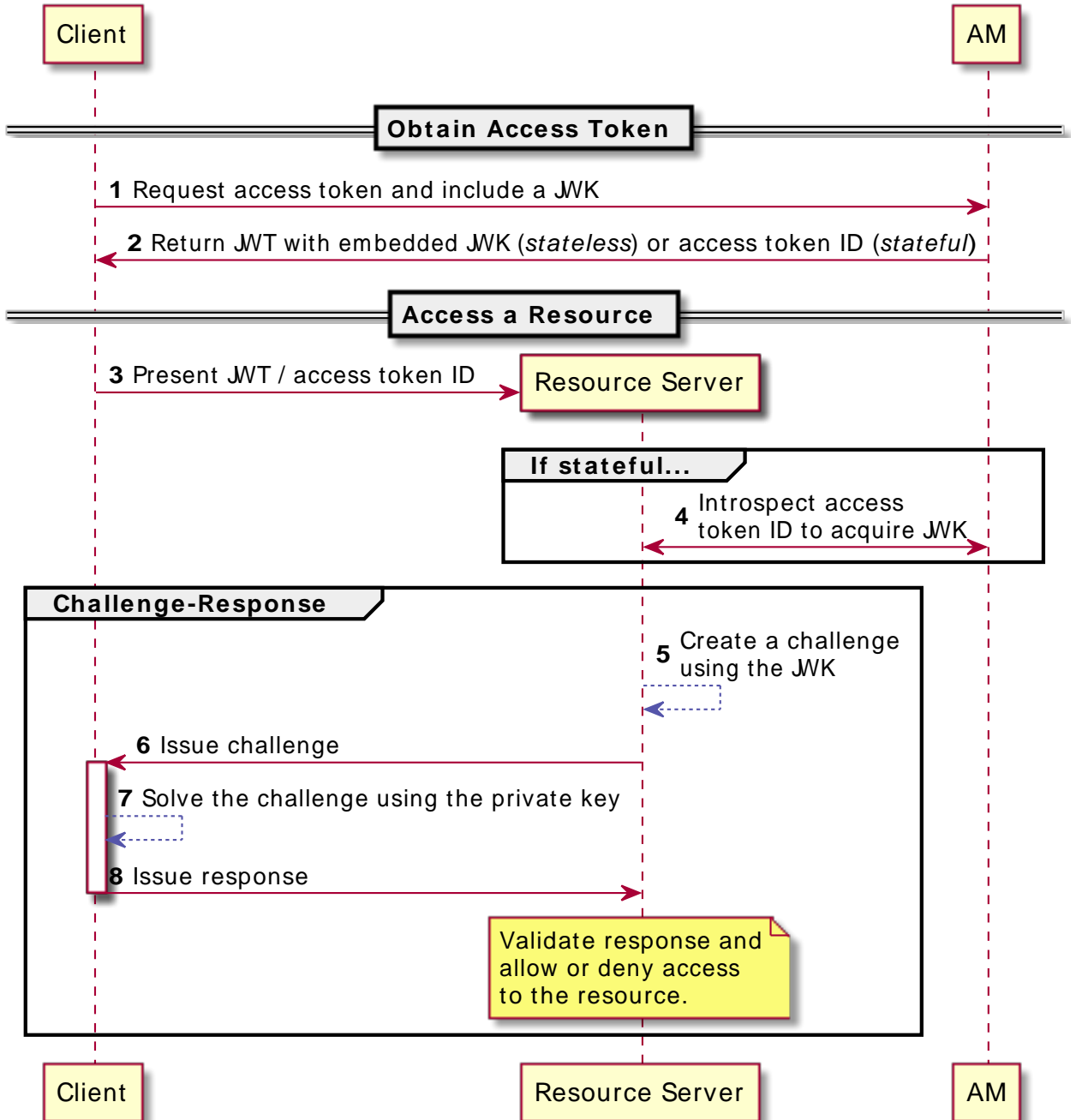
For information on adding proof-of-possession keys to access token requests, see "Using OAuth 2.0 JSON Web Token Proof-of-Possession".

Chapter 2
Implementing OAuth 2.0

This chapter covers implementing and configuring AM support for OAuth 2.0.

2.1. Configuring the OAuth 2.0 Authorization Service

You configure the OAuth 2.0 authorization service for a particular realm from the Realms > *Realm Name* > Dashboard page of the AM console.

To Set Up the OAuth 2.0 Authorization Service

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

You must set up a standard policy in the Top Level Realm (/) to protect the authorization endpoint. The policy must appear in a policy set of type `iPlanetAMWebAgentService`, which is the default in the AM policy editor. When configuring the policy, allow all authenticated users to perform HTTP GET and POST requests on the authorization endpoint. The authorization endpoint is described in "OAuth 2.0 Client and Resource Server Endpoints". For details on creating policies, see "*Implementing Authorization*" in the *Authorization Guide*.

In this configuration, AM serves the resources to protect, and no separate application is involved. AM therefore acts both as the policy decision point and also as the policy enforcement point that protects the OAuth 2.0 authorization endpoint.

1. In the AM console, select Realms > *Realm Name* > Dashboard > Configure OAuth Provider > Configure OAuth 2.0.

2. On the Configure OAuth 2.0 page, select the Realm for the authorization 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 unless you do not want the authorization service to supply a refresh token when returning an access token.

5. (Optional) Select Issue Refresh Tokens on Refreshing Access Tokens if you want the authorization service to supply a refresh token when refreshing an access token.

6. (Optional) If you want to use the default scope implementation, whereby scopes are taken to be resource owner profile attribute names, then keep the default setting.

If you have a custom scope validator implementation, put it on the AM classpath, and provide the class name as Scope Implementation Class. For an example, see "Customizing OAuth 2.0 Scope Handling".

7. Click Create to complete the process.

   To access the authorization server configuration in the AM console, browse to Realms > *Realm Name* > Services, and then click OAuth2 Provider.

8. (Optional) If you want to configure the OAuth 2.0 service to interact with AM's Authorization service to dynamically providing 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*.

9. (Optional) If your provider has a custom response type plugin, put it on the AM classpath, and then add the custom response types and the plugin class names to the list of Response Type Plugins.

10. (Optional) 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, then complete this step.

    The following steps describe how to configure AM authentication so OAuth 2.0 resource owners can log in using their email address, stored on the LDAP profile attribute, `mail`. Adapt the names if you use a different LDAP profile attribute, such as `cn`:

    a. When configuring the data store for the LDAP identity repository, make sure that you select Load Schema before saving, and that you set the Authentication Naming Attribute to `mail`. You can find the data store configuration under Realms > *Realm Name* > Data Stores.

    b. Add the `mail` profile attribute name to the list of attributes that can be used for authentication.

       To make the change, navigate to Realms > *Realm Name* > Services, click OAuth2 Provider, add the profile attributes to the list titled User Profile Attribute(s) the Resource Owner is Authenticated On, and then click Save Changes.

    c. Create an LDAP authentication module to use with the external directory:

       i. In the AM console under Realms > *Realm Name* > Authentication > Modules, create a module to access the LDAP identity repository, such as `LDAPAuthUsingMail`.

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

       iii. In the Attributes Used to Search for a User to be Authenticated list, remove the default `uid` attribute and add the `mail` attribute.

       iv. Click Save.

    d.   Create an authentication chain to include the module, such as `authUsingMail`.

        i.   When creating the authentication chain, choose the `LDAPAuthUsingMail` module from the Instance drop-down list, and set the criteria to REQUIRED.

        ii.   Click Save.

    e.   Set Organization Authentication Configuration to use the new chain, `authUsingMail`, and then click Save.

       At this point OAuth 2.0 resource owners can authenticate using their email address rather than their user ID.

11. Add a multi-valued string syntax profile attribute to your identity repository. AM stores resource owners' consent to authorize client access in this profile attribute. On subsequent requests from the same client for the same scopes, the resource owner no longer sees the authorization page.

    You are not likely to find a standard profile attribute for this. For evaluation purposes only, you might try an unused existing profile attribute, such as `description`.

    When moving to production, however, use a dedicated, multi-valued, string syntax profile attribute that clearly is not used for other purposes. For example, you might call the attribute `oAuth2SavedConsent`.

    Adding a profile attribute involves updating the identity repository to support use of the attribute, updating the AMUser Service for the attribute, and optionally allowing users to edit the attribute. The process is described in "Adding User Profile Attributes" in the *Setup and Maintenance Guide*, which demonstrates adding a custom attribute when using DS to store user profiles.

12. Navigate to Realms > *Realm Name* > Services, click OAuth2 Provider, and then specify the name of the attribute created in the previous step in the Saved Consent Attribute Name field.

13. Click Save Changes.

To further adjust the authorization server configuration after you create it, in the AM console navigate to Realms > *Realm Name* > Services, and then click OAuth2 Provider.

To adjust global defaults, in the AM console navigate to Configure > Global Services, and then click OAuth2 Provider.

2.2. Registering OAuth 2.0 Clients With the Authorization 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

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

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

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

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

4. When finished, save your work.

To Configure AM for OAuth 2.0 Dynamic Client Registration

AM supports dynamic registration as defined by RFC 7591, the *OAuth 2.0 Dynamic Client Registration Protocol*. The protocol describes 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.

An example is shown in "Open OAuth 2.0 Dynamic Client Registration Example".

• 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.

An example is shown in "OAuth 2.0 Dynamic Client Registration Example With Access Token".

• 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.

An example is shown in "OAuth 2.0 Dynamic Client Registration Example With Software Statement".

Follow these steps to configure AM for dynamic client registration:

FORGEROCK

1. Configure an authorization service.

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

   For additional details, see "Client Dynamic Registration".

3. (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.

Open OAuth 2.0 Dynamic Client Registration Example

The following example shows dynamic registration with the Allow Open Dynamic Client Registration option enabled.

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/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/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"]
```

```
        }
```

OAuth 2.0 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=amadmin&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-Japan-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/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"]
 }
```

OAuth 2.0 Dynamic Client Registration Example With Software Statement

The following example extends "OAuth 2.0 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.
eyJpc3MiOiJodHRwczovL2NsaWVudC5leGFtcGxlLmNvbSIsImlhdCI6MTUwNjY3MTg1MSwiZX
hwIjoxNTM4MjA3ODUxLCJhdWQiOiJvcGVuYW0uZXhhbXBsZS5jb20iLCJzdWIiOiI0TlJCMS0w
WFpBQlpJOUU2LTVTTTNSIiwicmVkaXJlY3RfdXJpcyI6WyJodHRwczovL2NsaWVudC5leGFtcG
xlLmNvbS9jYWxsYmFjayJdfQ.
IOxZaWTOzSPkEkrXC9nj8RDrpulzzMuZ-4R7_Ol_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"
    ]
}
```

To build your own JWTs for testing and evaluation, use an online service such as https://jwt.io/.

Prior to registration, obtain an access token:

```
$ curl \
 --request POST \
 --user "masterClient:password" \
 --data "grant_type=password&username=amadmin&password=password&scope=dynamic_client_registration" \
 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/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": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9
.eyJpc3MiOiJodHRwczovL2NsaWVudC5leGFtcGxlLmNvbSIsImlhdCI6MTUwNjY3MTg1MSwiZXhwIjoxNTM4MjA3ODUxLCJhdWQiOiJvcGVuYW0uzX
.IOxZaWTOzSPkEkrXC9nj8RDrpulzzMuZ-4R7_Ol_jhw"
 }' \
 https://openam.example.com:8443/openam/oauth2/register
```

2.3. Configuring 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/`
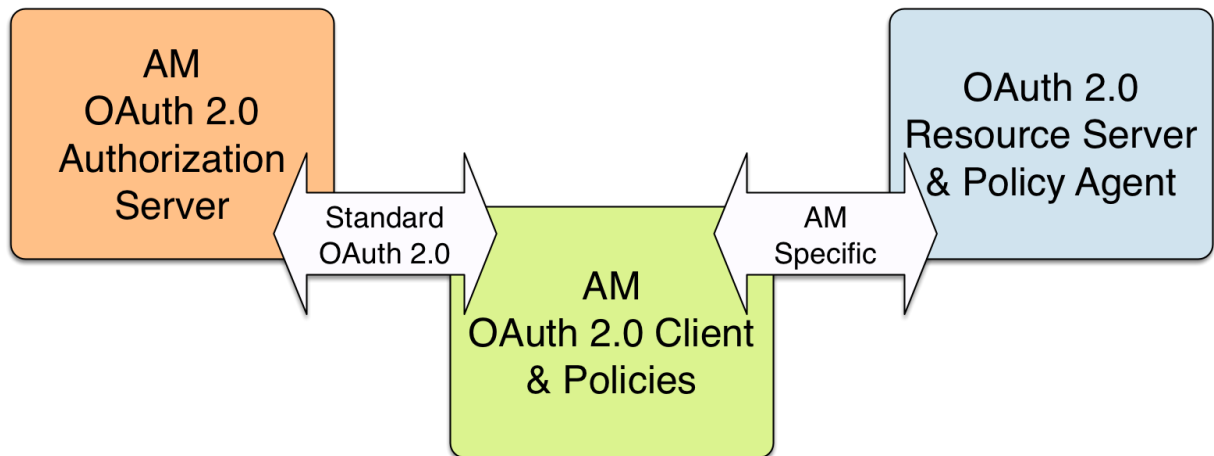
The example in this section uses three servers, `http://authz.example.com:8080/openam` as the OAuth 2.0 authorization server, `http://client.example.com:8080/openam` as the OAuth 2.0 client, which also handles policy, `http://www.example.com:8080/` as the OAuth 2.0 resource server protected with an AM 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 Authorization 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 Authorization Service".

5. Log out and access the protected resources to see the process in action.

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

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

OAuth2 Access Token Profile Service Parameter name

Identifies the parameter that contains the access token value, which in this example is `access_token`.

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.oauth2.DefaultAccountMapper`.

Account Mapper Configuration

In this example, `cn=cn`.

Attribute Mapper

In this example, `org.forgerock.openam.authentication.modules.oauth2.DefaultAttributeMapper`.

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 Authorization 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`.

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=/#login&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

2.4. Managing OAuth 2.0 Consent

This section covers configuring an OAuth 2.0 remote consent service, as well as how OAuth 2.0 clients can manage their own consent.

AM also exposes a RESTful API that lets administrators read, list, and delete OAuth 2.0 tokens. For details, see "OAuth 2.0 Token Administration Endpoint (Legacy)".

2.4.1. 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 `jwk_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 `jwk_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`, a Remote Consent Service secret, and then select Create.

4. 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".

5. Save your changes.

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

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

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

1. In the AM console, 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 Authorization 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. On the Consent tab, select Enable Remote Consent, and from the Remote Consent Service ID drop-down list, select the name of the Remote Consent Agent, for example `myRCSAgent`.

5. (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

6. 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.

> **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 `http://openam.example.com:8080/openam`. For information on configuring this instance, see "To Configure AM to use a Remote Consent Service" and "To Configure the OAuth 2.0 Provider to Use a Remote Consent Agent Profile".

- One instance that acts as the example remote consent service, for example `https://rcs.exampe.com:8443/openam`. To configure this instance, perform the following steps:

1. In the AM console, select Realms, and then select the realm that you are working with.

2. Navigate to 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 `test`. 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 `selfserviceenctest`. 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 jwk_uri, enter the URI where the remote consent service can obtain the keys that vam uses to sign and encrypt the consent request.

For this example, enter `http://openam.example.com:8080/openam/oauth2/connect/jwk_uri`.

5. Select Create.

6. Verify the configuration. For more information about the available properties, see "Remote Consent Service".

7. Save your changes.

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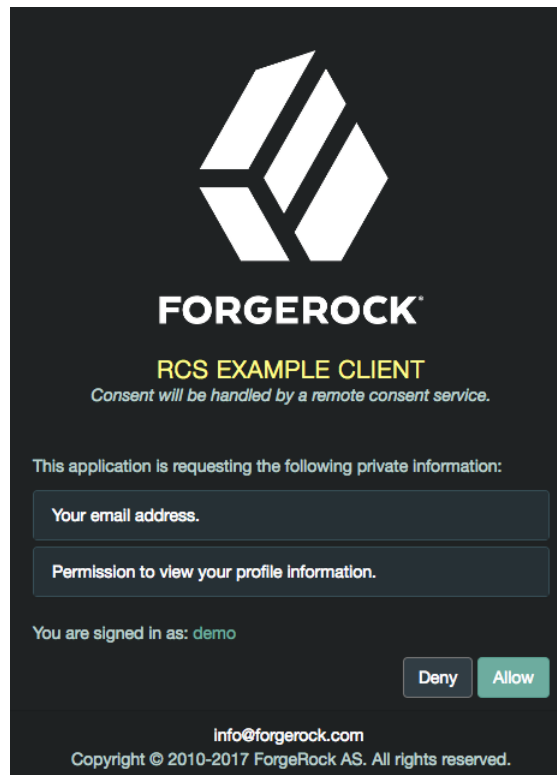
Note that the *fr-dark-theme* has been applied to AM instance acting as the the remote consent service for the purpose of this demonstration.

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

2.4.2. Allowing Clients To Skip Consent

Companies that have internal applications that use OAuth 2.0 or OpenID Connect 1.0 can allows clients to skip consent and make consent confirmation optional so as not to disrupt their online experience.

To Allow a Client To Skip Consent
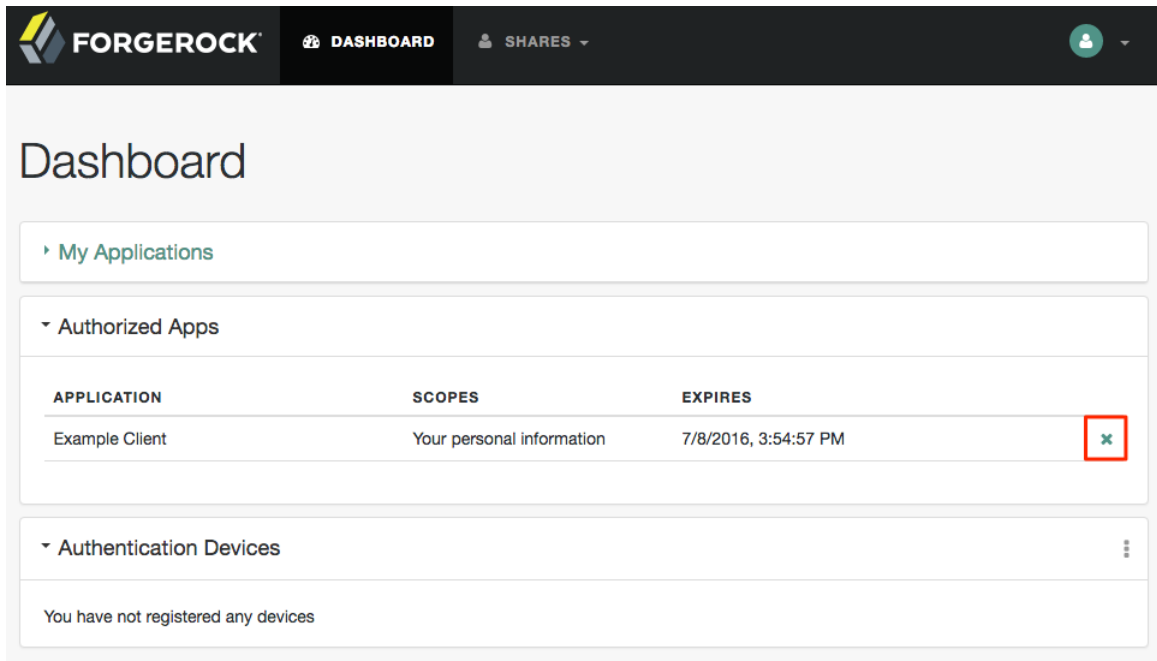
1. In the AM console, select Realms, and then select the realm that you are working with.

2. First, create or update your OAuth2 provider:

    a. Select Dashboard > Configure OAuth Provider > Configure OpenID Connect, and then click Create.

    b. Click Services > OAuth2 Provider.

    c. On the OAuth2 Provider tabs, click the drop-down arrow, and select Consent.

    d. Enable Allow Clients to Skip Consent.

    e. Click Save Changes.

3. Next, create or update an OpenID Connect client:

    a. Navigate to Realms > *Realm Name* > Applications > OAuth 2.0.

    b. Click Add Client. Enter a `Client ID`, `Redirection URIs`, `Scope(s)`, and `Default Scope(s)` as needed, and then click Create.

    c. On the newly created client page, click the Advanced tab.

    d. Enable Implied consent.

    e. Click Save Changes.

        When both settings are set on the OAuth2 provider and OAuth 2.0 Client settings, AM will treat the requests as if the client has already saved its consent and will suppress any user consent pages to the client.

2.4.3. User Consent Management

Users of OAuth 2.0 clients can now 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 their OAuth 2.0 tokens or remove them by clicking the Revoke Access button, effectively removing their consent to the application.

OAuth 2.0 Self-Service

2.5. Stateless OAuth 2.0 Access and Refresh Tokens

AM supports *stateless* access and refresh tokens for OAuth 2.0. Stateless access and refresh tokens allow clients to directly validate the tokens without storing session information in server memory.

The stateless OAuth 2.0 access token is a JWT, which allows any AM instance in the issuing cluster to validate an OAuth 2.0 token without the need for cross-server communication.

To Configure Stateless OAuth 2.0 Access and Refresh Tokens

1.   Open the AM console.

2.   Under Realms, select the realm that you are working with.

3.   Click Services, and then select OAuth2 Provider.

4.   For Use Stateless Access & Refresh Tokens, slide the toggle button to the right to enable the feature.

5.   Optional. For Issue Refresh Tokens, slide the toggle button to the right to enable the feature.

6. For Issue Refresh Tokens on Refreshing Access Tokens, slide the toggle button to the right to enable the feature.

2.6. Configuring Stateless OAuth 2.0 Token Blacklisting

AM provides a blacklisting feature that prevents stateless OAuth v2.0 tokens from being reused if the authorization code has been replayed or tokens have been revoked by either the client or resource owner.

To Configure Stateless OAuth 2.0 Token Blacklisting

1. On the AM console, navigate to Configure > Global Services > Global > OAuth2 Provider.

2. Under Global Attributes, enter the number of blacklisted tokens in the Token Blacklisting Cache Size field.

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

   Default: 10000

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

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

   Default: 60 seconds

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

   When stateless blacklisting is enabled, AM tracks OAuth v2.0 access and refresh 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, the AM tracks the access token for 101 minutes. You can increase the blacklist purge delay if you expect system clock skews in an AM server cluster to be greater than one minute. There is no need to increase the blacklist purge delay for servers running a clock synchronization protocol, such as Network Time Protocol.

   Default: 1 minute

5. Click Save to apply your changes.

2.7. Configuring Stateless OAuth 2.0 Token Encryption

AM supports encryption of stateless OAuth 2.0 access and refresh tokens, to protect the data stored within.

The encryption method AM uses for stateless OAuth 2.0 access and refresh tokens uses the HMAC message authentication code, as per RFC 2104, with the SHA-256 hash function to provide message authentication. Due to this authentication, enabling encryption disables signing as it is not neccessary to do both.

To Enable Stateless OAuth 2.0 Token Encryption

1. In the AM console navigate to Realms > *Realm Name* > Services, and then click 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 Authorization Service".

2. On the Core tab, enable the Use Stateless Access & Refresh Tokens property.

3. On the Advanced tab:

    a. Enable the Enable Stateless Token Encryption property

    b. In the Token Encryption Secret Key Alias property, enter the alias of the encrpytion key alias to use to encrypt the OAuth 2.0 tokens.

        The alias is retrieved from the keystore referenced by the `com.sun.identity.saml.xmlsig.keystore` property.

4. Save your changes.

    Stateless OAuth 2.0 access and refresh tokens will now be encrypted.

2.8. Configuring 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).

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

- RS256 - RSA using SHA-256
- 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

If you intend to use an ECDSA signing algorithm, you must generate a public/private key pair for use with ECDSA. To generate the public and private key pair, see step 1 in "Configuring Elliptic Curve Digital Signature Algorithms" in the *Authentication and Single Sign-On Guide*.

To Configure Digital Signatures

1. In the AM console navigate to Realms > *Realm Name* > Services, and then click 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 Authorization Service".

2. On the Advanced tab, in the OAuth2 Token Signing Algorithm drop-down list, select the signing algorithm to use for your digital signatures.

3. Take one of the following actions depending on the token signing algorithm:

   a. If you are using an HMAC signing algorithm, enter the Base64-encoded key used by HS256, HS384 and HS512 in the Token Signing HMAC Shared Secret field.

   b. If you are using RS256, enter the public/private key pair used by RS256 in the Token Signing RSA public/private key pair field. The public/private key pair will be retrieved from the keystore referenced by the property `com.sun.identity.saml.xmlsig.keystore`.

   c. If you are using an ECDSA signing algorithm, enter the list of public/private key pairs used for the elliptic curve algorithms (ES256/ES384/ES512) in the Token Signing ECDSA public/private key pair alias field. For example, `ES256|es256test`. Each of the public/private key pairs will be retrieved from the keystore referenced by the property `com.sun.identity.saml.xmlsig.keystore`.

   d. Click Save Changes.

4. Next, update the OpenID Connect client:

   a. Under Agent, click New, enter a Name and Password for the agent, and then click Create.

   b. In the ID Token Signing Algorithm field, enter the signing algorithm that the ID token for this client must be signed with. Default: `RS256`.

      - HS256 (HMAC with SHA-256)
      - HS384 (HMAC with SHA-384)
      - HS512 (HMAC with SHA-512)

- RS256 (RSA using SHA-256)
- 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)

c. Click Save.

To Obtain the OAuth 2.0/OpenID Connect 1.0 Public Signing Key

AM exposes the public keys used to digitally sign OAuth 2.0 and OpenID Connect 1.0 access and refresh tokens at a JSON web key (JWK) URI endpoint, which is exposed from all realms for an OAuth2 provider. The following steps show how to access the public keys:

1. To find the JWK URI, perform an HTTP GET at `/oauth2/realms/root/.well-known/openid-configuration`:

```
$ curl http://openam.example.com:8080/openam/oauth2/realms/root/.well-known/openid-configuration
{
    "id_token_encryption_alg_values_supported":[
        "RSA1_5"
    ],
    "response_types_supported":[
        "token id_token",
        "code token",
        "code token id_token",
        "token",
        "code id_token",
        "code",
        "id_token"
    ],
    "registration_endpoint":"http://openam.example.com:8080/openam/oauth2/realms/root/connect/register"
,
    "token_endpoint":"http://openam.example.com:8080/openam/oauth2/realms/root/access_token",
    "end_session_endpoint":"http://openam.example.com:8080/openam/oauth2/realms/root/connect/
endSession",
    "scopes_supported":[
        "phone",
        "address",
        "email",
        "openid",
        "profile"
    ],
    "acr_values_supported":[

    ],
    "version":"3.0",
    "userinfo_endpoint":"http://openam.example.com:8080/openam/oauth2/realms/root/userinfo",
    "token_endpoint_auth_methods_supported":[
        "client_secret_post",
        "private_key_jwt",
        "client_secret_basic"
    ],
    "subject_types_supported":[
        "public"
    ],
    "issuer":"http://openam.example.com:8080/openam/oauth2/realms/root",
    "id_token_encryption_enc_values_supported":[
```

```
            "A256CBC-HS512",
            "A128CBC-HS256"
        ],
    "claims_parameter_supported":true,
    "jwks_uri":"http://openam.example.com:8080/openam/oauth2/realms/root/connect/jwk_uri",
    "id_token_signing_alg_values_supported":[
            "ES384",
            "ES256",
            "ES512",
            "HS256",
            "HS512",
            "RS256",
            "HS384"
        ],
    "check_session_iframe":"http://openam.example.com:8080/openam/oauth2/realms/root/connect/
checkSession",
    "claims_supported":[
            "zoneinfo",
            "phone_number",
            "address",
            "email",
            "locale",
            "name",
            "family_name",
            "given_name",
            "profile"
        ],
    "authorization_endpoint":"http://openam.example.com:8080/openam/oauth2/realms/root/authorize"
}
```

2. Perform an HTTP GET at the JWKS URI to get the public signing key:

```
$ curl http://openam.example.com:8080/openam/oauth2/realms/root/connect/jwk_uri
{
    "keys":
    [
        {
            "kty":"RSA",
            "kid":"SylLC6Njt1KGQktD9Mt+0zceQSU=",
            "use":"sig",
            "alg":"RS256",
            "n":"AK0kHP1O-RgdgLSoWxkuaYoi5Jic6hLKeuKw8WzCfsQ68ntBDf6tVOTn_kZA7Gjf4oJ
AL1dXLlxIEy-kZWnxT3FF-0MQ4WQYbGBfaW8LTM4uAOLLvYZ8SIVEXmxhJsSlvaiTWCbNFaOf
iII8bhFp4551YB07NfpquUGEwOxOmci_",
            "e":"AQAB"
        }
    ]
}
```

Chapter 3
Using OAuth 2.0

This chapter covers examples and usage of AM with OAuth 2.0.

AM exposes the following REST endpoints for different OAuth 2.0 purposes:

- Endpoints for OAuth 2.0 clients and resource servers, mostly defined in RFC 6749, *The OAuth 2.0 Authorization Framework*, with additional `tokeninfo` and `introspect` endpoints useful to resource servers and clients.

- An endpoint for reading OAuth 2.0 resource sets. This is specific to AM.

- An endpoint for OAuth 2.0 token administration. This is specific to AM.

- An endpoint for OAuth 2.0 client administration. This is specific to AM.

When accessing the APIs, browser-based REST clients can rely on AM to handle the session as usual. First authenticate with AM. Then perform the operations in the browser session.

Clients not running in a browser can authenticate as described in "Authentication and Logout", whereby AM returns a `tokenId` value. Clients pass the `tokenId` value in a header named after the authentication cookie, by default `iplanetDirectoryPro`.

3.1. OAuth 2.0 Client and Resource Server Endpoints

AM exposes REST endpoints for making calls to AM acting as an authorization server.

In addition to the standard authorization and token endpoints described in RFC 6749, AM also exposes a token information endpoint for resource servers to get information about access tokens so they can determine how to respond to requests for protected resources, and an introspection endpoint to retrieve metadata about a token, such as approved scopes and the context in which the token was issued.

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`.

When acting as an OAuth 2.0 authorization server, AM exposes the following endpoints for clients and resource servers:

/oauth2/authorize

Authorization endpoint defined in RFC 6749, used to obtain consent and an authorization grant from the resource owner.
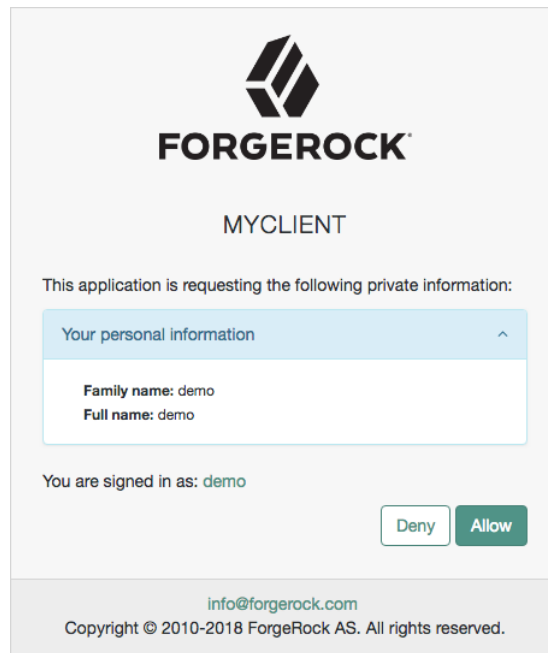
The `/oauth2/authorize` endpoint is protected by the policy you created after OAuth 2.0 authorization server configuration, which grants all authenticated users access.

The following is an example URL for obtaining consent:

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

After logging in, the URL above presents the OAuth 2.0 consent screen, similar to the following:

OAuth 2.0 Consent Screen

You must specify the realm if the AM OAuth 2.0 provider is configured for a subrealm rather than the top-level realm. For example, if the OAuth 2.0 provider is configured for the `/customers` realm, then use `/oauth2/realms/root/realms/customers/authorize`.

If creating your own consent page or when using REST calls, create a POST request to the endpoint with the following additional mandatory parameters:

decision

> Whether the resource owner consents to the requested access, or denies consent.
>
> Valid values are `allow` or `deny`.

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.
>
> To save consent, set the `save_consent` property to `on`.
>
> You must provide 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".

csrf

> Duplicates the contents of the `iPlanetDirectoryPro` cookie, which contains the SSO token of the resource owner giving consent.
>
> Duplicating the cookie value helps prevent against Cross-Site Request Forgery (CSRF) attacks.

Example:

```
$ curl \
--request POST
 \
--header  "Content-Type: application/x-www-form-urlencoded"
 \
--Cookie "iPlanetDirectoryPro=AQIC5w...*"
 \
--data "redirect_uri=http://www.example.net"
 \
--data "scope=profile"
 \
--data "response_type=code"
 \
--data "client_id=myClient"
 \
--data "csrf=AQIC5w...*"
 \
--data "decision=allow"
 \
--data "save_consent=on" \
"https://openam.example.com:8443/openam/oauth2/realms/root/authorize?
response_type=code&client_id=myClient"\
"&scope=profile&redirect_uri=http://www.example.net"
```

The `/oauth2/authorize` endpoint can take additional parameters, such as:

- `module` and `service`. 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 tree or chain to use when authenticating the resource owner.

- `response_mode=form_post`. Use this parameter 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.

- `code_challenge`. Use this parameter when *Proof Key for Code Exchange* (PKCE) support is enabled in the OAuth2 Provider service. To configure it, navigate to Realms > *Realm Name* > Services > OAuth2 Provider > Advanced and enable the Code Verifier Parameter Required property. For more information about the PKCE support, see Proof Key for Code Exchange by OAuth Public Clients - *RFC 7636*.

`/oauth2/access_token`

Token endpoint defined in RFC 6749, used to obtain an access token from the authorization server.

Also used to obtain an access token in the OAuth 2.0 Device flow. For more information, see "OAuth 2.0 Device Flow Endpoints".

The `/oauth2/access_token` endpoint can take an additional parameter, `auth_chain=authentication-chain`, which allows clients to specify the authentication chain to use when supporting the Resource Owner Password Credentials grant type.

The following example shows how a client can specify the authentication chain, `myAuthChain`:

```
$ curl \
--request POST
 \
--user "myClientID:password"
 \
--data "grant_type=password&username=amadmin&password=cangetinam&scope=profile&auth_chain=myAuthChain"
 \
https://openam.example.com:8443/openam/oauth2/access_token
```

> **Note**
>
> To use the Resource Owner Password Credentials grant type, as described in RFC 6749, the default authentication chain in the relevant realm must allow authentication using only a username and password, for example by using a `DataStore` module. Attempting to use the Resource Owner Password Credentials grant type with a chain that requires any additional input returns an HTTP `500 Server Error` message.

The `/oauth2/access_token` 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.

For example, if the OAuth 2.0 provider is configured for the `/customers` realm, then use `/oauth2/realms/root/realms/customers/access_token`.

/oauth2/device

Device flow endpoint as defined by the Internet-Draft OAuth 2.0 Device Flow, used by a client device to obtain a device code or an access token.

Example: `https://openam.example.com:8443/openam/oauth2/realms/root/device/code`

For more information, see "OAuth 2.0 Device Flow Endpoints".

/oauth2/token/revoke

When a user logs out of an application, the application revokes any OAuth 2.0 tokens (access and refresh tokens) that are associated with the user. The client can also revoke a token without the need of an `SSOToken` by sending a request to the `/oauth2/token/revoke` endpoint as follows:

```
$ curl \
--request POST
 \
--data "token=d06ab31e-9cdb-403e-855f-bd77652add84"
 \
--data "client_id=MyClientID"
 \
--data "client_secret=password" \
https://openam.example.com:8443/openam/oauth2/realms/root/token/revoke
```

If you are revoking an access token, then that token will be revoked. If you are revoking a refresh token, then both the refresh token and any other associated access tokens will also be revoked. *Associated access tokens* means that any other tokens that have come out of the same authorization grant will also be revoked. For cases where a client has multiple access tokens for a single user that were obtained via different authorization grants, then the client will have to make multiple calls to the `/oauth2/token/revoke` endpoint to invalidate each token.

/oauth2/tokeninfo

Endpoint *not* defined in RFC 6749, used to validate tokens, and to retrieve information, such as scopes.

The `/oauth2/tokeninfo` endpoint takes an HTTP GET on `/oauth2/tokeninfo?access_token=token-id`, and returns information about the token.

Resource servers — or any party having the token ID — can get token information through this endpoint without authenticating. This means any application or user can validate the token without having to be registered with AM.

Given an access token, a resource server can perform an HTTP GET on `/oauth2/tokeninfo?access_token=token-id` to retrieve a JSON object indicating `token_type`, `expires_in`, `scope`, and the `access_token` ID.

Example: `https://openam.example.com:8443/openam/oauth2/realms/root/tokeninfo`

The following example shows AM issuing an access token, and then returning token information:

```
$ curl \
```

```
--request POST
 \
--user "myClientID:password"
 \
--data "grant_type=password&username=demo&password=changeit&scope=cn%20mail" \
https://openam.example.com:8443/openam/oauth2/realms/root/access_token
{
  "expires_in": 599,
  "token_type": "Bearer",
  "refresh_token": "f6dcf133-f00b-4943-a8d4-ee939fc1bf29",
  "access_token": "f9063e26-3a29-41ec-86de-1d0d68aa85e9"
 }

$ curl https://openam.example.com:8443/openam/oauth2/realms/root/tokeninfo
\
?access_token=f9063e26-3a29-41ec-86de-1d0d68aa85e9
{
  "mail": "demo@example.com",
  "grant_type":"password",
  "scope": [
      "mail",
      "cn"
  ],
  "cn": "demo",
  "realm": "/",
  "cnf": {
      "jwk": {
          "alg": "RS512",
          "e": "AQAB",
          "n": "k7qLlj...G2oucQ",
          "kty": "RSA",
          "use": "sig",
          "kid": "myJWK"
      }
  }
  "token_type": "Bearer",
  "expires_in": 577,
  "client_id": "MyClientID",
  "access_token": "f9063e26-3a29-41ec-86de-1d0d68aa85e9"
}
```

> **Note**
>
> Running a GET method to the `/oauth2/tokeninfo` endpoint as shown in the previous example writes the token ID to the access log. To not expose the token ID in the logs, send the OAuth 2.0 access token as part of the authorization bearer header:
>
> ```
> $ curl \
> --request GET
>  \
> --header "Authorization Bearer aec6b050-b0a4-4ece-a86f-bd131decbb9c" \
> "https://openam.example.com:8443/openam/oauth2/tokeninfo"
> ```

The resource server making decisions about whether the token is valid can thus use the `/oauth2 /tokeninfo` endpoint to retrieve expiration information about the token. Depending on the scopes implementation, the JSON response about the token can also contain scope information. As

described in "Using Your Own Client and Resource Server", the default scopes implementation in AM considers scopes to be names of attributes in the resource owner's user profile. Notice that the JSON response contains the values for those attributes from the user's profile, as in the preceding example, with scopes set to `mail` and `cn`.

/oauth2/introspect

Endpoint defined in RFC7662 - OAuth 2.0 Token Introspection, used to retrieve metadata about a token, such as approved scopes and the context in which the token was issued.

Given an access token, a client can perform an HTTP POST on `/oauth2/introspect?token=`*`access_token`* to retrieve a JSON object indicating the following:

active

Is the token active.

scope

A space-separated list of the scopes associated with the token.

client_id

Client identifier of the client that requested the token.

user_id

The user who authorized the token.

token_type

The type of token.

exp

When the token expires, in seconds since January 1 1970 UTC.

sub

Subject of the token.

iss

Issuer of the token.

cnf

Confirmation key claim containing the optional decoded JSON web key (JWK) associated with the access token. For more information, see "OAuth 2.0 JSON Web Token Proof-of-Possession".

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 `/oauth2/introspect` endpoint requires authentication. Clients can use one of the following three methods to authenticate to the endpoint:

- **Basic Authorization**. Authenticate by passing an authorization header with a bearer type of "Basic" with a value of the base64-encoded string of *client_id*:*client_secret*. For an example, see below.

- **Header Values**. Authenticate by passing in the *client_id* and *client_secret* as header values.

- **JWT Bearer Token**. Authenticate by passing in a JWT assertion using the `client_assertion_type` query parameter set to `urn:ietf:params:oauth:client-assertion-type:jwt-bearer` and a `client_assertion` query parameter that contains the JWT assertion. For an example, see "JWT Bearer Profile".

The next example shows the response when authenticating to the `/oauth2/introspect` endpoint with basic authorization:

```
$ curl \
--request POST
 \
--header "Authorization: Basic ZGVtbzpjaGFuZ2VpdA==" \
https://openam.example.com:8443/openam/oauth2/realms/root/introspect
\
?token=f9063e26-3a29-41ec-86de-1d0d68aa85e9
{
    "active": true,
    "scope": "mail cn",
    "client_id": "myOAuth2Client",
    "user_id": "demo",
    "token_type": "access_token",
    "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
    }
}
```

Before the resource owner granted consent to the client, the resource owner authenticated with AM, and AM assigned an authentication level of 0. Notice that AM returns that authentication level as the value of the `auth_level` claim.

> **Note**
>
> Running a POST method to the `/oauth2/introspect` endpoint as shown in the previous example writes the token ID to the access log. To hide the token ID in the logs, send the OAuth 2.0 access token as part of the POST body:

```
$ curl \
--request POST
 \
--header "Authorization: Basic ZGVtbzpjaGFuZ2VpdA=="
 \
--data "token=f9063e26-3a29-41ec-86de-1d0d68aa85e9"
"https://openam.example.com:8443/openam/oauth2/introspect"
```

3.1.1. Using OAuth 2.0 JSON Web Token Proof-of-Possession

To use the proof-of-possession feature and associate a JSON web key with an OAuth 2.0 access token, perform the following steps:

To Use 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. To verify proof-of-possession of the access token, an OAuth 2.0 resource server can challenge your OAuth 2.0 client to decrypt a message or nonce that it has encrypted with the public key. The private key is required to decrypt such challenges.

    Your OAuth 2.0 client should never reveal the private key, however. Only the public key from the pair should be added to the request for an access token. The key should be represented in JWK format and may resemble the following example:

    ```
    {
        "jwk": {
            "alg": "RS256",
            "e": "AQAB",
            "n": "xea7Tb7rbQ4ZrHNKrg...QFXtJ-didSTtXWCWUlQrcj0hnDjvkuUFWoSQ_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.

2. Base64-encode the JWK. The result may resemble the following example:

    ```
    ew0KICAgICJKV0siOiB7DQogICAgICAgICJhbGciOiAiUlMyNTYiLA0KICAgICAgICAiZSI6IC
    JBUUFCIiwNDQogICAgICAgICJraWQiOiAibXlQdWJsaWNKU09OV2ViS2V5Ig0KICAgICAgIH0NCn0=
    ```

FORGEROCK

3. Include the base64-encoded JWK as the value of the `cnf_key` parameter in the request to the authorization server for an access token:

```
$ curl \
--request POST
 \
--header "Authorization: Basic bXlDbGllbnQ6cGFzc3dvcmQ="
 \
--data
 "grant_type=client_credentials"\
--data "scope=profile"
 \
--data "cnf_key=ew0KICAgICJKV0siOiB7DQogICAgICAgICAgICJhb
            GciOiAiUlMyNTYiLA0KICAgICAgICAgICAiZSI6IC
            JBUUFCIiwNDQogICAgICAgICAICJraWQiOiAibXhl
            QdWJsaWNKU09OV2ViS2V5Ig0KICAgIH0NCn0=" \
 https://openam.example.com:8443/openam/oauth2/realms/root/access_token
```

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

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

4. To access a resource on the resource server:

   a. Present the value of the `access_token` element to the resource server.

   b. If stateful 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 bXlDbGllbnQ6cGFzc3dvcmQ=" \
https://openam.example.com:8443/openam/oauth2/realms/root/introspect
\
?token=f08f1fcf-3ecb-4120-820d-fb71e3f51c04
{
    "active": true,
    "scope": "profile",
    "client_id": "myClient",
    "user_id": "myClient",
    "token_type": "access_token",
    "exp": 1477666348,
    "sub": "myClient",
    "iss": "http://openam.example.com:8080/openam/oauth2/realms/root",
    "cnf": {
        "jwk": {
            "alg": "RS256",
            "e": "AQAB",
            "n": "xea7Tb7rbQ4ZrHNKrg...QFXtJ-didSTtXWCWU1Qrcj0hnDjvkuUFWoSQ_7Q",
            "kty": "RSA",
            "use": "enc",
            "kid": "myPublicJSONWebKey"
        },
    "auth_level": 0
    }
}
```

c. 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.

3.2. OAuth 2.0 Device Flow Endpoints

If a client device has a limited user interface, it can obtain an OAuth 2.0 device code and ask a user to authorize the client on a more full-featured user agent, such as an Internet browser.

AM provides the `/oauth2/device/code`, `/oauth2/device/user`, and `/oauth2/access_token` endpoints to support the OAuth 2.0 Device Flow.

The following procedures show how to use the OAuth 2.0 device flow endpoints:

- "To Request a User Code in the OAuth 2.0 Device Flow".

- "To Grant Consent in the OAuth 2.0 Device Flow".

- "To Poll for Authorization in the OAuth 2.0 Device Flow".

> **Note**
>
> In the examples `nonce` and `state` OAuth 2.0 parameters are omitted, but should be used in production.

To Request a User Code in the OAuth 2.0 Device Flow

Devices can display a user code and instructions to a user, which can be used on a separate client to provide consent, allowing the device to access resources.

As user codes may be displayed on lower resolution devices, the list of possible characters used has been optimized to reduce ambiguity. User codes consist of a random selection of eight of the following characters:

```
234567ABCDEFGHIJKLMNOPQRSTVWXYZabcdefghijkmnopqrstvwxyz
```

To request a user code in the OAuth 2.0 device flow:

1. Ensure that an OAuth 2.0 client profile is configured in AM, as described in "OAuth 2.0 and OpenID Connect 1.0 Client Settings".

2. Create a POST request to the `/oauth2/device/code` endpoint to acquire a device code. The following URL parameters are required:

   `response_type`

   > Specifies the response type required by the request. Must be set to `token`.

   `scope`

   > Specifies the list of scopes requested by the client, separated by URL-encoded space characters.

   `client_id`

   > Specifies the name of the client agent profile in AM.

   ```
   $ curl \
   --data response_type=token
    \
   --data scope=phone%20email%20profile%20address
    \
   --data client_id=myDeviceAgentProfile \
   http://openam.example.com:8080/openam/oauth2/realms/root/device/code
   ```

   On success, AM returns a verification URL, 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.

3. 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. See "To Grant Consent in the OAuth 2.0 Device Flow".
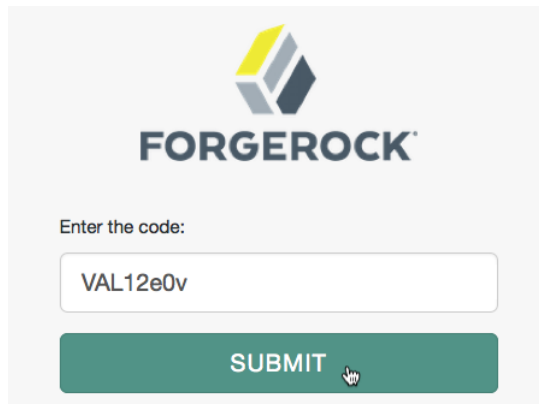
4. The client device should also begin polling the authorization server for the access token, once consent has been given. See "To Poll for Authorization in the OAuth 2.0 Device Flow".

To Grant Consent in the OAuth 2.0 Device Flow

OAuth 2.0 device flow requires that the user grants consent to allow the client device to access resources.

- You can grant consent in the OAuth 2.0 device flow using the AM user interface, or by making calls to AM endpoints.

    - To use the AM user interface, the user should visit the verification URL in a web browser and enter the user code:

OAuth 2.0 User Code

The user can then authorize the device flow client by allowing the requested scopes:

OAuth 2.0 Consent Page

- To use endpoint calls, create a POST request to the `/oauth2/device/user` endpoint. The following URL parameter is required:

`user_code`

   The user code as provided by the `/oauth2/device/code` endpoint.

The form data should be in `x-www-form-urlencoded` format, and contain the following fields:

`user_code`

   The user code as provided by the `/oauth2/device/code` endpoint.

`scope`

   Specifies the list of scopes consented to by the user, separated by URL-encoded space characters.

client_id

> Specifies the name of the client agent profile in AM.

response_type

> Must be `token`.

decision

> To allow client access, specify `allow`. Any other value will deny consent.

csrf

> Duplicates the contents of the `iPlanetDirectoryPro` cookie, which contains the SSO token of the user granting access.
>
> Duplicating the cookie value helps prevent against Cross-Site Request Forgery (CSRF) attacks.

The `iPlanetDirectoryPro` cookie is required and should contain the SSO token of the user granting access to the client.

```
$ curl \
-X POST
 \
--header "Cookie: iPlanetDirectoryPro=AQIC5..."
 \
--header "Content-Type: application/x-www-form-urlencoded"
 \
--data scope=phone%20email%20profile%20address
 \
--data user_code=VAL12e0v
 \
--data response_type=token
 \
--data client_id=myDeviceAgentProfile
 \
--data decision=allow
 \
--data csrf=AQIC5... \
http://openam.example.com:8080/openam/oauth2/realms/root/device/user?user_code=VAL12e0v
```

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 : "http://openam.example.com:8080/openam/XUI",
    realm : "//XUI",
    done: true
}
```

If the supplied user code has already been used, or is incorrect, the following is returned:

```
pageData = {
    locale: "*",
    errorCode: "not_found",
    realm : "/",
    baseUrl : "http://openam.example.com:8080/openam/XUI"
}
```

If the user gives consent, AM adds the OAuth 2.0 client to the user's profile page in the *Authorized Apps* section. For more information, see "User Consent Management".

> **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 Poll for Authorization in the OAuth 2.0 Device Flow

• On the client device, create a POST request to poll the `/oauth2/access_token` endpoint to request an access token. Include the client ID, client secret, and the device code as query parameters in the request. You must also specify a `grant_type` of `http://oauth.net/grant_type/device/1.0`.

The client device must wait for the number of seconds previously provided as the value of `interval` between polling AM for an access token.

```
$ curl \
--data client_id=myDeviceAgentProfile
 \
--data client_secret=password
 \
--data grant_type=http://oauth.net/grant_type/device/1.0
 \
--data code=7a95a0a4-6f13-42e3-ac3e-d3d159c94c55 \
http://openam.example.com:8080/openam/oauth2/realms/root/access_token
{
    "scope": "phone email address profile",
    "code": "20c1fc0c-3153-4a11-8d1f-d815c1a522b5"
}
```

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"
}
```

3.3. OAuth 2.0 Resource Set Endpoint

AM provides a read-only REST endpoint for viewing a resource set registered to a particular user. The endpoint is /users/*user*/oauth2/resourcesets/ *resource_set_ID*.

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*.

The following URL could be used to display the resource sets belonging to a user named demo in a subrealm of the top-level realm named myrealm:

```
https://openam.example.com:8443/openam/json/realms/root/realms/myrealm/users/demo/oauth2/resourcesets
/43225628-4c5b-4206-b7cc-5164da81decd0
```

To read a resource set, either the resource set owner or an administrator such as amadmin must have logged in to AM (the authorization server) and have been issued an SSO token.

To Read an OAuth 2.0 Resource Set

- Create a GET request to the resourcesets endpoint, including the SSO token in a header based on the configured session cookie name (for example: iPlanetDirectoryPro), and with the resource set ID in the URL.

  The following example uses an SSO token acquired by the amadmin user to view a resource set, and related policy, belonging to the demo user in the top level realm:

```
$ curl \
--header "iPlanetDirectoryPro: AQIC5wM2LY4Sfcxs...EwNDU2NjE0*" \
https://openam.example.com:8443/openam/json/realms/root/users/demo
\
/oauth2/resourcesets/43225628-4c5b-4206-b7cc-5164da81decd0
{
    "scopes": [
        "http://photoz.example.com/dev/scopes/view",
```

```
            "http://photoz.example.com/dev/scopes/comment"
        ],
        "_id": "43225628-4c5b-4206-b7cc-5164da81decd0",
        "resourceServer": "UMA-Resource-Server",
        "name": "My Videos",
        "icon_uri": "http://www.example.com/icons/cinema.png",
        "policy": {
            "permissions": [
                {
                    "subject": "user.1",
                    "scopes": [
                        "http://photoz.example.com/dev/scopes/view"
                    ]
                },
                {
                    "subject": "user.2",
                    "scopes": [
                        "http://photoz.example.com/dev/scopes/comment",
                        "http://photoz.example.com/dev/scopes/view"
                    ]
                }
            ]
        },
        "type": "http://www.example.com/rsets/videos"
}
```

> **Tip**
>
> You can specify the fields that are returned with the `_fields` query string filter. For example `?_fields=scopes, resourceServer, name`

On success, an HTTP 200 OK status code is returned, with a JSON body representing the resource set. If a policy relating to the resource set exists, a representation of the policy is also returned in the JSON.

If the specified resource set 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 set owner or an administrator, an HTTP 403 Forbidden status code is returned, as follows:

```
{
    "code": 403,
    "reason": "Forbidden",
    "message": "User, user.1, not authorized."
}
```

3.4. OAuth 2.0 Token Administration Endpoint (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 deprecated and it does not work with stateless OAuth 2.0 tokens.
>
> Use the following endpoints instead:
>
> - `/oauth2/introspect`. Use this endpoint to read and list OAuth 2.0 tokens.
>
> - `/oauth2/token/revoke`. Use this endpoint to delete (revoke) OAuth 2.0 tokens.

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
 \
--user "myClientID:password"
 \
--data "grant_type=password&username=demo&password=changeit&scope=cn" \
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 a header the SSO token of the current user. The following example shows a search for the demo user's access tokens:

```
$ curl \
 --request GET \
 --header "iplanetDirectoryPro: AQIC5wM2LY4Sfcw..." \
 https://openam.example.com:8443/openam/frrest/oauth2/token/?_queryId=access_token
{
    "result": [
    {
      "_rev": "1753454107",
      "tokenName": [
        "access_token"
      ],
      "expireTime": "Indefinitely",
      "scope": [
        "openid"
      ],
      "grant_type": [
        "password"
      ],
      "clientID": [
        "myClientID"
      ],
      "tokenType": [
        "Bearer"
      ],
      "redirectURI": [],
      "nonce": [],
      "realm": [
        "/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 in a header the SSO token of an administrative user, such as `amAdmin`. 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
 \
--user "myClientID:password"
 \
--data "grant_type=password&username=demo&password=changeit&scope=cn" \
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"
}
```

3.5. OAuth 2.0 Client Administration Endpoint

The OAuth 2.0 administration endpoint lets AM administrators and agent administrators create (that
is, register) and delete OAuth 2.0 clients.

AM exposes this endpoint at `/json/realm-config/agents/OAuth2Client/`, for example `https://openam.example
.com:8443/openam/json/realms/root/realm-config/agents/OAuth2Client/`.

You can use the AM API Explorer for detailed information about the parameters supported by this
endpoint, and to test it against your deployed AM instance.

In the AM console, click the Help icon, and then navigate to API Explorer > /realm-config > /agents > /OAuth2Client.

To create an OAuth 2.0 client, perform an HTTP POST to `/realm-config/agents/OAuth2Client/Client ID` with a JSON object fully specifying the client.

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`.

The following example creates an OAuth 2.0 client named `myClient` in a subrealm of the top-level realm named `subrealm1`:

```
$ curl \
 --request PUT \
 --header 'Content-Type: application/json' --header 'Accept: application/json' \
 --header "iplanetDirectoryPro: AQIC5wM...3MTYxOA..*" \
 --data '{
     "userpassword": "secret12",
     "com.forgerock.openam.oauth2provider.clientType":"Confidential",
     "com.forgerock.openam.oauth2provider.redirectionURIs":
         ["www.client.com","www.example.com"],
     "com.forgerock.openam.oauth2provider.scopes":
         ["cn","sn"],
     "com.forgerock.openam.oauth2provider.defaultScopes":
         ["cn"],
     "com.forgerock.openam.oauth2provider.name":
         ["My Test Client"],
     "com.forgerock.openam.oauth2provider.description":
         ["OAuth 2.0 Client"]
 }' \
 'http://openam.example.com:8080/openam/json/realms/root/realms/subrealm1/realm-config/agents/OAuth2Client
/testClient'
{
    "_id" : "testClient",
    "_rev" : "2001898072",
    "com.forgerock.openam.oauth2provider.redirectionURIs" : [ "www.client.com", "www.example.com" ],
    "com.forgerock.openam.oauth2provider.tokenEndPointAuthMethod" : "client_secret_basic",
    "com.forgerock.openam.oauth2provider.jwks" : null,
    "com.forgerock.openam.oauth2provider.claims" : [ "[0]=" ],
    "com.forgerock.openam.oauth2provider.jwtTokenLifeTime" : 0,
    "com.forgerock.openam.oauth2provider.accessTokenLifeTime" : 0,
    "com.forgerock.openam.oauth2provider.defaultMaxAge" : 600,
    "idTokenEncryptionEnabled" : false,
    "userpassword" : "secret12",
    "com.forgerock.openam.oauth2provider.contacts" : [ "[0]=" ],
    "com.forgerock.openam.oauth2provider.subjectType" : "Public",
    "com.forgerock.openam.oauth2provider.postLogoutRedirectURI" : [ "[0]=" ],
    "com.forgerock.openam.oauth2provider.clientType" : "Confidential",
    "com.forgerock.openam.oauth2provider.scopes" : [ "cn", "sn" ],
    "com.forgerock.openam.oauth2provider.description" : [ "OAuth 2.0 Client" ],
    "idTokenPublicEncryptionKey" : null,
    "idTokenEncryptionMethod" : "A128CBC-HS256",
    "com.forgerock.openam.oauth2provider.jwksURI" : "http://openam.example.com:8080/openam/oauth2/realms/
root/realms/subrealm1/connect/jwk_uri",
    "com.forgerock.openam.oauth2provider.clientJwtPublicKey" : null,
    "com.forgerock.openam.oauth2provider.authorizationCodeLifeTime" : 0,
```

```
        "com.forgerock.openam.oauth2provider.accessToken" : null,
        "com.forgerock.openam.oauth2provider.sectorIdentifierURI" : null,
        "idTokenEncryptionAlgorithm" : "RSA1_5",
        "com.forgerock.openam.oauth2provider.refreshTokenLifeTime" : 0,
        "com.forgerock.openam.oauth2provider.clientSessionURI" : null,
        "com.forgerock.openam.oauth2provider.defaultScopes" : [ "cn" ],
        "com.forgerock.openam.oauth2provider.clientName" : [ "[0]=" ],
        "com.forgerock.openam.oauth2provider.idTokenSignedResponseAlg" : "HS256",
        "com.forgerock.openam.oauth2provider.defaultMaxAgeEnabled" : false,
        "com.forgerock.openam.oauth2provider.name" : [ "My Test Client" ],
        "com.forgerock.openam.oauth2provider.responseTypes" : [
            "[6]=code token id_token",
            "[0]=code",
            "[4]=token id_token",
            "[2]=id_token",
            "[3]=code token",
            "[1]=token",
            "[5]=code id_token" ],
        "com.forgerock.openam.oauth2provider.publicKeyLocation" : "jwks_uri",
        "sunIdentityServerDeviceStatus" : "Active",
        "isConsentImplied" : false,
        "_type" : {
          "_id" : "OAuth2Client",
          "name" : "OAuth2 Clients",
          "collection" : true
        }
}
```

To delete an OAuth 2.0 client, perform an HTTP DELETE on `/json/realm-config/agents/`
`OAuth2Client/`*`client-id`*, as in the following example:

```
$ curl \
 --request DELETE \
 --header "iplanetDirectoryPro: AQIC5wM...3MTYxOA..*" \
 https://openam.example.com:8443/openam/json/realms/root/realm-config/agents/OAuth2Client/myClient
{"success":"true"}
```

> **Tip**
>
> 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`*.
>
> The following example deletes an OAuth 2.0 client with ID `myClient` from a subrealm in the top-level realm
> named `myRealm`:

```
$ curl \
  --request DELETE \
  --header "iplanetDirectoryPro: AQIC5wM...3MTYxOA..*" \
  https://openam.example.com:8443/openam/json/realms/root/realms/myRealm/realm-config/agents/
OAuth2Client/myClient
{"success":"true"}
```

3.6. OAuth 2.0 User Applications Endpoint

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
}
```

3.7. 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
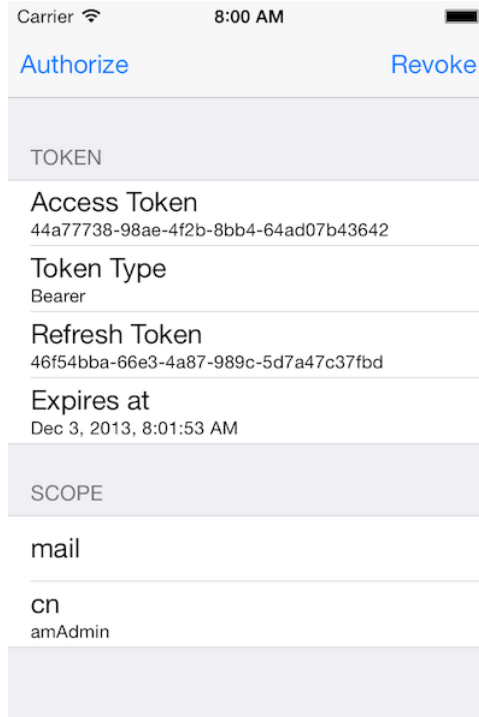
Chapter 4
Customizing OAuth 2.0

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

4.1. 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 profile attributes for the resource owner. When a resource server or other entity uses the access token to get token information from AM, AM 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 this 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.

4.1.1. 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 example plugin sets whether `read` and `write` permissions were granted.

```
public class CustomScopeValidator implements ScopeValidator {
    @Override
    public Set<String> validateAuthorizationScope(
            ClientRegistration clientRegistration,
            Set<String> scope,
            OAuth2Request oAuth2Request) {
        if (scope == null || scope.isEmpty()) {
            return clientRegistration.getDefaultScopes();
        }

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

FORGEROCK

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

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

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

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

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

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

@Override
public UserInfoClaims getUserInfo(
        AccessToken token,
        OAuth2Request request)
        throws UnauthorizedClientException {
    Map<String, Object> response = mapScopes(token);
```

```java
        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
    }
}
```

4.1.2. 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.jar` in the `/path/to/openam-samples-external/openam-scope-sample/target` directory of the project.

4.1.3. 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.

4.1.4. 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.

Chapter 5
Reference

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

5.1. 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

Internet Draft: 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

5.2. OAuth2 Provider

amster service name: `oauth-oidc`

5.2.1. 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`

Authenticity Secret

A secret to use when signing data that will be sent back to AM so that authenticity can be assured when they are presented back to OpenAM.

amster attribute: `idTokenAuthenticitySecret`

ID Token Signing Key Alias for Agent Clients

The alias for the RSA key that should be used signing ID tokens for Agent OAuth2 Clients

Default value: `test`

amster attribute: `agentIdTokenSigningKeyAlias`

Stateless Grant Token Upgrade Compatibility Mode

Enable OpenAM to consume and create stateless OAuth 2.0 tokens in two different formats simultaneously.

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

Default value: `false`

amster attribute: `statelessGrantTokenUpgradeCompatibilityMode`

5.2.2. Core

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

Use Stateless Access & Refresh Tokens

When enabled, OpenAM 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 token will never expire.

Default value: `604800`

amster attribute: `refreshTokenLifetime`

Access Token Lifetime (seconds)

The time an access token is valid for, in seconds.

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`

5.2.3. Advanced

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

Custom Login URL Template

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

Supports Freemarker syntax, with the following variables:

Variable	Description
`gotoUrl`	The URL to redirect to after login.

`acrValues`	The Authentication Context Class Reference (acr) values for the authorization request.
`realm`	The OpenAM realm the authorization request was made on.
`module`	The name of the OpenAM authentication module requested to perform resource owner authentication.
`service`	The name of the OpenAM authentication chain requested to perform resource owner authentication.
`locale`	A space-separated list of locales, ordered by preference.

The following example template redirects users to a non-OpenAM 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 OpenAM 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 OpenAM console.

amster attribute: `defaultScopes`

OAuth2 Token Signing Algorithm

Algorithm used to sign stateless OAuth 2.0 tokens in order to detect tampering.

OpenAM 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`

Stateless Token Compression

Whether stateless access and refresh tokens should be compressed.

amster attribute: `tokenCompressionEnabled`

Token Signing HMAC Shared Secret

Base64-encoded key used by HS256, HS384 and HS512.

amster attribute: `tokenSigningHmacSharedSecret`

Token Signing RSA Public/Private Key Pair

The public/private key pair used by RS256.

The public/private key pair will be retrieved from the keystore referenced by the property `com.sun` `.identity.saml.xmlsig.keystore`.

Default value: `test`

amster attribute: `keypairName`

Token Signing ECDSA Public/Private Key Pair Alias

The list of public/private key pairs used for the elliptic curve algorithms (ES256/ES384/ES512). Add an entry to specify an alias for a specific elliptic curve algorithm, for example `ES256|` `es256Alias`.

Each of the public/private key pairs will be retrieved from the keystore referenced by the property `com.sun.identity.saml.xmlsig.keystore`.

Default value:

```
ES512|es512test
ES384|es384test
ES256|es256test
```

amster attribute: `tokenSigningECDSAKeyAlias`

Enable Stateless Token Encryption

Whether stateless 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`

Token Encryption Secret Key Alias

The secret key used for encryption.

The secret key will be retrieved from the keystore referenced by the property `com.sun.identity.saml.xmlsig.keystore`.

Default value: `directenctest`

amster attribute: `tokenEncryptionKeyAlias`

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 draft specification for this feature.

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, `updated_at` claim uses the `createdTimestampAttribute` attribute. For more information, see "Configuring as an OP for Mobile Connect" in the *OpenID Connect 1.0 Guide*.

amster attribute: `modifiedTimestampAttribute`

Created Timestamp Attribute Name

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

This attribute is paired together with the `Modified Timestamp Attribute Name` (`modifyTimestampAttribute`). 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, `updated_at` claim uses the `createdTimestampAttribute` attribute. For more information, see "Configuring as an OP for Mobile Connect" in the *OpenID Connect 1.0 Guide*.

amster attribute: `createdTimestampAttribute`

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 Resource Owner Password Credentials grant type requires the `grant_type=password` parameter.

Default value: `false`

amster attribute: `moduleMessageEnabledInPasswordGrant`

5.2.4. 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`

5.2.5. 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`.

OpenAM 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:

```
RS384
PS284
ES384
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.

OpenAM 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
RSA1_5
A256KW
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.

OpenAM 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`

Token Encryption RSA Public/Private Key Pair Alias

The list of public/private key pairs used for the RSA algorithms (RSA1_5/RSA-OAEP/RSA-OAEP-256). Add an entry to specify an alias for a specific RSA algorithm, for example `RSA1_5| rsa1_5Alias`.

Each of the public/private key pairs will be retrieved from the keystore referenced by the property `com.sun.identity.saml.xmlsig.keystore`.

Default value:

```
RSA1_5|test
RSA-OAEP|test
RSA-OAEP-256|test
```

amster attribute: `tokenEncryptionSigningKeyAlias`

5.2.6. 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 OpenAM 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 OpenAM 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 parameterOpenAM 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
RS384
ES384
HS256
HS512
ES256
RS256
HS384
ES512
PS256
PS512
RS512
```

amster attribute: `supportedRequestParameterSigningAlgorithms`

Request Parameter Encryption Algorithms Supported

Encryption algorithms supported to decrypt Request parameter.

OpenAM 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
RSA1_5
A256KW
dir
A192KW
```

amster attribute: `supportedRequestParameterEncryptionAlgorithms`

Request Parameter Encryption Methods Supported

Encryption methods supported to decrypt Request parameter.

OpenAM 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`

Require Pre-registered request_uri Values

When enabled, any `request_uri` values used must be pre-registered using the `request_uris` registration parameter.

Default value: `false`

amster attribute: `requireRequestUriRegistration`

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`

5.2.7. 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`

5.2.8. Consent

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

Saved Consent Attribute Name

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

When the resource owner chooses to save the decision to authorize access for a client application, then OpenAM 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.

OpenAM 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
RS384
ES384
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.

OpenAM 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.

OpenAM 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.

OpenAM 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
RS384
ES384
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.

OpenAM 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: `supportedRcsResponseEncryptionAlgorithms`

Remote Consent Service Response Encryption Methods Supported

Encryption methods supported to decrypt Remote Consent Service responses.

OpenAM 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`

5.3. Remote Consent Service

amster service name: `RemoteConsentService`

5.3.1. 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 referencedin 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 jwk_uri

The jwk_uri for retrieving the authorization server signing and encryption keys.

amster attribute: `jwksUriAS`

JWK Store Cache Timeout (in minutes)

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

Default value: `60`

amster attribute: `jwkStoreCacheTimeout`

JWK Store Cache Miss Cache Time (in minutes)

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

Default value: `1`

amster attribute: `jwkStoreCacheMissCacheTime`

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`

5.4. 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*.

5.4.1. 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 (#).

Redirection URIs are required for OpenID Connect 1.0 clients.

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`

5.4.2. 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.

Only required if the *Require request URI supported* property is enabled in the OAuth2 Provider service. See "Advanced OpenID Connect"

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.

For more information, see 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.

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.

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

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

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

Post Logout Redirect URIs

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

Client Session URI

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

Default Max Age

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

Minimum value: `1`.

Default: `600`

Default Max Age Enabled

Enable the default max age feature.

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`

5.4.4. Signing and Encryption

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 maximum amount of time, in milliseconds, that the content of the JWKS URI can be cached before being refreshed. This avoids fetching the JWKS URI content for every token encryption.

Default: `3600000`

JWKs URI content cache miss cache time

Specify the minimum amount of time, in milliseconds, that the content of the JWKS URI is cached. This avoids fetching the JWKS URI content for every token signature verification, for example if the key ID (`kid`) is not in the JWKS content already cached.

Default: `60000`

Token Endpoint Authentication Signing Algorithm

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

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

Default: `RS256`

Json Web Key

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

ID Token Signing Algorithm

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

Enable ID Token Encryption

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

ID Token Encryption Algorithm

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

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

ID Token Encryption Method

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

Default value: `A128CBC-HS256`.

Client ID Token Public Encryption Key

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

Client JWT Bearer Public Key Certificate

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

The following is an example of the certificate:

```
-----BEGIN CERTIFICATE-----
MIIDETCCAfmgAwIBAgIEU8SXLjANBgkqhkiG9w0BAQsFADA5MRswGQYDVQQKExJvcGVuYW0uZXhh
bXBsZS5jb20xGjAYBgNVBAMTEWp3dC1iZWFyZXItY2xpZW50MB4XDTE0MTAyNzExNTY1NloXDTI0
MTAyNDExNTYlowOTEbMBkGA1UEChMSb3BlbmFtLmV4YW1wbGUuY29tMRowGAYDVQQDExFqd3Qt
YmVhcmVyLWNsaWVudDCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAID4ZZ/DIGEBr4QC
2uz0GYFOCUlAPanxX21aYHSvELsWyMa7DJlD+mnjaF8cPRRMkhYZFXDJo/AVcjyblyT3ntqL+2Js
3D7TmS6BSjkxZWsJHyhJIYEoUwwloc0kizgSm15MwBMcbnksQVN5VWiOe4y4JMbi30t6k38lM62K
KtaSPP6jvnW1LTmL9uiqLWz54AM6hU3NlCI3J6Rfh8waBIPAEjmHZNquOl2uGgWumzubYDFJbomL
SQqO58RuKVaSVMwDbmENtMYWXIKQL2xTt5XAbwEQEgJ/zskwpA2aQt1HE6de3UymOhONhRiu4rk3
AIEnEVbxrvy4Ik+wXg7LZVsCAwEAAaMhMB8wHQYDVR0OBBYEFIuI7ejuZTg5tJsh1XyRopGOMBcs
MA0GCSqGSIb3DQEBCwUAA4IBAQBM/+/tYYVIS6LvPl3mfE8V7x+VPXqj/uK6UecAbfmRTrPk1ph+
jjI6nmLX9ncomYALWL/JFiSXcVsZt3/412fOqjakFVS0PmK1vEPxDlav1drnVA33icy1wORRRu5/
qA6mwDYPAZSbm5cDVvCR7Lt6VqJ+D0V8GABFxUw9IaX6ajTqkWhldY77usvNeTD0Xc4R7OqSBrnA
SNCaUlJogWyzhbFlmE9Ne28j4RVpbz/EZn0oc/cHTJ6Lryzsivf4uDO1m3M3kM/MUyXc1Zv3rqBj
TeGSgcqEAd6XlGXY1+M/yIeouUTi0Flbk1rNlqJvd57Xb4CEq17tVbGBm0hkECM8
-----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-----
MIIDETCCAfmgAwIBAgIEU8SXLjANBgkqhkiG9w0BAQsFADA5MRswGQYDVQQKExJvcGVuYW0uZXhh
bXBsZS5jb20xGjAYBgNVBAMTEWp3dC1iZWFyZXItY2xpZW50MB4XDTE0MTAyNzExNTY1NloXDTI0
MTAyNDExNTY1NlowOTEbMBkGA1UEChMSb3BlbmFtLmV4YW1wbGUuY29tMRowGAYDVQQDExFqd3Qt
YmVhcmVyLWNsaWVudDCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAID4ZZ/DIGEBr4QC
2uz0GYFOCUlAPanxX21aYHSvELsWyMa7DJlD+mnjaF8cPRRMkhYZFXDJo/AVcjyblyT3ntqL+2Js
3D7TmS6BSjkxZWsJHyhJIYEoUwwloc0kizgSm15MwBMcbnksQVN5VWiOe4y4JMbi30t6k38lM62K
KtaSPP6jvnW1LTmL9uiqLWz54AM6hU3NlCI3J6Rfh8waBIPAEjmHZNquOl2uGgWumzubYDFJbomL
SQqO58RuKVaSVMwDbmENtMYWXIKQL2xTt5XAbwEQEgJ/zskwpA2aQt1HE6de3UymOhONhRiu4rk3
AIEnEVbxrvy4Ik+wXg7LZVsCAwEAAaMhMB8wHQYDVR0OBBYEFIuI7ejuZTg5tJsh1XyRopGOMBcs
MA0GCSqGSIb3DQEBCwUAA4IBAQBM/+/tYYVIS6LvPl3mfE8V7x+VPXqj/uK6UecAbfmRTrPk1ph+
jjI6nmLX9ncomYALWL/JFiSXcVsZt3/412fOqjakFVS0PmK1vEPxDlav1drnVA33icy1wORRRu5/
qA6mwDYPAZSbm5cDVvCR7Lt6VqJ+DOV8GABFxUw9IaX6ajTqkWhldY77usvNeTD0Xc4R7OqSBrnA
SNCaUlJogWyzhbFlmE9Ne28j4RVpbz/EZn0oc/cHTJ6Lryzsivf4uDO1m3M3kM/MUyXc1Zv3rqBj
TeGSgcqEAd6XlGXY1+M/
yIeouUTi0F1bk1rNlqJvd57Xb4CEq17tVbGBm0hkECM8
-----END CERTIFICATE-----
```

Public key selector

Select the public key for this client, which comes from either the `JWKs_URI`, manual JWKs, or X.509 field.

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 JWE 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 JWE 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`

5.4.5. 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.

5.5. 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.

AM supports the 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: `remoteConsentRequestSigningAlgorithm`

Enable consent request Encryption

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

Default: `true`

amster attribute: `remoteConsentRequestEncryptionEnabled`

Consent request Encryption Algorithm

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

AM supports the following encryption algorithms:

- `A128KW` - AES Key Wrapping with 128-bit key derived from the client secret.

- `A192KW` - AES Key Wrapping with 192-bit key derived from the client secret.

- `A256KW` - AES Key Wrapping with 256-bit key derived from the client secret.

- `RSA-OAEP` - RSA with Optimal Asymmetric Encryption Padding (OAEP) with SHA-1 and MGF-1.

- `RSA-OAEP-256` - RSA with OAEP with SHA-256 and MGF-1.

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

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

Default value: `RSA-OAEP-256`

amster attribute: `remoteConsentRequestEncryptionAlgorithm`

Consent request Encryption Method

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

AM supports the following encryption methods:

- `A128GCM`, `A192GCM`, and `A256GCM` - AES in Galois Counter Mode (GCM) authenticated encryption mode.

- `A128CBC-HS256`, `A192CBC-HS384`, and `A256CBC-HS512` - AES encryption in CBC mode, with HMAC-SHA-2 for integrity.

Default value: `A128GCM`

amster attribute: `remoteConsentRequestEncryptionMethod`

Consent response signing algorithm

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

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

- `ES256` - ECDSA with SHA-256 and NIST standard P-256 elliptic curve.

- `ES384` - ECDSA with SHA-384 and NIST standard P-384 elliptic curve.

- `ES512` - ECDSA with SHA-512 and NIST standard P-521 elliptic curve.

- `HS256` - HMAC with SHA-256.

- `HS384` - HMAC with SHA-384.

- `HS512` - HMAC with SHA-512.

- `RS256` - RSASSA-PKCS-v1_5 using SHA-256.

Default value: `RS256`

amster attribute: `remoteConsentResponseSigningAlg`

Consent response encryption algorithm

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

AM supports the following encryption algorithms:

- `A128KW` - AES Key Wrapping with 128-bit key derived from the client secret.

- `A192KW` - AES Key Wrapping with 192-bit key derived from the client secret.

- `A256KW` - AES Key Wrapping with 256-bit key derived from the client secret.

- `RSA-OAEP` - RSA with Optimal Asymmetric Encryption Padding (OAEP) with SHA-1 and MGF-1.

- `RSA-OAEP-256` - RSA with OAEP with SHA-256 and MGF-1.

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

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

Default value: `RSA-OAEP-256`

amster attribute: `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 maximum amount of time, in milliseconds, that the content of the JWKS URI can be cached before being refreshed. This avoids fetching the JWKS URI content for every response decryption.

Default: `3600000`

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

JWKs URI content cache miss cache time

Specify the minimum amount of time, in milliseconds, that the content of the JWKS URI is cached. This avoids fetching the JWKS URI content for every response signature verification, for example if the key ID (`kid`) is not in the JWKS content that is already cached.

Default: `60000`

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

Json Web Key

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

Example:

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

amster attribute: `jwkSet`

Consent Request Time Limit

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

Default: `180`

amster attribute: `requestTimeLimit`

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.

A.1. 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.)

A.2. 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.

A.2.1. 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.

A.2.2. 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".

A.2.3. 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.

A.2.4. 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.

A.2.5. 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.

A.2.6. 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.

A.2.7. 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.

A.2.8. 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.

A.2.9. 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.

A.2.10. 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`:

A.2.10.1. 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 list of values included in a patch are 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" ]
}
```

A.2.10.2. 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".

A.2.10.3. 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.

A.2.10.4. 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".

A.2.10.5. 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.

A.2.10.6. 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" ],
  }
]
```

A.2.10.7. 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"
      }
    }
  }
]
```

A.2.10.8. 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.

A.2.11. 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.

A.2.12. 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.

A.2.13. 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.

A.3. 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.

A.4. 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.

A.4.1. Supported REST API Versions

The REST API version numbers supported in AM 6 are as follows:

Supported protocol versions

The *protocol* versions supported in AM 6 are:

`1.0`

Supported resource versions

The *resource* versions supported in AM 6 are shown in the following table.

Supported resource Versions

Base	End Point	Supported Versions
/json	/authenticate	1.1, 2.0
	/users	1.1, 1.2, 2.0, 2.1, 3.0
	/groups	1.1, 2.0, 2.1, 3.0
	/agents	1.1, 2.0, 2.1, 3.0
	/realms	1.0
	/dashboard	1.0
	/sessions	1.2, 2.1, 3.1
	/serverinfo/*	1.1

Base	End Point	Supported Versions
	/users/*{user}*/devices/trusted	1.0
	/users/*{user}*/uma/policies	1.0
	/applications	1.0, 2.0
	/resourcetypes	1.0
	/policies	1.0, 2.0
	/applicationtypes	1.0
	/conditiontypes	1.0
	/subjecttypes	1.0
	/subjectattributes	1.0
	/decisioncombiners	1.0
	/subjectattributes	1.0
/xacml	/policies	1.0
/frrest	/token	1.0
	/client	1.0

For information about the supported protocol and resource versions available in AM 6, 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.

A.4.2. 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"
 \
--data "{}" \
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".

A.4.3. 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.
```

A.4.4. 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"
 \
--data "{}" \
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...TU3OQ*",
    "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".

A.5. 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=*
```

A.6. Authentication and Logout

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*.

The simplest user name/password authentication returns a `tokenId` that applications can present as a cookie value for other operations that require authentication. The type of `tokenId` returned varies depending on whether client-based sessions are enabled in the realm to which the user authenticates:

• If client-based sessions are not enabled, the `tokenId` is an AM SSO token.

• If client-based sessions are enabled, the `tokenId` is an AM SSO token that includes an encoded AM session.

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*.

When authenticating with a user name and password, use HTTP POST to prevent the web container from logging the credentials. Pass the user name in an `X-OpenAM-Username` header, and the password in an `X-OpenAM-Password` header:

```
$ 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"
 \
--data "{}" \
https://openam.example.com:8443/openam/json/realms/root/authenticate
{
    "tokenId": "AQIC5w...NTcy*",
    "successUrl": "/openam/console",
    "realm":"/"
}
```

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 `dëmø`, 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"
 \
--data "{}" \
https://openam.example.com:8443/openam/json/realms/root/authenticate
{
    "tokenId": "AQIC5w...NTcy*",
    "successUrl": "/openam/console",
    "realm":"/"
}
```

This zero page login mechanism works only for name/password authentication. If you include a POST body with the request, it must be an empty JSON string as shown in the example. Alternatively, you can leave the POST body empty. Otherwise, AM interprets the body as a continuation of an existing authentication attempt, one that uses a supported callback mechanism.

The authentication service at `/json/authenticate` supports callback mechanisms that make it possible to perform other types of authentication in addition to simple user name/password login.

Callbacks that are not completed based on the content of the client HTTP request are returned in JSON as a response to the request. Each callback has an array of output suitable for displaying to the end user, and input which is what the client must complete and send back to AM. The default is still user name/password authentication:

```
$ 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": "...jwt-value...",
    "template": "",
    "stage": "DataStore1",
    "callbacks": [
        {
            "type": "NameCallback",
            "output": [
                {
                    "name": "prompt",
                    "value": " User Name: "
                }
            ],
            "input": [
                {
                    "name": "IDToken1",
                    "value": ""
                }
            ]
        },
        {
            "type": "PasswordCallback",
            "output": [
                {
                    "name": "prompt",
                    "value": " Password: "
                }
            ],
            "input": [
                {
                    "name": "IDToken2",
                    "value": ""
                }
            ]
        }
    ]
}
```

The `authID` value is a JSON Web Token (JWT) that uniquely identifies the authentication context to AM, and so must also be sent back with the requests.

To respond to the callback, send back the JSON object with the missing values filled, as in this case where the user name is `demo` and the password is `changeit`:

```
$ curl \
--request POST
 \
--header "Content-Type: application/json"
 \
--header "Accept-API-Version: resource=2.0, protocol=1.0"
 \
--data '{ "authId": "...jwt-value...", "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":"/"
}
```

The response is a token ID holding the SSO token value.

Alternatively, you can authenticate without requesting a session using the `noSession` query string parameter:

```
$ curl \
--request POST
 \
--header "Content-Type: application/json"
 \
--header "Accept-API-Version: resource=2.0, protocol=1.0"
 \
--data '{ "authId": "...jwt-value...", "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?noSession=true
{
    "message":"Authentication Successful",
    "successUrl":"/openam/console",
    "realm":"/"
}
```

AM can be configured to return a failure URL value when authentication fails. No failure URL is configured by default. The Default Failure Login URL can be set per realm; see "Post Authentication Processing" in the *Authentication and Single Sign-On Guide* for details. Alternatively, failure URLs can be configured per authentication chain, which your client can specify using the `service` parameter described below. On failure AM then returns HTTP status code 401 Unauthorized, and the JSON in the reply indicates the failure URL:

```
$ 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: badpassword" \
https://openam.example.com:8443/openam/json/realms/root/authenticate
{
    "code":401,
    "reason":"Unauthorized",
    "message":"Invalid Password!!",
    "failureUrl": "http://www.example.com/401.html"
}
```

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, to authenticate to a subrealm `customers` within the top-level realm, then the authentication endpoint URL is as follows: `https://openam.example.com:8443/openam/json/realms/root/realms/customers/authenticate`

The following additional parameters are supported:

You can use the `authIndexType` and `authIndexValue` query string parameters as a pair to provide additional information about how you are authenticating. The `authIndexType` can be one of the following types:

composite

Set the value to a composite advice string.

level

Set the value to the authentication level.

module

Set the value to the name of an authentication module.

resource

Set the value to a URL protected by an AM policy.

role

Set the value to an AM role.

service

> Set the value to the name of an authentication tree or authentication chain.

user

> Set the value to an AM user ID.

For example, to log into 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'
 \
--header 'X-OpenAM-Username: demo'
 \
--header 'X-OpenAM-Password: changeit' \
'http://openam.example.com:8080/openam/json/realms/root/authenticate?
authIndexType=service&authIndexValue=ldapService'
```

You can use the query string parameter, `sessionUpgradeSSOTokenId=tokenId`, to request session upgrade. Before the *tokenId* is searched for in the query string for session upgrade, the token is grabbed from the cookie. For an explanation of session upgrade, see "Session Upgrade" in the *Authentication and Single Sign-On Guide*.

AM uses the following callback types depending on the authentication module in use:

- `ChoiceCallback`: Used to display a list of choices and retrieve the selected choice.

- `ConfirmationCallback`: Used to ask for a confirmation such as Yes, No, or Cancel and retrieve the selection.

- `HiddenValueCallback`: Used to return form values that are not visually rendered to the end user.

- `HttpCallback`: Used for HTTP handshake negotiations.

- `LanguageCallback`: Used to retrieve the locale for localizing text presented to the end user.

- `NameCallback`: Used to retrieve a name string.

- `PasswordCallback`: Used to retrieve a password value.

- `PollingWaitCallback`: Used to restrict polling requests by specifying an amount of time to wait before responding.

- `RedirectCallback`: Used to redirect the client user-agent.

- `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.

- `TextInputCallback`: Used to retrieve text input from the end user.

- `TextOutputCallback`: Used to display a message to the end user.

- `X509CertificateCallback`: Used to retrieve the content of an x.509 certificate.

A.6.1. Logout

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"
}
```

A.6.2. logoutByHandle

To log out a session using a session handle, first perform an HTTP GET to the resource URL, `/json/sessions/`, using the `queryFilter` action to get the session handle:

```
$ 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%20%22demo
%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 a session using a session handle, perform an HTTP POST to the resource URL, `/json/sessions/`, using the `logoutByHandle` action.

```
$ 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
    }
}
```

A.6.3. 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*.

A.6.4. 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 `ZGVtbzpjaGFuZ2VpdA==`:

FORGEROCK

```
$ 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 ZGVtbzpjaGFuZ2VpdA=="
 \
--data "{}" \
https://openam.example.com:8443/openam/json/realms/root/authenticate
{
    "tokenId":"AQIC5w...NTcy*",
    "successUrl":"/openam/console",
    "realm":"/"
}
```

A.7. 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"
 \
--data "{}" \
https://openam.example.com:8443/openam/json/realms/root/authenticate
{
    "tokenId":"AQIC5wM...TU3OQ*",
    "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". 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.

A.8. 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,
            "refererWhitelist": [
                ""
            ],
            "allowedWithoutReferer": true
        },
        "realm": "/",
        "xuiUserSessionValidationEnabled": true,
        "FQDN": "openam.example.com"
}
```

A.9. Token Encoding

Valid tokens in AM requires 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:

```
AQIC5wM2LY4SfczntBbXvEAOuECbqMY3J4NW3byH6xwgkGE=@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:

```
AQIC5wM2LY4SfczntBbXvEAOuECbqMY3J4NW3byH6xwgkGE.*AAJTSQACMDE.*
```

A.10. Logging

AM 6 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.

A.10.1. 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.

A.10.2. 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.

A.11. Reference

This reference section covers return codes and system settings relating to REST API support in AM.

A.11.1. REST APIs

amster service name: `rest`

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 OpenAM 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 OpenAM 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.

B.1. The Scripting Environment

This section introduces how AM executes scripts, and covers thread pools and security configuration.

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 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.

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.
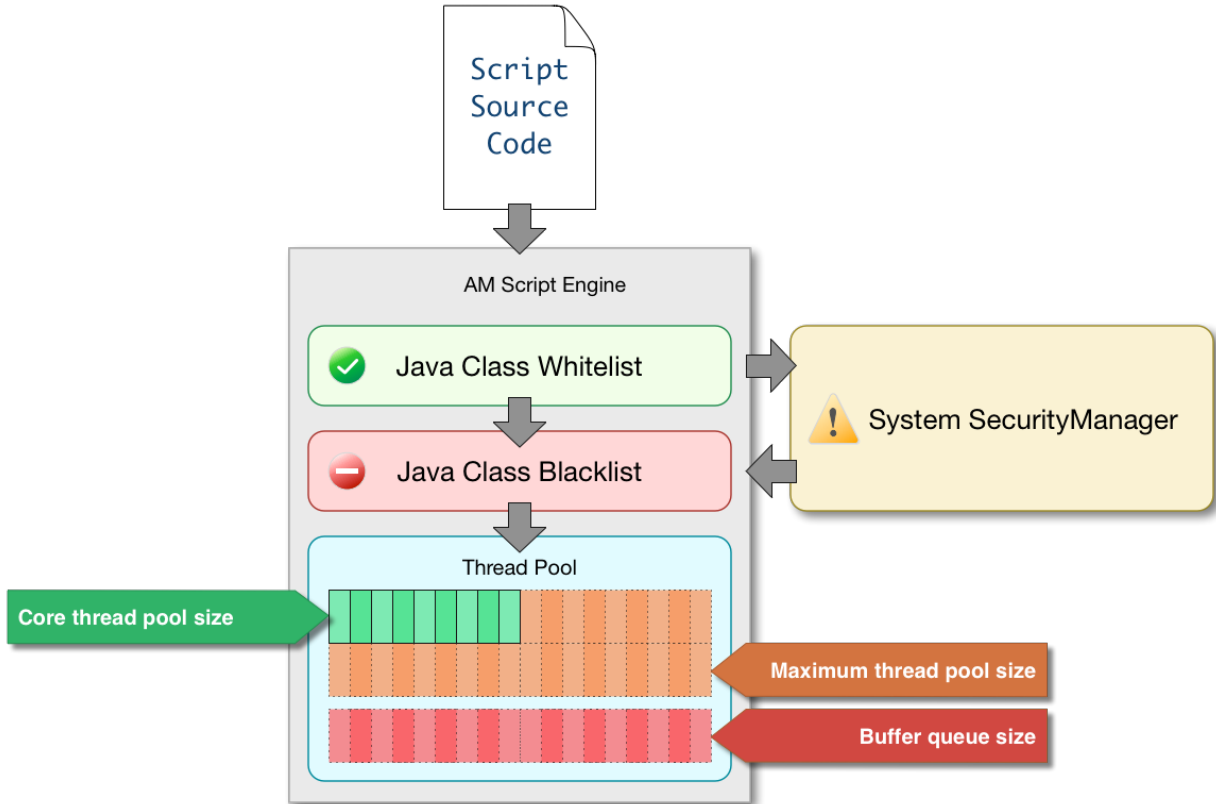
B.1.1. Security

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.

B.1.2. 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".

B.2. 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

B.2.1. 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
HttpClientResponse.getCookies	Void	Map<String, String>	Get the cookies for the returned response, if any exist.
HttpClientResponse.getEntity	Void	String	Get the entity of the returned response.
HttpClientResponse.getHeaders	Void	Map<String, String>	Get the headers for the returned response, if any exist.
HttpClientResponse .getReasonPhrase	Void	String	Get the reason phrase of the returned response.
HttpClientResponse.getStatusCode	Void	Integer	Get the status code of the returned response.
HttpClientResponse.hasCookies	Void	Boolean	Indicate whether the returned response had any cookies.

Method	Parameters	Return Type	Description
`HttpClientResponse.hasHeaders`	`Void`	`Boolean`	Indicate whether the returned response had any headers.

B.2.2. 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
`logger.error`	*Error Message* (type: `String`)	`Void`	Write *Error Message* to AM debug logs if ERROR level logging is enabled.
`logger.errorEnabled`	`Void`	`Boolean`	Return `true` when ERROR level debug messages are enabled.
`logger.message`	*Message* (type: `String`)	`Void`	Write *Message* to AM debug logs if MESSAGE level logging is enabled.
`logger.messageEnabled`	`Void`	`Boolean`	Return `true` when MESSAGE level debug messages are enabled.
`logger.warning`	*Warning Message* (type: `String`)	`Void`	Write *Warning Message* to AM debug logs if WARNING level logging is enabled.
`logger.warningEnabled`	`Void`	`Boolean`	Return `true` when WARNING level debug messages are enabled.

B.3. 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.

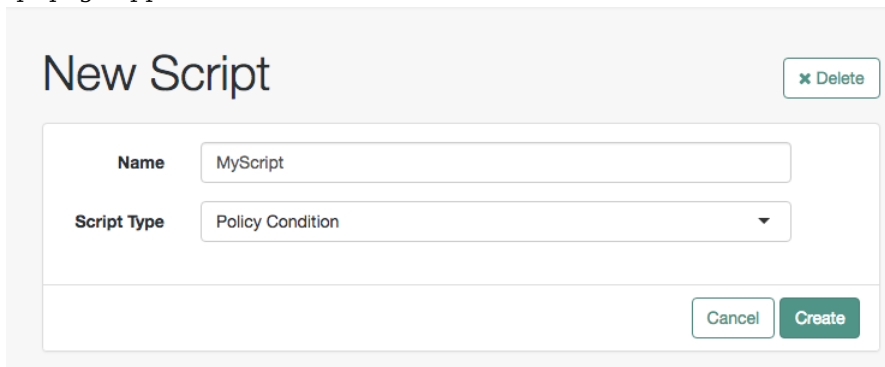## B.3.1. Managing Scripts With the AM Console

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:

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:

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.

B.3.2. 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": "dmFyIFFNUQVJUX1RJ...",
  "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:

```
dmFyIGEgPSAxMjM7IA0KdmFyIGIgPSA0NTY7
```

`language`

The language the script is written in - `JAVASCRIPT` or `GROOVY`.

Language Support per Context

Script Context	Supported Languages
`POLICY_CONDITION`	`JAVASCRIPT`, `GROOVY`
`AUTHENTICATION_SERVER_SIDE`	`JAVASCRIPT`, `GROOVY`
`AUTHENTICATION_CLIENT_SIDE`	`JAVASCRIPT`
`OIDC_CLAIMS`	`JAVASCRIPT`, `GROOVY`
`AUTHENTICATION_TREE_DECISION_NODE`	`JAVASCRIPT`, `GROOVY`

`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.

B.3.2.1. 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/realms/root/realms/myrealm/scripts?_queryFilter=true
{
    "result": [
        {
            "_id": "9de3eb62-f131-4fac-a294-7bd170fd4acb",
            "name": "Scripted Policy Condition",
```

```
            "description": "Default global script for Scripted Policy Conditions",
            "script": "LyoqCiAqIFRoaXMgXMg...",
            "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": "dmFyIFNUQVJUJUX1RJ...",
            "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
_id	Equals (eq), Contains (co), Starts with (sw)
name	Equals (eq), Contains (co), Starts with (sw)
description	Equals (eq), Contains (co), Starts with (sw)
script	Equals (eq), Contains (co), Starts with (sw)
language	Equals (eq), Contains (co), Starts with (sw)
context	Equals (eq), Contains (co), Starts with (sw)

B.3.2.2. 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": "LyoqCiAqIFRoaXMgXMg...",
    "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
}
```

B.3.2.3. 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 `GROOVY`, 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": "dmFyIGEgPSAxMjM7dmFyIGIgPSA0NTY7Cg==",
    "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": "dmFyIGEgPSAxMjM7dmFyIGIgPSA0NTY7ID1WQUxxJREFUSU9OIFNIT1VMRCBGQUlMPQo=",
    "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"
        }
    ]
}
```

B.3.2.4. 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
}
```

B.3.2.5. 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": "dmFyIGEgPSAxMjM7CnZhciBiiID0gNDU2Ow==",
    "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": "dmFyIGEgPSAxMjM7CnZhciBiiID0gNDU2Ow==",
    "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
}
```

B.3.2.6. 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
{}
```

B.3.3. 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
```

B.4. Scripting

amster service name: `scripting`

B.4.1. 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

Default value: `POLICY_CONDITION`

amster attribute: `defaultContext`

B.4.2. Secondary Configurations

This service has the following Secondary Configurations.

B.4.2.1. 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 OpenAM.

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. OpenAM 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). OpenAM 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. OpenAM 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, OpenAM creates new threads in the pool for script execution up to the configured maximum.

ssoadm attribute: `maxThreads`

Thread pool queue size

The number of threads to use for buffering script execution requests when the maximum thread pool size is reached.

ssoadm attribute: `queueSize`

Thread idle timeout (seconds)

Length of time (in seconds) for a thread to be idle before OpenAM 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, OpenAM 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

For more information or resources about AM and ForgeRock Support, see the following sections:

C.1. Accessing Documentation Online

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.

C.2. Using the ForgeRock.org Site

The ForgeRock.org site has links to source code for ForgeRock open source software, as well as links to the ForgeRock forums and technical blogs.

If you are a *ForgeRock customer*, raise a support ticket instead of using the forums. ForgeRock support professionals will get in touch to help you.

C.3. Getting Support and Contacting ForgeRock

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, visit https://www.forgerock.com, or send an email to ForgeRock at info@forgerock.com.

Glossary

Access control	Control to grant or to deny access to a resource.
Account lockout	The act of making an account temporarily or permanently inactive after successive authentication failures.
Actions	Defined as part of policies, these verbs indicate what authorized identities can do to resources.
Advice	In the context of a policy decision denying access, a hint to the policy enforcement point about remedial action to take that could result in a decision allowing access.
Agent administrator	User having privileges only to read and write agent profile configuration information, typically created to delegate agent profile creation to the user installing a web or Java agent.
Agent authenticator	Entity with read-only access to multiple agent profiles defined in the same realm; allows an agent to read web service profiles.
Application	In general terms, a service exposing protected resources.
	In the context of AM policies, the application is a template that constrains the policies that govern access to protected resources. An application can have zero or more policies.
Application type	Application types act as templates for creating policy applications.
	Application types define a preset list of actions and functional logic, such as policy lookup and resource comparator logic.

	Application types also define the internal normalization, indexing logic, and comparator logic for applications.
Attribute-based access control (ABAC)	Access control that is based on attributes of a user, such as how old a user is or whether the user is a paying customer.
Authentication	The act of confirming the identity of a principal.
Authentication chaining	A series of authentication modules configured together which a principal must negotiate as configured in order to authenticate successfully.
Authentication level	Positive integer associated with an authentication module, usually used to require success with more stringent authentication measures when requesting resources requiring special protection.
Authentication module	AM authentication unit that handles one way of obtaining and verifying credentials.
Authorization	The act of determining whether to grant or to deny a principal access to a resource.
Authorization Server	In OAuth 2.0, issues access tokens to the client after authenticating a resource owner and confirming that the owner authorizes the client to access the protected resource. AM can play this role in the OAuth 2.0 authorization framework.
Auto-federation	Arrangement to federate a principal's identity automatically based on a common attribute value shared across the principal's profiles at different providers.
Bulk federation	Batch job permanently federating user profiles between a service provider and an identity provider based on a list of matched user identifiers that exist on both providers.
Circle of trust	Group of providers, including at least one identity provider, who have agreed to trust each other to participate in a SAML v2.0 provider federation.
Client	In OAuth 2.0, requests protected web resources on behalf of the resource owner given the owner's authorization. AM can play this role in the OAuth 2.0 authorization framework.
Client-based sessions	AM sessions for which AM returns session state to the client after each request, and require it to be passed in with the subsequent request. For browser-based clients, AM sets a cookie in the browser that contains the session information.

For browser-based clients, AM sets a cookie in the browser that contains the session state. When the browser transmits the cookie back to AM, AM decodes the session state from the cookie.

Conditions	Defined as part of policies, these determine the circumstances under which which a policy applies.
	Environmental conditions reflect circumstances like the client IP address, time of day, how the subject authenticated, or the authentication level achieved.
	Subject conditions reflect characteristics of the subject like whether the subject authenticated, the identity of the subject, or claims in the subject's JWT.
Configuration datastore	LDAP directory service holding AM configuration data.
Cross-domain single sign-on (CDSSO)	AM capability allowing single sign-on across different DNS domains.
CTS-based 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 user data stores, and when different organizations use the same AM deployment.

Administrators can delegate realm administration. The administrator assigns administrative privileges to users, allowing them to perform administrative tasks within the realm.

Resource	Something a user can access over the network such as a web page. Defined as part of policies, these can include wildcards in order to match multiple actual resources.
Resource owner	In OAuth 2.0, entity who can authorize access to protected web resources, such as an end user.
Resource server	In OAuth 2.0, server hosting protected web resources, capable of handling access tokens to respond to requests for such resources.
Response attributes	Defined as part of policies, these allow AM to return additional information in the form of "attributes" with the response to a policy decision.
Role based access control (RBAC)	Access control that is based on whether a user has been granted a set of permissions (a role).
Security Assertion Markup Language (SAML)	Standard, XML-based language for exchanging authentication and authorization data between identity providers and service providers.
Service provider (SP)	Entity that consumes assertions about a principal (and provides a service that the principal is trying to access).
Authentication Session	The interval while the user or entity is authenticating to AM.
Session	The interval that starts after the user has authenticated and ends when the user logs out, or when their session is terminated. For browser-based clients, AM manages user sessions across one or more applications by setting a session cookie. See also CTS-based sessions and Client-based sessions.
Session high availability	Capability that lets any AM server in a clustered deployment access shared, persistent information about users' sessions from the CTS token store. The user does not need to log in again unless the entire deployment goes down.
Session token	Unique identifier issued by AM after successful authentication. For a CTS-based sessions, the session token is used to track a principal's session.

Single log out (SLO)	Capability allowing a principal to end a session once, thereby ending her session across multiple applications.
Single sign-on (SSO)	Capability allowing a principal to authenticate once and gain access to multiple applications without authenticating again.
Site	Group of AM servers configured the same way, accessed through a load balancer layer. The load balancer handles failover to provide service-level availability. The load balancer can also be used to protect AM services.
Standard metadata	Standard federation configuration information that you can share with other access management software.
Stateless Service	Stateless services do not store any data locally to the service. When the service requires data to perform any action, it requests it from a data store. For example, a stateless authentication service stores session state for logged-in users in a database. This way, any server in the deployment can recover the session from the database and service requests for any user. All AM services are stateless unless otherwise specified. See also Client-based sessions and CTS-based sessions.
Subject	Entity that requests access to a resource When an identity successfully authenticates, AM associates the identity with the Principal that distinguishes it from other identities. An identity can be associated with multiple principals.
User data store	Data storage service holding principals' profiles; underlying storage can be an LDAP directory service or a custom `IdRepo` implementation.
Web Agent	Native library installed in a web server that acts as a policy enforcement point with policies based on web page URLs.