FORGEROCK®

OpenID Connect 1.0 Guide
/ AM 5.0.0

Latest update: 5.0.0

Copyright © 2011-2017 ForgeRock AS.

Abstract

Guide showing you how to use ForgeRock Access Management with OpenID Connect 1.0.

Table of Contents

Preface

This guide covers concepts, configuration, and usage procedures for working with OpenID Connect 1.0 and AM.

This guide is written for anyone using OpenID Connect 1.0 with AM 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 OpenID Connect 1.0

This chapter covers OpenAM support for OpenID Connect 1.0.

OpenID Connect 1.0 is an authentication layer built on OAuth 2.0. OpenID Connect 1.0 is a specific implementation of OAuth 2.0 where the identity provider that runs the authorization server also holds the protected resource that the third-party application aims to access. This resource is the *UserInfo*, information about the authenticated end user expressed in a standard format. In this way, OpenID Connect 1.0 allows relying parties both to verify the identity of the end user and also to obtain user information using REST. This contrasts with OAuth 2.0, which only defines the authorization mechanism.

The names used in OpenID Connect 1.0 differ from those used in OAuth 2.0. In OpenID Connect 1.0, the key entities are the following:

- The *end user* (OAuth 2.0 resource owner) whose user information the application needs to access.

  The end user wants to use an application through existing identity provider account without signing up to and creating credentials for yet another web service.

- The *Relying Party* (RP) (OAuth 2.0 client) needs access to the end user's protected user information.

  For example, an online mail application needs to know which end user is accessing the application in order to present the correct inbox.

  As another example, an online shopping site needs to know which end user is accessing the site in order to present the right offerings, account, and shopping cart.

- The *OpenID Provider* (OP) (OAuth 2.0 authorization server and also resource server) that holds the user information and grants access.

  OpenAM can play this role in an OpenID Connect deployment.

  The OP effectively has the end user's consent to providing the RP with access to some of its user information. As OpenID Connect 1.0 defines unique identification for an account (subject identifier + issuer identifier), the RP can use this as a key to its own user profile.

  In the case of the online mail application, this key could be used to access the mailboxes and related account information. In the case of the online shopping site, this key could be used to access the offerings, account, shopping cart and so forth. The key makes it possible to serve users as if they had local accounts.

In OpenID Connect, the relying party can verify claims about the identity of the end user, and log the user out at the end of a session. OpenID Connect also makes it possible to discover the OpenID Provider for an end user, and to register relying party client applications dynamically. OpenID connect services are built on OAuth 2.0, JSON Web Token (JWT), WebFinger and Well-Known URIs.

In its role as OpenID Provider, OpenAM lets OpenID Connect relying parties (clients) discover its capabilities, handles both dynamic and static registration of OpenID Connect relying parties, responds to relying party requests with authorization codes, access tokens, and user information according to the Authorization Code and Implicit flows of OpenID Connect, and manages sessions.

This section describes how OpenAM fits into the OpenID Connect picture in terms of the roles that it plays in the authorization code and implicit flows, provider discovery, client registration, and session management.

1.1. OpenID Connect Scopes and Claims

This section explains how scopes and claims can be used when OpenAM is acting as an OpenID Connect provider.

When OpenAM is configured as an OAuth 2.0 provider, a scope is considered to be a concept, rather than directly relating to a piece of data in the user profile. For example, Facebook has an OAuth 2.0 scope named `read_stream`. OpenAM returns whether the scope is allowed or not, with no associated data.

When OpenAM is configured as an OpenID Connect provider, scopes can relate to data in a user profile by making use of one or more claims. Each claim maps directly to an attribute in the user profile.

For example, OpenAM supports a scope named `profile` when configured as an OpenID Connect provider, which by default is made up of the following claims:

OpenID Connect Scope Default Claim Mappings

Claim	User profile attribute
given_name	givenname
zoneinfo	preferredtimezone
family_name	sn
locale	preferredlocale
name	cn

The mappings between scopes, claims, and user profile attributes are controlled by the OIDC Claims Script specified in the OAuth 2.0 provider. For more information, see "Scripting OpenID Connect 1.0 Claims" and "OAuth2 Provider".

As each claim represents a piece of information from the user profile, OpenAM displays the actual data the relying party is given if the user clicks Allow:

OpenID Connect Consent Page

You can configure OpenAM to support requests for individual claims as query parameters, as described in section 5.5 of the OpenID Connect specification, by enabling the `claims_parameter_supported` option.

In section 5.6 of the specification, AM supports *Normal Claims*. The optional *Aggregated Claims* and *Distributed Claims* representations are not supported by AM.

For more information, see "OAuth2 Provider".

1.2. OpenID Connect Authorization Code Flow

The OpenID Connect Authorization Code Flow specifies how the relying party interacts with the OpenID Provider, in this case OpenAM, based on use of the OAuth 2.0 authorization grant. The following sequence diagram shows successful processing from the authorization request, through grant of the authorization code, access token, and ID token, and optional use of the access token to get information about the end user.

FORGEROCK

OpenID Connect Authorization Code Flow

OpenID Connect Authorization Code Flow

In addition to what OAuth 2.0 specifies, OpenID Connect uses an ID token so the relying party can validate claims about the end user. It also defines how to get user information, such as profile, email, address, and phone details from the UserInfo endpoint with a valid access token.

1.3. OpenID Connect Implicit Flow

The OpenID Connect Implicit Flow specifies how the relying party interacts with the OpenID Provider, in this case OpenAM, based on use of the OAuth 2.0 implicit grant. The following sequence diagram shows successful processing from the authorization request, through grant of the access and ID tokens, and optional use of the access token to get information about the end user.

OpenID Connect Implicit Flow
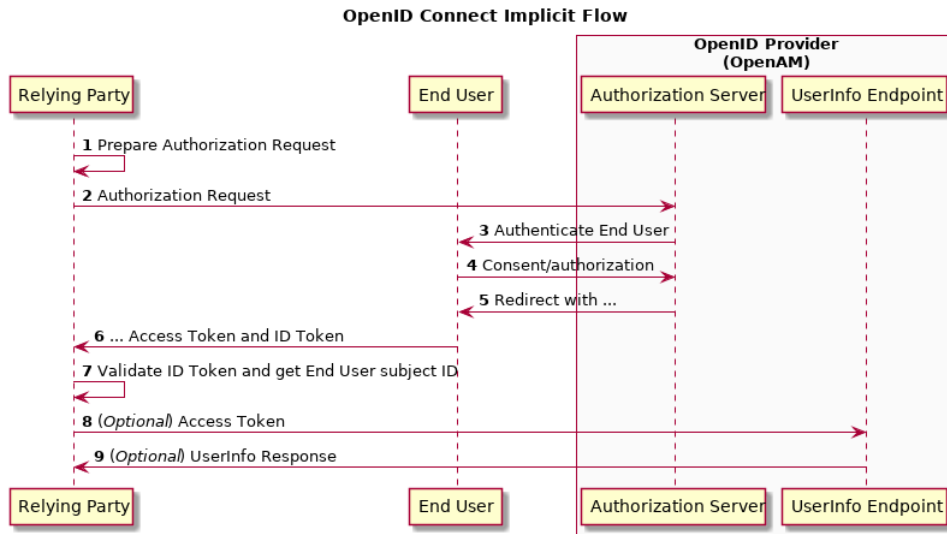
As for the Authorization Code Flow, the Implicit Flow specifies an ID token so that the relying party can validate claims about the end user. It also defines how to get user information, such as profile, email, address, and phone details from the UserInfo endpoint with a valid access token.

1.4. OpenID Connect Discovery

OpenID Connect defines how a relying party can discover the OpenID Provider and corresponding OpenID Connect configuration for an end user. The discovery mechanism relies on WebFinger to get the information based on the end user's identifier. The server returns the information in JSON Resource Descriptor (JRD) format.

1.5. OpenID Connect Relying Party Registration

OpenID Connect relying parties register OAuth 2.0 client profiles with OpenAM. Relying parties can register with OpenAM as a provider both statically, as for other OAuth 2.0 clients, and also dynamically, as specified by OpenID Connect Discovery. To allow dynamic registration, you register an initial OAuth 2.0 client that other relying parties can use to get access tokens for registration.

You can also enable OpenID Connect relying parties to register dynamically without having to provide an access token. For details, see the documentation on the advanced server property, `org.forgerock`

`.openam.openidconnect.allow.open.dynamic.registration`, in "Advanced Properties" in the *Reference*. Take care to limit or throttle dynamic registration if you enable this capability on production systems.

1.6. OpenID Connect Session Management

OpenID Connect lets the relying party track whether the end user is logged in at the provider, and also initiate end user logout at the provider. The specification has the relying party monitor session state using an invisible iframe and communicate status using the HTML 5 postMessage API.

OpenAM currently supports draft 10 of the OpenID Connect Session Management 1.0 specification.

1.7. Security Considerations

OpenAM provides security mechanisms to ensure that OpenID Connect 1.0 ID tokens are properly protected against malicious attackers: TLS, digital signatures, and token encryption.

While designing a security mechanism, you can also take into account the points developed in the section on *Security Considerations* in the OpenID Connect Core 1.0 incorporating errata set 1 specification.

OpenID Connect 1.0 requires the protection of network messages with Transport Layer Security (TLS). For information about protecting traffic to and from the web container in which OpenAM runs, see "*Setting Up Keys and Keystores*" in the *Setup and Maintenance Guide*.

OpenAM supports digital signatures for OAuth 2.0 and OpenID Connect 1.0 tokens. To configure the signatures, see "Configuring Digital Signatures".

Chapter 2
Implementing OpenID Connect 1.0

This chapter covers implementing and configuring OpenAM support for OpenID Connect 1.0.

2.1. Configuring as an OpenID Connect Provider

You can configure OpenAM's OAuth 2.0 provider service to double as an OpenID Connect provider service.

To Set Up the OAuth 2.0 Provider Service for OpenID Connect

Follow the steps in this procedure to set up the OAuth2 provider service with OpenID Connect defaults by using the Configure OAuth Provider wizard:

When you create the service with the Configure OAuth Provider wizard, the wizard also creates a standard policy in the Top Level Realm (/) to protect the authorization endpoint. In this configuration, OpenAM serves the resources to protect, and no separate application is involved. OpenAM therefore acts both as the policy decision point and policy enforcement point that protects the OAuth 2.0 authorization endpoint used by OpenID Connect.

There is no requirement to use the wizard or to create the policy in the Top Level Realm. However, if you create the OAuth 2.0 provider service without the wizard, then you must set up the policy independently, if required. The policy must appear in a policy set of type `iPlanetAMWebAgentService`. 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" in the *OAuth 2.0 Guide*. For details on creating policies, see "*Implementing Authorization*" in the *Authorization Guide*.

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

2. On the Configure OAuth2/OpenID Connect Service page, select the Realm for the provider service.

3. (Optional) If necessary, adjust the lifetimes for authorization codes, 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 new refresh token when refreshing an access token.

6. (Optional) If you have a custom scope validator implementation, put it on the OpenAM classpath, for example `/path/to/tomcat/webapps/openam/WEB-INF/lib/`, and specify the class name in the Scope Implementation Class field. For an example, see "Customizing OAuth 2.0 Scope Handling" in the *OAuth 2.0 Guide*.

7. Click Create to save your changes.

   OpenAM creates an OAuth2 provider service, with OpenID Connect default parameter values, and a policy to protect the OAuth2 authorization endpoints.

   > **Warning**
   >
   > If an OAuth2 provider service already exists, it will be overwritten with the new OpenID Connect parameter values.

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

   For OpenID Connect providers you may want to configure the following settings:

   • The optional Remote JSON Web Key URL field allows you to set a URL to a JSON web key set with the public key(s) for the provider.

     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.

   • The Subject Types supported map allows you to support pairwise subject types as described in the OpenID Connect core specification section concerning Subject Identifier Types.

   • The ID Token Signing Algorithms supported list allows you to change the list of algorithms used to sign ID Tokens.

   • The Supported Claims list allows you to restrict the claims supported by OpenAM's userinfo endpoint.

     For more information, see "OpenID Connect Scopes and Claims".

   • The Alias of ID Token Signing Key alias allows you to set the key pair alias for the key used to sign ID Tokens when using a signing algorithm that involves asymmetric keys.

     For instructions on changing the key pair, see "To Change Default test Signing Key" in the *Setup and Maintenance Guide*.

   • The Allow Open Dynamic Client Registration checkbox enables relying parties to register without using an access token.

- The Generate Registration Access Tokens checkbox has OpenAM generate Registration Access Tokens for dynamic client registration when Allow Open Dynamic Client Registration is enabled. This allows the client to view and update its registration.

9. Click Save to complete the process.

If your provider is part of a GSMA Mobile Connect deployment, see "Configuring as an OP for Mobile Connect".

2.2. Configuring for OpenID Connect Discovery

In order to allow relying parties to discover the OpenID Connect Provider for an end user, OpenAM supports OpenID Connect Discovery 1.0. In addition to discovering the OpenID Provider for an end user, the relying party can also request the OpenID Provider configuration.

OpenAM exposes REST endpoints for discovering information about the provider configuration, and about the provider for a given end user.

The following REST endpoints are available:

- `/oauth2/.well-known/openid-configuration` allows clients to retrieve OpenID Provider configuration by HTTP GET as specified by OpenID Connect Discovery 1.0.

  When the OpenID Connect provider is configured in a subrealm, relying parties can get the configuration by passing in the full path to the realm in the URL. For example, if the OpenID Connect provider is configured in a subrealm named `subrealm1`, which is a child of the top-level realm, the URL would resemble the following: `https://openam.example.com:8443/openam/oauth2/realms/root/realms/subrealm1/.well-known/openid-configuration`.

- `/.well-known/webfinger` allows clients to retrieve the provider URL for an end user by HTTP GET as specified by OpenID Connect Discovery 1.0.

  This endpoint does not support specifying a realm in the path, and is always located after the deployment URI. For example, `https://openam.example.com:8443/openam/.well-known/webfinger`.

> **Note**
>
> OpenAM supports a provider service that allows the realm to have a configured option for obtaining the base URL (including protocol) for components that need to return a URL to the client. This service is used to provide the URL base that is used in the `.well-known` endpoints used in OpenID Connect 1.0 and UMA.
>
> For more information, see "Configuring the Base URL Source Service".

A relying party needs to be able to discover the OpenID Connect provider for an end user. In this case you should consider redirecting requests to URIs at the server root, such as `http://www.example.com/`

`.well-known/webfinger` and `http://www.example.com/.well-known/openid-configuration`, to these Well-Known URIs in OpenAM's space.

Discovery relies on WebFinger, a protocol to discover information about people and other entities using standard HTTP methods. WebFinger uses Well-Known URIs, which defines the path prefix `/.well-known/` for the URLs defined by OpenID Connect Discovery.

Unless you deploy OpenAM in the root context of a container listening on port 80 on the primary host for your domain, relying parties need to find the right *host:port/deployment-uri* combination to locate the well-known endpoints. Therefore you must manage the redirection to OpenAM. If you are using WebFinger for something else than OpenID Connect Discovery, then you probably also need proxy logic to route the requests.

OpenID Connect Discovery requires an OAuth 2.0 provider service to be configured within OpenAM. The service must have `openid` as a supported scope in order to use the `/oauth2/.well-known/openid-configuration` endpoint. For information on configuring an OAuth 2.0 provider service for OpenID Connect in OpenAM, see "Configuring as an OpenID Connect Provider".

To retrieve the OpenID Connect provider for an end user, the relying party needs the following:

`host`

> The server where the relying party can access the WebFinger service.
>
> Notice that this is a host name rather than a URL to the endpoint, which is why you might need to redirect relying parties appropriately as described above.

`resource`

> Identifies the end user that is the subject of the request.
>
> The relying party must percent-encode the resource value when using it in the query string of the request, so when using the `acct` URI scheme and the resource is `acct:user@example.com`, then the value to use is `acct%3Auser%40example.com`.

`rel`

> URI identifying the type of service whose location is requested.
>
> In this case `http://openid.net/specs/connect/1.0/issuer`, which is `http%3A%2F%2Fopenid.net%2Fspecs%2Fconnect%2F1.0%2Fissuer`.

If you have not set up the redirection to the root of the domain yet, you can test the endpoint for the demo user account (output lines folded to make them easier to read):

```
$ curl \
 "https://openam.example.com:8443/openam/.well-known/webfinger
\
?resource=acct%3Ademo%40example.com\
&rel=http%3A%2F%2Fopenid.net%2Fspecs%2Fconnect%2F1.0%2Fissuer"
{
  "subject": "acct:demo@example.com",
  "links": [
    {
      "rel": "http://openid.net/specs/connect/1.0/issuer",
      "href": "https://openam.example.com:8443/openam/oauth2"
    }
  ]
}
```

This shows that the OpenID Connect provider for the OpenAM demo user is indeed the OpenAM server.

The relying party can also discover the OpenID Connect provider configuration. If you have not set up the redirection to the root of the domain yet, you can test this with the following **curl** command:

```
$ curl https://openam.example.com:8443/openam/.well-known/openid-configuration
  {
    "response_types_supported": [
      "code token id_token",
      "code",
      "code id_token",
      "device_code",
      "id_token",
      "code token",
      "token",
      "token id_token"
    ],
    "claims_parameter_supported": false,
    "introspection_endpoint": "https://openam.example.com:8443/openam/oauth2/introspect",
    "end_session_endpoint": "https://openam.example.com:8443/openam/oauth2/connect/endSession",
    "version": "3.0",
    "check_session_iframe": "https://openam.example.com:8443/openam/oauth2/connect/checkSession",
    "scopes_supported": [
      "address",
      "phone",
      "openid",
      "profile",
      "email"
    ],
    "issuer": "https://openam.example.com:8443/openam/oauth2",
    "id_token_encryption_enc_values_supported": [
      "A256GCM",
      "A192GCM",
      "A128GCM",
      "A128CBC-HS256",
      "A192CBC-HS384",
      "A256CBC-HS512"
    ],
    "acr_values_supported": [],
```

```
    "authorization_endpoint": "https://openam.example.com:8443/openam/oauth2/authorize",
    "userinfo_endpoint": "https://openam.example.com:8443/openam/oauth2/userinfo",
    "claims_supported": [
      "zoneinfo",
      "address",
      "profile",
      "name",
      "phone_number",
      "given_name",
      "locale",
      "family_name",
      "email"
    ],
    "id_token_encryption_alg_values_supported": [
      "RSA-OAEP",
      "RSA-OAEP-256",
      "A128KW",
      "RSA1_5",
      "A256KW",
      "dir",
      "A192KW"
    ],
    "jwks_uri": "https://openam.example.com:8443/openam/oauth2/connect/jwk_uri",
    "subject_types_supported": [
      "public"
    ],
    "id_token_signing_alg_values_supported": [
      "ES384",
      "HS256",
      "HS512",
      "ES256",
      "RS256",
      "HS384",
      "ES512"
    ],
    "registration_endpoint": "https://openam.example.com:8443/openam/oauth2/connect/register",
    "token_endpoint_auth_methods_supported": [
      "client_secret_post",
      "private_key_jwt",
      "client_secret_basic"
    ],
    "token_endpoint": "https://openam.example.com:8443/openam/oauth2/access_token"
  }
```

When the OpenID Connect provider is configured in a subrealm, then relying parties can get the configuration by passing in the realm in the URL.

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 OpenID Connect provider is configured in a subrealm named `subrealm1` which is a child of the top-level realm, the URL would resemble the following: `https://openam.example.com:8443/oauth2/realms/root/realms/subrealm1/.well-known/openid-configuration`.

2.3. Configuring the Base URL Source Service

In many deployments, OpenAM determines the base URL of a provider using the incoming HTTP request. However, there are often cases when the base URL of a provider cannot be determined from the incoming request alone, especially if the provider is behind some proxying application. For example, if an OpenAM instance is part of a site where the external connection is over SSL but the request to the OpenAM instance is over plain HTTP, then OpenAM would have difficulty in reconstructing the base URL of the provider.

In these cases, OpenAM supports a provider service that allows a realm to have a configured option for obtaining the base URL including protocol for components that need to return a URL to the client.

To Configure the Base URL Source Service

1. Log in to the AM console as an administrative user, such as `amAdmin`, and then navigate to Realms > *Realm Name* > Services.

2. Click Add a Service, select Base URL Source, and then click Create.

3. For Base URL Source, select one of the following options:

Base URL Source Options

Option	Description
Extension class	Click the Extension class to return a base URL from a provided `HttpServletRequest` object. In the Extension class name field, enter `org.forgerock.openam.services.baseurl.BaseURLProvider`.
Fixed value	Click Fixed value to enter a specific base URL value. In the Fixed value base URL field, enter the base URL.
Forwarded header	Click Forwarded header to retrieve the base URL from the `Forwarded` header field in the HTTP request. The Forwarded HTTP header field is standardized and specified in *RFC 7239*.
Host/protocol from incoming request (default)	Click Host/protocol from incoming request to get the hostname, server name, and port from the HTTP request.
X-Forwarded-* headers	Click X-Forwarded-* headers to use non-standard header fields, such as `X-Forwarded-For`, `X-Forwarded-By`, and `X-Forwarded-Proto`.

4. In the Context path, enter the context path for the base URL. If provided, the base URL includes the deployment context path appended to the calculated URL. For example, `/openam`.

5. Click Finish to save your configuration.

2.4. Registering OpenID Connect Relying Parties

OpenID Connect relying parties can register with OpenAM both statically through the AM console for example, and also dynamically using OpenID Connect 1.0 Dynamic Registration.

To Register a Relying Party With the AM Console

Registering a relying party by using the AM console consists of first creating an OAuth 2.0 Client agent profile, and then editing the profile for the settings pertinent to OpenID Connect 1.0.

1. In the AM console, navigate to Realms > *Realm Name* > Applications > OAuth 2.0, select New, provide the client identifier and client password, and finally click Create to create the profile.

2. To edit the profile to match the relying party configuration, follow the hints in "OAuth 2.0 and OpenID Connect 1.0 Client Settings" .

   In order to read and edit the relying party profile dynamically later without using The AM console, be sure to set an access token in the Access Token field.

To Register a Relying Party Dynamically

For dynamic registration you need the relying party profile data, and an access token to write the configuration to OpenAM by HTTP POST. To obtain the access token, register an initial client statically after creating the provider, as described in "To Register a Relying Party With the AM Console". Relying parties can then use that client to obtain the access token needed to perform dynamic registration.

> **Tip**
>
> As described in "OpenID Connect Relying Party Registration", you can allow relying parties to register without having an access token by setting the advanced server property, `org.forgerock.openam.openidconnect.allow` `.open.dynamic.registration`, to `true`. When using that setting in production systems, take care to limit or throttle dynamic registration.

On successful registration, OpenAM responds with information including an access token to allow the relying party subsequently to read and edit its profile.

1. Create an OAuth 2.0 provider service in the relevant realm, by following the steps in "To Set Up the OAuth 2.0 Provider Service for OpenID Connect".

2. Register an initial OAuth 2.0 client statically with a client ID, such as `masterClient` and client secret like `password`.

   Add at least one scope to the list of supported scopes, for example `cn`.

3. Obtain an access token using the client you registered.

For example, if you created the client as described in the previous step, and OpenAM administrator `amadmin` has password `password`, you can use the OAuth 2.0 resource owner password grant as in the following example:

```
$ curl \
 --request POST \
 --user "masterClient:password" \
 --data "grant_type=password&username=amadmin&password=password&scope=cn" \
 https://openam.example.com:8443/openam/oauth2/realms/root/access_token
{
    "expires_in": 59,
    "token_type": "Bearer",
    "refresh_token": "26938cd0-6870-4e31-ade9-df31afc37ee1",
    "access_token": "515d6551-4512-4279-98b6-c0ef3f03a722"
}
```

4. HTTP POST the relying party registration profile to the `/oauth2/connect/register` endpoint, using bearer token authorization with the access token you obtained from OpenAM.

   Ensure that you provide a `client_name` when registering the client. Without the `client_name` value the auto-generated `client_id` will be used on consent screens. The client ID is a UUID string and may not be desirable on end-user facing pages.

   For an example written in JavaScript, see the registration page in the OpenID Connect examples. Successful registration shows a response that includes the client ID and client secret. Lines are folded in the following example:

```
{
  "issued_at": 1392364349,
  "expires_at": 0,
  "client_secret": "7f446ca9-3f1f-48fb-bf8c-150b9e643f29",
  "client_name": "Example.com OpenID Connect Client",
  "redirect_uris": [
    "https://openam.example.com:8443/openid/cb-basic.html",
    "https://openam.example.com:8443/openid/cb-implicit.html"
  ],
  "registration_access_token": "515d6551-4512-4279-98b6-c0ef3f03a722",
  "client_id": "6e4abd50-3f03-41dc-b807-c6705c3e45d7",
  "registration_client_uri":
     "https://openam.example.com:8443/openam/oauth2/realms/root/connect/register
     ?client_id=6e4abd50-3f03-41dc-b807-c6705c3e45d7"
}
```

2.5. Managing OpenID Connect User Sessions

OpenID Connect Session Management 1.0 allows the relying party to manage OpenID Connect sessions, making it possible to know when the end user should be logged out.

Registered clients can use OpenID Connect Session Management 1.0 to handle end user logout actions.

- `/oauth2/connect/checkSession` allows clients to retrieve session status notifications.

- `/oauth2/connect/endSession` allows clients to terminate end user sessions.

As described in the OpenID Connect Session Management 1.0 - Draft 10 specification, OpenAM's OpenID Provider exposes both a `check_session_iframe` URL that allows the relying party to receive notifications when the end user's session state changes at the provider, and also an `end_session_endpoint` URL to which to redirect an end user for logout.

When registering your relying party that uses session management, you set the OAuth 2.0 client agent profile properties Post Logout Redirect URI and Client Session URI, described in "OAuth 2.0 and OpenID Connect 1.0 Client Settings". The Post Logout Redirect URI is used to redirect the end user user-agent after logout. The Client Session URI is the relying party URI where OpenAM sends notifications when the end user's session state changes.

2.6. Stateless OpenID Connect 1.0 Access and Refresh Tokens

OpenAM supports *stateless* access, refresh, and ID tokens for OpenID Connect 1.0 (OIDC). Stateless tokens allow clients to directly validate the tokens by storing session information within the token itself and bypassing storage in an external CTS data store. This feature also allows any OpenAM instance in the issuing cluster to validate an OIDC tokens without cross-server communication.

To Configure Stateless OpenID Connect 1.0 Access and Refresh

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. Enable Use Stateless Access & Refresh Tokens.

5. Enable Issue Refresh Tokens.

6. Enable Issue Refresh Tokens on Refreshing Access Tokens.

7. Generate some OIDC tokens using the REST API. Notice how each token is larger than a non-stateless example:

```
curl --request POST --user "MyClient:password" \
--data "grant_type=password&username=demo&password=changeit&scope=cn%20openid%20profile"\
http://openam.example.com:8080/openam/oauth2/realms/root/access_token
{
    "scope":"cn openid profile",
    "expires_in":5998,
    "token_type":"Bearer",
    "refresh_token":"eyAidHlwIjogIkpXVCIsICJhbGciOiAiSFMyNTYiIH0.eyAidG9rZW5OYW1l
```

```
    IjogInJlZnJlc2hfdG9rZW4iLCAic3ViIjogImRlbW8iLCAic2NvcGUiOiBbICJjbiIsICJvcGVu
    uaWQiLCAicHJvZmlsZSIgXSwgImF1dGhHcmFudElkIjogIjU2Y2VhYzM2LTZjNTItNGQ2NS05MT
    hiLTY4ZmY3MThiOTAzMyIsICJuYmYiOiAxNDY1NDE4OTc5LCAiaXNzIjogImh0dHA6Ly9vcGVuY
    W0uZXhhbXBsZS5jb206ODA4MC9vcGVuYW0vb2F1dGgyIiwgImV4cGlyZXNfaW4iOiA2MDAwMDAw
    LCAiaWF0IjogMTQ2NTQxODk3OSwgImV4cCI6IDE0NjU0Mjk5NzksICJhdWRpdFRyYWNraW5nSWQ
    iOiAiZGU4NjM4ZDUtMzhjNC00N2E1LWE5ODMtZDBjNDMzMTQyYTRhIiwgInJlYWxtIjogIi8iLC
    AiYXVkIjogIk15Q2xpZW50IiwgImp0aSI6ICJlNjY0OjgwZS03ZmY0LTRjRjMGEtOGVlZC01ZTViM
    2QwNGU4YWEiLCAidG9rZW5fdHlwZSI6ICJCZWFyZXIiIH0.VhXDFhI7K7BhouirMNgWQbeQvtrJ
    9IZg4MUH4bAOO3M",
  "id_token":"eyAidHlwIjogIkpXVCIsICJhbGciOiAiUlMyNTYiLCAia2lkIjogIlN5bExDNk5qd
    DFLR1FrdEQ5TXQrMHpjZVFFVVT0iIH0.eyAidG9rZW5OYW1lIjogImlkX3Rva2VuIiwgImF6cCI6
    ICJNeUNsaWVudCIsICJzdWIiOiAiZGVtbyIsICJhdF9oYXNoIjogIkNIb0VVzF1V3RUS1RMM1F
    rMUdMdnciLCAiaXNzIjogImh0dHA6Ly9vcGVuYW0uZXhhbXBsZS5jb206ODA4MC9vcGVuYW0vb2
    F1dGggyIiwgIm9yZy5mb3JnZXJvY2suc3ViIjY2ub3BlbWlrY29ubWJjdC5vcGVuYW0vb2
    S00NzBjBjLWFjMDQtMGMzNTM0NGRmYzNmIiwgImlhdCI6IDE0NjU0MTg5NzksICJhdХoX3RpbWUi
    OiAxNDY1NDE4OTc5LCAiZXhwIjogMTQ2NTQyNDk3OSwgInRva2VuVHlwZSI6ICJKV1RUb2tlbiI
    sICJyZWFsbSI6ICIvIiwiIiwgIm5hbWUiOiAiZGVtbyIsICJhdWQiOiAiTXlDbGllbnQiLCAiZmFtaW
    x5X25hbWUiOiAiZGVtbyIgfQ.RpWyfifklukI_YmNASbexM-tLUw4-RGlDouo8vAe5BTQbYdjAC
    HPDfngq0iFFVUVnJHhCIlJeo7GBn459lNR7boefgkaglTz2Q9wYo7TGX-B7ioV0qMnkYsZniTvx
    X2qQc5le_BJnp_2BJOfzzK83WnW93d9A4JGEAKCrfojrXI",
  "access_token":"eyAidHlwIjogIkpXVCIsICJhbGciOiAiSFMyNTYiIH0.eyAidG9rZW5OYW1lI
    jogImFjY2Vzc190b2tlbiIsICJzdWIiOiAiZGVtbyIsICJzY29wZSI6IFsgImNuIiwgIm9wZW5p
    ZCIsICJwcm9maWxlIiBdLCAiYXV0aEdyYW50SWQiOiAiNTZjZWFjMzYtNmM1Mi00ZDY1LTkxOGI
    tNjhmZjcxOGI5MDMzIiwgIm5iZiI6IDE0NjU0MTg5NzksICJpc3MiOiAiaHR0cDovL29wZW5hbS5
    leGFtcGxlLmNvbTo4MDgwL29wZW5hbS9vYXV0aDIiLCAiZXhwaXJlc19pbiI6IDYwMDAwMDAsI
    CJpYXQiOiAxNDY1NDE4OTc5LCAiZXhwIjogMTQ2NTQyNDk3OSwgImF1ZGl0VHJhY2tpbmdJZCI6
    ICI2ZTI2MzA4ZC05YzY2LTRkNjQtODE2Zi1iZTdmYTcyMDc2MTgiLCAicmVhbG0iOiAiLyIsICJ
    hdWQiOiAiTXlDbGllbnQiLCAianRpIjogImY4MDEwZjE2LWZiYTQtNDg1ZS04NGM1LWM2OGU2Mj
    k2ZjIxYyIsICJ0b2tlbl90eXBlIjogIkJlYXJlciIgfQ.JOAG50dLwfB6lKQr4fdKB1zRdKZyfY
    5bRRof61knJDs"
}
```

8. Decode the stateless access token to view its contents:

```
curl http://openam.example.com:8080/openam/oauth2/realms/root/tokeninfo?access_token=eyAid...1knJDs
{
    "tokenName":"access_token",
    "sub":"demo",
    "scope":["cn","openid","profile"],
    "iss":"http://openam.example.com:8080/openam/oauth2",
    "nbf":1465418979,
    "authGrantId":"56ceac36-6c52-4d65-918b-68ff718b9033",
    "expires_in":6000000,
    "iat":1465418979,
    "exp":1465424979,
    "auditTrackingId":"6e26308d-9c66-4d64-816f-be7fa7207618",
    "cn":"demo",
    "realm":"/",
    "aud":"MyClient",
    "openid":"",
    "jti":"f8010f16-fba4-485e-84c5-c68e6296f21c",
    "token_type":"Bearer",
    "access_token":"eyAid...1knJDss",
    "profile":""
}
```

2.6.1. Validating OpenID Connect 1.0 ID Tokens

Clients can use an OpenID Connect 1.0 endpoint on OpenAM to quickly validate a *stateless* OIDC ID token and optionally retrieve any claims within the token. The endpoint is used globally and not within a realm.

* `/openam/oauth2/idtokeninfo`

> **Note**
>
> The endpoint does not support the validation of encrypted OIDC ID tokens.

The endpoint validates an OIDC ID token based on rules 1-10 in section 3.1.3.7 of the OpenID Connect Core and runs the following steps:

1. Extracts the first `aud` (audience) claim from the ID token. The `client_id`, which is passed in as authentication of the request, will be used as the client and validated against the `aud` claim.

2. Extracts the `realm` claim, if present, default to the root realm if the token was not issued by OpenAM.

3. Looks up the client in the given realm, producing an error if it does not exist.

4. Verifies the signature of the ID token, according to the settings for the client (ID token signed response algorithm, public key selector).

5. Verifies the `issuer`, `audience`, `expiry`, `not-before`, and `issued-at` claims as per the specification.

To invoke the endpoint, the client sends an HTTP POST request to `/openam/oauth2/idtokeninfo` using the following parameters in the POST body in application/x-www-form-url-encoded format or as query parameters:

* `id_token` - OIDC ID token to validate (required)
* `claims` - optional comma-separated list of claims to return from the ID token

For example, you can run the following command:

```
$ curl -X POST -d "id_token=$IDTOKEN" \
  http://openam.example.com:8080/openam/oauth2/realms/root/idtokeninfo
```

where *$IDTOKEN* is an OIDC ID token.

If the ID token validates successfully, the endpoint unpacks the claims from the ID token and returns them as JSON. You can also use an optional `claims` parameter in the request to return those specific claims. If a claim is requested that does not exist, no error occurs; it will simply not be present in the response.

For example, you can run the following command to retrieve the claims in an OIDC ID token:

```
$ curl -i -X POST -d "id_token=$IDTOKEN" \
  'http://openam.example.com:8080/openam/oauth2/realms/root/idtokeninfo?claims=sub,exp,realm'

HTTP/1.1 200 OK
Date: Wed, 01 Jun 2016 07:31:39 GMT
Accept-Ranges: bytes
Server: Restlet-Framework/2.3.4
Vary: Accept-Charset, Accept-Encoding, Accept-Language, Accept
Content-Type: application/json;charset=UTF-8
Content-Length: 50
{
    "sub": "demo",
    "exp": 1461065147,
    "realm": "/"
}
```

For invalid requests, the endpoint returns a 400 HTTP code with a JSON error response:

```
$ curl -i -X POST 'http://openam.example.com:8080/openam/oauth2/realms/root/idtokeninfo?claims=sub,exp
,realm'

HTTP/1.1 400 Bad Request
Date: Wed, 01 Jun 2016 08:32:43 GMT
Accept-Ranges: bytes
Server: Restlet-Framework/2.3.4
Vary: Accept-Charset, Accept-Encoding, Accept-Language, Accept
Content-Type: application/json
Transfer-Encoding: chunked
Connection: close
{
    "error":"bad_request",
    "error_description":"no id_token in request"
}
```

2.7. Configuring for GSMA Mobile Connect

GSMA Mobile Connect is an application of OpenID Connect (OIDC). Mobile Connect builds on OIDC to facilitate use of mobile phones as authentication devices independently of the service provided and independently of the device used to consume the service. Mobile Connect thus offers a standard way for Mobile Network Operators to act as general-purpose identity providers, providing a range of levels of assurance and profile data to Mobile Connect-compliant Service Providers.

This section includes an overview, as well as the following:

• "Authorization Request Parameters"

• "ID Token Properties"

• "Configuring as an OP for Mobile Connect"

In a Mobile Connect deployment, OpenAM can play the OpenID Provider role, implementing the Mobile Connect Profile as part of the Service Provider - Identity Gateway interface.

OpenAM can also play the Authenticator role as part of the Identity Gateway - Authenticators interface. In this role, OpenAM serves to authenticate users at the appropriate Level of Assurance (LoA). In Mobile Connect, LoAs represent the authentication level achieved. A Service Provider can request LoAs without regard to the implementation, and the Identity Gateway includes a claim in the ID Token that indicates the LoA achieved.

In OpenAM, Mobile Connect LoAs map to an authentication mechanism. Service Providers acting as OpenID Relying Parties (RP) request an LoA by using the `acr_values` field in an OIDC authentication request. In OIDC, `acr_values` specifies Authentication Context Class Reference values. The RP sets `acr_values` as part of the OIDC Authentication Request. OpenAM returns the corresponding `acr` claim in the Authentication Response as the value of the ID Token `acr` field.

OpenAM as OP supports LoAs 1 (low - little or no confidence), 2 (medium - some confidence, as in single-factor authentication), and 3 (high - high confidence, as in multi-factor authentication), though out of the box it does not include support for 4, which involves digital signatures.

As Mobile Connect OP, OpenAM supports mandatory request parameters, and a number of optional request parameters:

Authorization Request Parameters

Request Parameter	Support	Description
`response_type`	Supported	OAuth 2.0 grant type to use. Set this to `code` for the authorization grant.
`client_id`	Supported	Set this to the client identifier.
`scope`	Supported	Space delimited OAuth 2.0 scope values. Required: `openid` Optional: `profile`, `email`, `address`, `phone`, `offline_access`
`redirect_uri`	Supported	OAuth 2.0 URI where the authorization request callback should go. Must match the `redirect_uri` in the client profile that you registered with OpenAM.
`state`	Supported	Value to maintain state between the request and the callback. Required for Mobile Connect.
`nonce`	Supported	String value to associate the client session with the ID Token. Optional in OIDC, but required for Mobile Connect.
`display`	Supported	String value to specify the user interface display.
`login_hint`	Supported	String value indicating the the ID to use for login. When provided as part of the OIDC Authentication Request, the `login_hint` is set as the value of a cookie named `oidcLoginHint`, which is an HttpOnly cookie (only sent over HTTPS). Authentication modules can then retrieve the cookie's value.

Request Parameter	Support	Description
`acr_values`	Supported	Authentication Context class Reference values used to communicate acceptable LoAs.
		When the OIDC relying party on the server provider supplies `acr_values` in the authorization request, OpenAM uses the OP configuration to map the values to authentication chains. It runs through the list of `acr_values` in order, attempting to use the first authentication chain that matches. OpenAM then returns the authentication chain used as the value of the ID token `acr` claims property. In this way the relying part on the service provider can determine the LoA achieved during authentication.
`dtbs`	Not supported	Data To Be Signed
		At present OpenAM does not support LoA 4.

As Mobile Connect OP, OpenAM responds to a successful authorization request with a response containing all the required fields, and also the optional `expires_in` field. OpenAM supports the mandatory ID Token properties, though the relying party is expected to use the `expires_in` value, rather than specifying `max_age` as a request parameter:

ID Token Properties

Request Parameter	Support	Description
`iss`	Supported	Issuer identifier
`sub`	Supported	Subject identifier
		By default OpenAM returns the identifier from the user profile.
`aud`	Supported	Audience, an array including the `client_id`.
`exp`	Supported	Expiration time in seconds since the epoch.
`iat`	Supported	Issued at time in seconds since the epoch.
`nonce`	Supported	The nonce supplied in the request.
`at_hash`	Supported.	Base64url-encoding of the SHA-256 hash of the "access_token" value.
`acr`	Supported	Authentication Context class Reference for the LoA achieved.
		For example, if the request specifies `acr_values=loa-3 loa-2` and OpenAM achieves LoA 2, then the ID token includes `"acr": "loa-2"`.
`amr`	Supported	Authentication Methods Reference to indicate the authentication method.

Request Parameter	Support	Description
		OpenAM maps these to authentication modules.
		Suggested values include the following: `OK, DEV_PIN, SIM_PIN, UID_PWD, BIOM, HDR, OTP`.
`azp`	Supported	Authorized party identifier, which is the `client_id`.

In addition to the standard OIDC user information returned with `userinfo`, OpenAM as OP for Mobile Connect returns the `updated_at` property, representing the time last updated as seconds since the epoch.

Configuring as an OP for Mobile Connect

You configure OpenAM as an OpenID Connect provider for Mobile Connect by changing the OAuth2 Provider configuration.

Follow the steps in this procedure to set up the OAuth2 provider service with Mobile Connect defaults by using the Configure OAuth Provider wizard.

When you create the OAuth2 provider service with the Configure OAuth Provider wizard, the wizard also creates a standard policy in the Top Level Realm (/) to protect the authorization endpoint. In this configuration, OpenAM serves the resources to protect, and no separate application is involved. OpenAM therefore acts both as the policy decision point and policy enforcement point that protects the OAuth 2.0 authorization endpoint used by OpenID Connect.

There is no requirement to use the wizard or to create the policy in the Top Level Realm. However, if you create the OAuth 2.0 provider service without the wizard, then you must set up the policy independently as well. The policy must appear in a policy set of type `iPlanetAMWebAgentService`. 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" in the *OAuth 2.0 Guide*. For details on creating policies, see "*Implementing Authorization*" in the *Authorization Guide*.

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

2. On the Configure Mobile Connect page, select the Realm for the provider service.

3. (Optional) If necessary, adjust the lifetimes for authorization codes, 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 have a custom scope validator implementation, put it on the OpenAM classpath, for example `/path/to/tomcat/webapps/openam/WEB-INF/lib/`, and specify the class name in the Scope

Implementation Class field. For an example, see "Customizing OAuth 2.0 Scope Handling" in the *OAuth 2.0 Guide*.

7. Click Create to save your changes.

    OpenAM creates an OAuth2 provider service with Mobile Connect default parameter values, as well as a policy to protect the OAuth2 authorization endpoints.

    > **Warning**
    >
    > If an OAuth2 provider service already exists, it will be overwritten with the new Mobile Connect parameter values.

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

    For Mobile Connect providers you may want to configure the following settings:

    a. For the OpenID Connect acr_values to Auth Chain Mapping, configure the mapping between `acr_values` in the authorization request and OpenAM authentication chains.

        For example, if the relying party request includes `acr_values=loa-3 loa-2` and the map includes `[loa-2]=ldapService`, and `[loa-3]=msisdnAndHotpChain`, then the authentication chain for the request is `msisdnPlusHotpChain`.

        The **ssoadm** attribute is `forgerock-oauth2-provider-loa-mapping`.

    b. For the OpenID Connect default acr claim, set the "acr" claim value to return in the ID Token when falling back to the default authentication chain.

        The **ssoadm** attribute is `forgerock-oauth2-provider-default-acr`.

    c. For the OpenID Connect id_token amr values to Auth Module mappings, set the "amr" values to return in the ID Token after successfully authenticating with specified authentication modules.

        For example, you could set `[UID_PWD]=LDAP` to return `"amr": [ "UID_PWD" ]` in the ID Token after authenticating with the LDAP module.

        The **ssoadm** attribute is `forgerock-oauth2-provider-amr-mappings`.

    d. Configure the identity Data Store attributes used to return `updated_at` values in the ID Token.

        For Mobile Connect clients, the user info endpoint returns `updated_at` values in the ID Token. If the user profile has never been updated `updated_at` reflects creation time.

        The `updated_at` values are read from the profile attributes you specify. When using OpenDJ directory server as an identity Data Store, the value is read from the `modifyTimestamp` attribute, or the `createTimestamp` attribute for a profile that has never been modified.

The **ssoadm** attribute for Modified Timestamp attribute name is `forgerock-oauth2-provider-modified-attribute-name`.

The **ssoadm** attribute is for Created Timestamp attribute name is `forgerock-oauth2-provider-created-attribute-name`.

In addition, you must also add these attributes to the list of LDAP User Attributes for the data store. Otherwise, the attributes are not returned when OpenAM reads the user profile. To edit the list in the AM console, browse to Realms > *Realm Name* > Data Stores > *Data Store Name* > LDAP User Attributes.

9. Click Save to complete the process.

A simple, non-secure GSMA Mobile Connect relying party example is available online.

2.8. Encrypting OpenID Connect ID Tokens

OpenAM supports the ability to encrypt OpenID Connect 1.0 ID tokens, which are Java Web Tokens (JWTs). OpenAM uses RSAES-PKCS1-v1_5, which is an encryption and decryption scheme in version 1.5 of PKCS #1, as the encryption algorithm for the ID token.

The supported encryption methods are A256CBC-HS512, which specifies the AES_256_CBC_HMAC_SHA_512 authenticated encryption algorithm (512-bit key), and A128CBC-HS256, which specifies the AES_128_CBC_HMAC_SHA_256 authenticated encryption algorithm (256-bit key).

To Configure OpenID Connect ID Token Encryption

1. Start the AM console, and select the realm that you are working with.

2. Navigate to Dashboard > Configure OAuth > Configure OpenID, and then select Create.

3. Navigate to Applications > OAuth 2.0.

4. Under Agent, select New, configure the Name and Password fields for the agent, and then select Create.

5. On the OAuth 2.0/OpenID Connect Client page, click the agent you just created, and add the `openid` scope.

6. Select the Enabled checkbox for Enable ID Token Encryption.

7. Run Java code to generate an encoded public client encryption key. An example snippet is presented below:

```
KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
keyPairGenerator.initialize(1024);
StringWriter writer = new StringWriter();
PEMWriter pemWriter = new PEMWriter(writer);
pemWriter.writeObject(keyPairGenerator.generateKeyPair().getPublic());
pemWriter.flush();
return writer.toString();
```

8. Copy and paste the encoded public client key generated in the previous step into the Client ID Token Public Encryption Key field. This encoded public key will be used to encrypt ID tokens.

9. Run through the authorization OpenID Connect code flow to generate the encrypted ID token. For more information, see "OpenID Connect Authorization Code Flow".

2.9. Configuring Digital Signatures

OpenAM supports digital signature algorithms that secure the integrity of its JSON payload, which is outlined in the JSON Web Algorithm specification (RFC 7518).

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
- 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. Start the AM console. Under Realms, select the realm that you are working with.

2. First, create or update your OAuth2 provider:

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

   b. Click Services > OAuth2 Provider.

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

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

i. 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.

ii. 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`.

iii. 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`.

iv. Click Save Changes.

3. 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

OpenAM 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 OpenID Connect 1.0

This chapter covers examples and usage of OpenAM with OpenID Connect 1.0.

3.1. Authorizing OpenID Connect 1.0 Relying Parties

Registered clients can request authorization through OpenAM.

OpenID Connect 1.0 supports both a Basic Client Profile using the OAuth 2.0 authorization code grant, and an Implicit Client Profile using the OAuth 2.0 implicit grant. These client profiles rely on the OAuth 2.0 endpoints for authorization. Those endpoints are described in "OAuth 2.0 Client and Resource Server Endpoints" in the *OAuth 2.0 Guide*.

OpenID Connect Authorization Code Flow and Implicit Flow define how clients interact with the provider to obtain end user authorization and profile information. Although you can run the simple example relying parties that are mentioned in this section without setting up Transport Layer Security, do not deploy relying parties in production without securing the transport.

Code for the relying party examples shown here is available online. Clone the example project to deploy it in the same web container as OpenAM. Edit the configuration at the outset of the `.js` files in the project, register a corresponding profile for the example relying party as described in "Registering OpenID Connect Relying Parties", and browse the deployment URL to see the initial page.

OpenID Connect Client Profiles Start Page
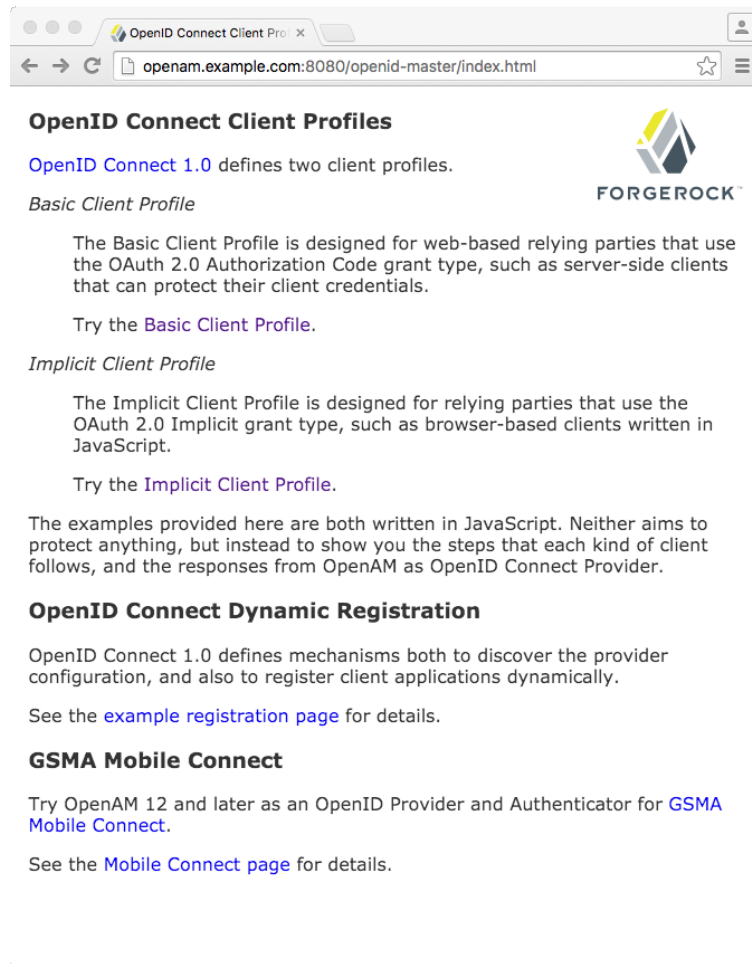
In addition, authorized clients can access end user information through the OpenID Connect 1.0 specific endpoint `/oauth2/userinfo`.

3.1.1. Authorization Code Flow Example

OpenID Connect Authorization Code Flow is designed for web-based relying parties that use the OAuth 2.0 Authorization Code grant type. This grant type makes it possible for the relying party to get the access code by using the authorization code directly, without passing through the end user's browser. To protect its client secret (password), part of the relying party must run on a server.

In the example, the Basic Client Profile Start Page describes the prerequisite configuration, which must be part of the relying party profile that is stored in the OpenAM realm where you set up the OpenID Provider. In the AM console, check that the OAuth 2.0 client profile matches the settings described.

OpenID Connect Basic Client Profile Start Page
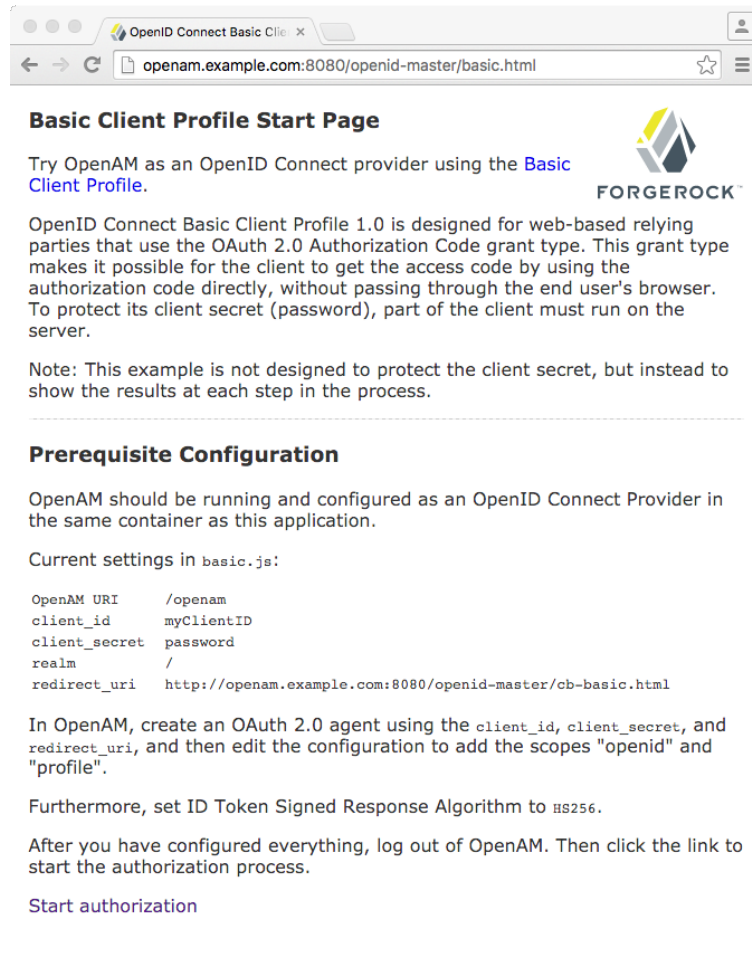
Log out of OpenAM, and click the link at the bottom of the page to request authorization. The link sends an HTTP GET request asking for `openid profile` scopes to the OpenID Provider authorization URI.

If everything is configured correctly, OpenAM's OpenID Provider has you authenticate as an end user, such as the demo user with username `demo` and password `changeit`, and grant (Allow) the relying party access to your profile.

If you successfully authenticate and allow the example relying party access to your profile, OpenAM returns an authorization code to the example relying party. The example relying party then uses the authorization code to request an access token and an ID token. It shows the response to that request. It also validates the ID token signature using the default (HS256) algorithm, and decodes the ID token to validate its content and show it in the output. Finally, it uses the access token to request information about the end user who authenticated, and displays the result.

OpenID Connect Basic Client Profile Response Page

Notice that in addition to the standard payload, the ID token indicates the end user's OpenAM realm, in this case `"realm": "/"`.

3.1.2. Implicit Flow Example

OpenID Connect Implicit Flow is designed for relying parties that use the OAuth 2.0 Implicit grant type. This grant type is designed for relying parties implemented in a browser. Rather than protect a client secret, the client profile must register a protected redirect URI in advance with the OpenID Provider.

In the example, the Implicit Client Profile Start Page describes the prerequisite configuration, which must be part of the relying party profile that is stored in the OpenAM realm where you set up the OpenID Provider. In the AM console, check that the OAuth 2.0 client profile matches the settings described. If you have already configured the agent profile for the Authorization Code Flow example, then you still need to add the redirect URI for the Implicit Flow.

OpenID Connect Implicit Client Profile Start Page

Log out of OpenAM, and click the link at the bottom of the page to request authorization. The link sends an HTTP GET request asking for `id_token token` response types and `openid profile` scopes to the OpenID Provider authorization URI.

If everything is configured correctly, OpenAM's OpenID Provider has you authenticate as an end user, such as the demo user with username `demo` and password `changeit`, and grant (Allow) the relying party access to your profile.

If you successfully authenticate and allow the example relying party access to your profile, OpenAM returns the access token and ID token directly in the fragment (after `#`) of the redirect URI. The relying party does not get an authorization code. The relying party shows the response to the request. It also validates the ID token signature using the default (HS256) algorithm, and decodes the ID

token to validate its content and show it in the output. Finally, the relying party uses the access token to request information about the end user who authenticated, and displays the result.

OpenID Connect Implicit Client Profile Response Page
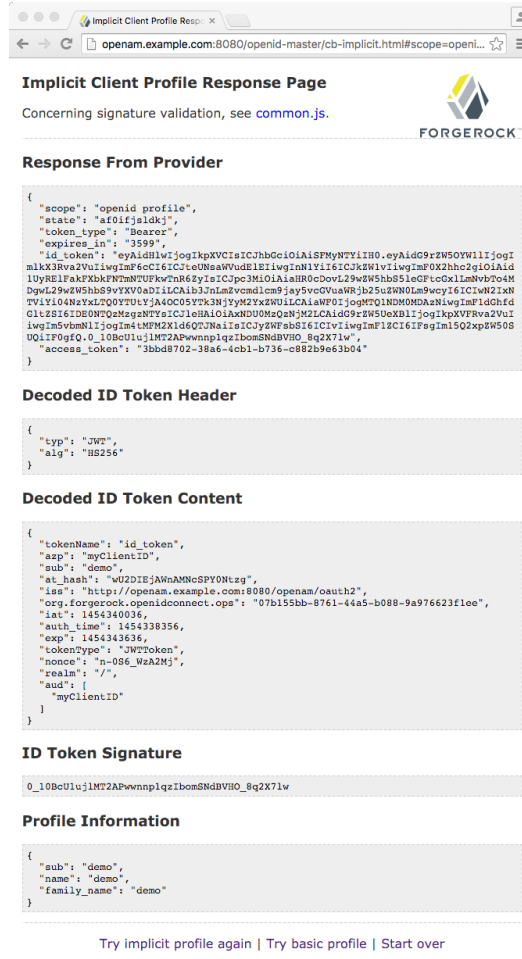
As for the Authorization Code Flow example, the ID Token indicates the end user's OpenAM realm and OpenAM token ID in addition to the standard information.

Chapter 4
Customizing OpenID Connect 1.0

This chapter covers customizing OpenAM's support for OpenID Connect 1.0.

4.1. Scripting OpenID Connect 1.0 Claims

This section demonstrates how to use the default OIDC claims script to return the profile attributes of a user in response to an OpenID Connect request for the `profile` scope.

The default OIDC claims script maps the following claims to the `profile` scope:

- zoneinfo

- family_name

- locale

- name

To examine the contents of the default OIDC claims script in the AM console browse to Realms > Top Level Realm > Scripts, and then click OIDC Claims Script.

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

For information about APIs available for use when scripting OpenID Connect 1.0 claims, see the following sections:

- "Global Scripting API Functionality"

- "OpenID Connect 1.0 Claims API Functionality"

4.1.1. Preparing

OpenAM requires a small amount of configuration before trying the example OIDC claims script. You must first create an OAuth2 provider with OpenID Connect settings, and register an OpenID Connect client, before you can authenticate to the client using a web browser.

The procedures in this section are:

- "To Create an OpenID Connect Provider Service"

- "To Register an OpenID Connect Client"

To Create an OpenID Connect Provider Service

Follow the steps in this procedure to create an OpenID Connect provider service by using the wizard.

1. Log in to OpenAM as an administrator, for example `amadmin`.

2. Click Realms > Top Level Realm > Configure OAuth Provider > Configure OpenID Connect.

3. On the Configure OpenID Connect page, accept the default values and then click Create.

4. Navigate to Realms > Top Level Realm > Services, click OAuth2 Provider, and verify that the value for OIDC Claims Script is the default script, `OIDC Claims Script`.

For a more detailed explanation and example of creating an OpenID Connect provider service, see "Configuring as an OpenID Connect Provider".

To Register an OpenID Connect Client

Follow the steps in this procedure to create an OpenID Connect client agent profile.

1. Log in to OpenAM as an administrator, for example `amadmin`.

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

3. In the Agent table, select New.

4. Enter a name for the client, such as `oidcTest`, provide a password, and then click Create.

5. On the OAuth 2.0/OpenID Connect Client page, click the agent name to configure the agent.

6. On the edit client page:

    a. In Redirection URIs, enter an example URI such as `http://www.example.com`.

    b. In Scope(s), enter both `profile` and `openid`.

    The `profile` scope will return details about the subject such as given name and timezone. The `openid` scope indicates this is an OpenID Connect client.

    c. In Display name, enter the name of the client as it will be displayed on the consent page, for example `OIDC Claims Script Client`.

7. Save your work.

For a more detailed explanation and examples of registering an OpenID Connect client, see "Registering OpenID Connect Relying Parties" and "OAuth 2.0 and OpenID Connect 1.0 Client Settings".

4.1.2. Trying the Default OIDC Claims Script

This section shows how to authenticate to a registered OpenID Connect client and request scopes from OpenAM, which in turn uses the default OIDC Claims script to populate the scope with claims and profile values.

To Authenticate to an OIDC Client and use the Default OIDC Claims Script
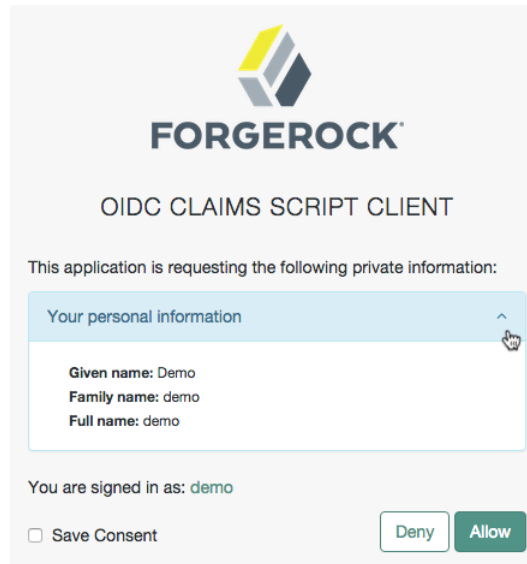
1. Log out of OpenAM.

2. In an Internet browser, navigate to the OpenAM OAuth 2.0 authorization endpoint, `/oauth2/authorize`, and specify the following query parameters, with the values you configured in the agent profile:

   Query parameters for OpenID Connect Authorization to an Agent Profile

Query Parameter	Agent Profile Property Value
`client_id`	Name of the agent, for example `oidcTest`.
`redirect_uri`	Redirection URIs, for example `http://www.example.com`.
`response_type`	Response Types, for example `code`.
`scope`	Scope(s), for example `openid profile`.

   For example: `http://openam.example.com:8080/openam/oauth2/realms/root/authorize?client_id=oidcTest&redirect_uri=http://www.example.com&response_type=code&scope=openid profile`

3. Log in to OpenAM as `demo`, with password `changeit`.

4. On the consent page, expand the panel labelled Your personal information to see the claim values the default OIDC script has populated into the requested `profile` scope.

5. Click Allow to be redirected to the Redirection URI specified in the agent profile. The authorization code is appended to the redirection URI as the value of the `code` query parameter.

Chapter 5
Reference

This reference section covers settings and other information relating to OpenID Connect 1.0 support in OpenAM.

5.1. OpenID Connect 1.0 Standards

OpenAM implements the following RFCs, Internet-Drafts, and standards relating to OpenID Connect 1.0:

OpenID Connect 1.0

OpenAM can be configured to play the role of OpenID provider. The OpenID Connect specifications depend on OAuth 2.0, JSON Web Token, Simple Web Discovery and related specifications. The following specifications make up OpenID Connect 1.0.

- OpenID Connect Core 1.0 defines core OpenID Connect 1.0 features.

  > **Note**
  >
  > In section 5.6 of the specification, AM supports *Normal Claims*. The optional *Aggregated Claims* and *Distributed Claims* representations are not supported by AM.

- OpenID Connect Discovery 1.0 defines how clients can dynamically recover information about OpenID providers.

- OpenID Connect Dynamic Client Registration 1.0 defines how clients can dynamically register with OpenID providers.

- OpenID Connect Session Management 1.0- Draft 10 describes how to manage OpenID Connect sessions, including logout.

- OAuth 2.0 Multiple Response Type Encoding Practices defines additional OAuth 2.0 response types used in OpenID Connect.

- OAuth 2.0 Form Post Response Mode defines how OpenID providers return OAuth 2.0 Authorization Response parameters in auto-submitting forms.

OpenID Connect 1.0 also provides implementer's guides for client developers.

- OpenID Connect Basic Client Implementer's Guide 1.0

• OpenID Connect Implicit Client Implementer's Guide 1.0

5.2. OpenID Connect 1.0 Claims API Functionality

This section covers functionality available when scripting OIDC claim handling using the OIDC claims script context type.

5.2.1. Accessing OpenID Connect Requests

Server-side scripts can access the OpenID Connect request through the following objects:

OIDC Request Objects

Object	Type	Description

scopes	Set<String>	Contains a set of the requested scopes. For example:
		
		``
		[
		    "profile",
		    "openid"
		]
		``

identity	Class	Contains a representation of the identity of the resource owner. For more details, see the com.sun.identity.idm.AMIdentity class in the OpenAM Javadoc.
session	Class	Contains a representation of the user's session object if the request contained a session cookie. For more details, see the com.iplanet.sso.SSOToken class in the OpenAM Javadoc.

claims	Map<String, Object>	Contains a map of the claims the server provides by default. For example:
		
		``
		{
		    "sub": "248289761001",
		    "updated_at": "1450368765"
		}
		``

requestedClaims	Map<String, Set<String>>	Contains requested claims if the claims query parameter is used in the request and *Enable "claims_parameter_supported"* is checked in the OAuth2 provider service configuration, otherwise is empty. For more information see "Requesting Claims using the "claims" Request Parameter" in the *OpenID Connect Core 1.0* specification.

Object	Type	Description

		Example:
		 ``{
		    "given_name": {
		        "essential": true,
		        "values": [
		            "Demo User",
		            "D User"
		        ]
		    },
		    "nickname": null,
		    "email": {
		        "essential": true
		    }
		}```

5.3. OAuth2 Provider

amster type ID: `oauth-oidc`

5.3.1. 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 data 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 data 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 data attribute: `blacklistPurgeDelay`

HMAC ID Token Authenticity Secret

A secret to use when signing a claim in HMAC-signed ID tokens so that authenticity can be assured when they are presented back to OpenAM.

Default value: `l6QZJe4O4be65x8TU7F2ihonPxCgimk5ekIOL+L50Zc=`

amster data 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 data attribute: `agentIdTokenSigningKeyAlias`

5.3.2. 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 data attribute: `statelessTokensEnabled`

Authorization Code Lifetime (seconds)

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

Default value: `120`

amster data 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`

FORGEROCK

amster data attribute: `refreshTokenLifetime`

Access Token Lifetime (seconds)

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

Default value: `3600`

amster data attribute: `accessTokenLifetime`

Issue Refresh Tokens

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

Default value: `true`

amster data attribute: `issueRefreshToken`

Issue Refresh Tokens on Refreshing Access Tokens

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

Default value: `true`

amster data attribute: `issueRefreshTokenOnRefreshedToken`

Saved Consent Attribute Name

Name of a case-sensitive and 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 data attribute: `savedConsentAttribute`

5.3.3. 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>`

amster data 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 data 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 data attribute: `responseTypeClasses`

FORGEROCK

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 data 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 data 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.

> **Important**
>
> Do not add the special token `am-introspect-all-tokens` to this list.
>
> Instead, add it to the list of scopes in the client profile of a client that should have access to introspect access tokens issued to other clients in the same realm.

amster data attribute: `supportedScopes`

Subject Types supported

List of subject types supported. Valid values are:

• `public` - Each client receives the same subject (`sub`) value.

FORGEROCK

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

Default value: `public`

amster data 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 data 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
ES256
ES384
ES512
```

Default value: `HS256`

amster data attribute: `tokenSigningAlgorithm`

Stateless Token Compression

Whether stateless access and refresh tokens should be compressed.

amster data attribute: `tokenCompressionEnabled`

Token Signing HMAC Shared Secret

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

Default value: `l6QZJe4O4be65x8TU7F2ihonPxCgimk5ekIOL+L50Zc=`

amster data 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 data 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 data attribute: `tokenSigningECDSAKeyAlias`

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 data 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 data attribute: `codeVerifierEnforced`

Modified Timestamp attribute name

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

amster data attribute: `modifiedTimestampAttribute`

Created Timestamp attribute name

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

amster data attribute: `createdTimestampAttribute`

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 data attribute: `clientsCanSkipConsent`

Enable auth module messages for Password Credentials Grant

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

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

Default value: `false`

amster data attribute: `moduleMessageEnabledInPasswordGrant`

5.3.4. 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 data 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:

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

amster data 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 data 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 data 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 data attribute: `supportedClaims`

OpenID Connect JWT Token Lifetime (seconds)

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

Default value: `3600`

amster data attribute: `jwtTokenLifetime`

5.3.5. 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 data 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 data 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 data attribute: `claimsParameterSupported`

FORGEROCK

Allow Open Dynamic Client Registration

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

Default value: `false`

amster data attribute: `allowDynamicRegistration`

Generate Registration Access Tokens

Whether to generate Registration Access Tokens for clients that register via 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 open dynamic client registration is disabled.

Default value: `true`

amster data attribute: `generateRegistrationAccessTokens`

OpenID Connect acr_values to Auth Chain Mapping

Maps OpenID Connect ACR values to authentication chains. See the acr_values parameter in the OpenID Connect authentication request specification for more details.

amster data attribute: `loaMapping`

OpenID Connect default acr claim

Default value to use as the `acr` claim in an OpenID Connect ID Token when using the default authentication chain.

amster data 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 data 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 data 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 data attribute: `storeOpsTokens`

Authorized OIDC SSO Clients

Specify a list of client names that are authorized to use OpenID Connect ID tokens as SSO Tokens.

Clients in this list can use ID tokens issued by AM to a user as if it were a full SSO token belonging to that user. For information on SSO tokens, see "About Sessions" in the *Authentication and Single Sign-On Guide*.

> **Important**
>
> Only add known trusted clients, as enabling this feature grants more authority than an ID Token normally provides.

Note that Java EE Agents 5 and Web Policy Agents 5 use OpenID Connect for communicating with AM. Agent profiles are automatically granted this privilege and do not need to be whitelisted.

amster attribute: `authorisedOpenIdConnectSSOClients`

5.3.6. 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 data 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 data attribute: `completionUrl`

Device Code Lifetime (seconds)

The lifetime of the device code, in seconds.

Default value: `300`

amster data attribute: `deviceCodeLifetime`

Device Polling Interval

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

Default value: `5`

amster data attribute: `devicePollInterval`

5.4. OAuth 2.0 and OpenID Connect 1.0 Client Settings

To register an OAuth 2.0 client with OpenAM 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 agent profile. After creating the agent profile, you can further configure the properties in the AM console by navigating to Realms > *Realm Name* > Applications > OAuth 2.0 > *Client Name*.

OAuth 2.0 and OpenID Connect 1.0 Client Configuration Fields

The following configuration fields are for OAuth 2.0 and OpenID Connect 1.0:

Group

Set this field if you have configured an OAuth 2.0 Client agent 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. OpenAM'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.

Scopes

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.

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.

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.

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.

Response Types

Specify the response type 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 OpenAM (as an authorization server) at the token endpoint. The authentication method applies to OIDC requests with scope `openid`. For more information, see the OpenID Connect Core 1.0 incorporating errata set 1.

- `client_secret_basic`. Clients authenticate with OpenAM (as an authorization server) using the HTTP Basic authentication scheme after receiving a `client_secret` value.

- `client_secret_post`. Clients authenticate with OpenAM (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.

Json Web Key URI

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

Json Web Key

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

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.

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.

Post Logout Redirect URIs

Specify the URI to which to redirect the user-agent after the client logout process.

Access Token

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

Client 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.

Client Name

Specify a human-readable name for the client.

Client JWT Bearer Public Key Certificate

Specify the public key certificate of the client's key pair that is used to sign JWTs issued by the client and used for client authentication or to request access tokens.

This is the base64-encoded X509 certificate containing the public key in PEM format, as in the following example.

```
-----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+D0V8GABFxUw9IaX6ajTqkWhldY77usvNeTD0Xc4R7OqSBrnA
SNCaUlJogWyzhbFlmE9Ne28j4RVpbz/EZn0oc/cHTJ6Lryzsivf4uDO1m3M3kM/MUyXc1Zv3rqBj
TeGSgcqEAd6XlGXY1+M/yIeouUTi0F1bk1rNlqJvd57Xb4CEq17tVbGBm0hkECM8
-----END CERTIFICATE-----
```

You can generate a key pair and export the certificate by using the Java **keytool** command.

```
$ keytool \
 -genkeypair \
 -keysize 2048 \
 -alias self-signed \
 -keyalg rsa \
 -dname "CN=jwt-bearer-client,O=openam.example.com" \
 -keystore keystore.jks \
 -keypass changeit \
 -storepass changeit \
 -validity 3650 \
 -v
Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA)
with a validity of 3,650 days
for: CN=jwt-bearer-client, O=openam.example.com
[Storing keystore.jks]

$ keytool \
 -list \
 -alias self-signed \
 -rfc \
 -keystore keystore.jks \
 -keypass changeit \
 -storepass changeit
Alias name: self-signed
Creation date: Oct 27, 2014
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
-----BEGIN CERTIFICATE-----
MIIDETCCAfmgAwIBAgIEU8SXLjANBgkqhkiG9w0BAQsFADA5MRswGQYDVQQKExJvcGVuYW0uZXhh
```

```
bXBsZS5jb20xGjAYBgNVBAMTEWp3dCliZWFyZXItY2xpZW50MB4XDTE0MTAyNzExNTY1NloXDTI0
MTAyNDExNTY1NlowOTEbMBkGA1UEChMSb3BlbmFtLmV4YW1wbGUuY29tMRowGAYDVQQDExFqd3Qt
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/
yIeouUTi0F1bk1rNlqJvd57Xb4CEq17tVbGBm0hkECM8
-----END CERTIFICATE-----
```

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.

Public key selector

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

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: 6000

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: 6000

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: 6000

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: 6000

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.

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`

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`

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.

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`

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.

Appendix A. About Scripting

You can use scripts for client-side and server-side authentication, policy conditions, and handling OpenID Connect claims.

A.1. The Scripting Environment

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

You can use scripts to modify default OpenAM 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.

OpenAM implements a configurable scripting engine for each of the context types that are executed on the server.

The scripting engines in OpenAM have two main components: security settings, and the thread pool.
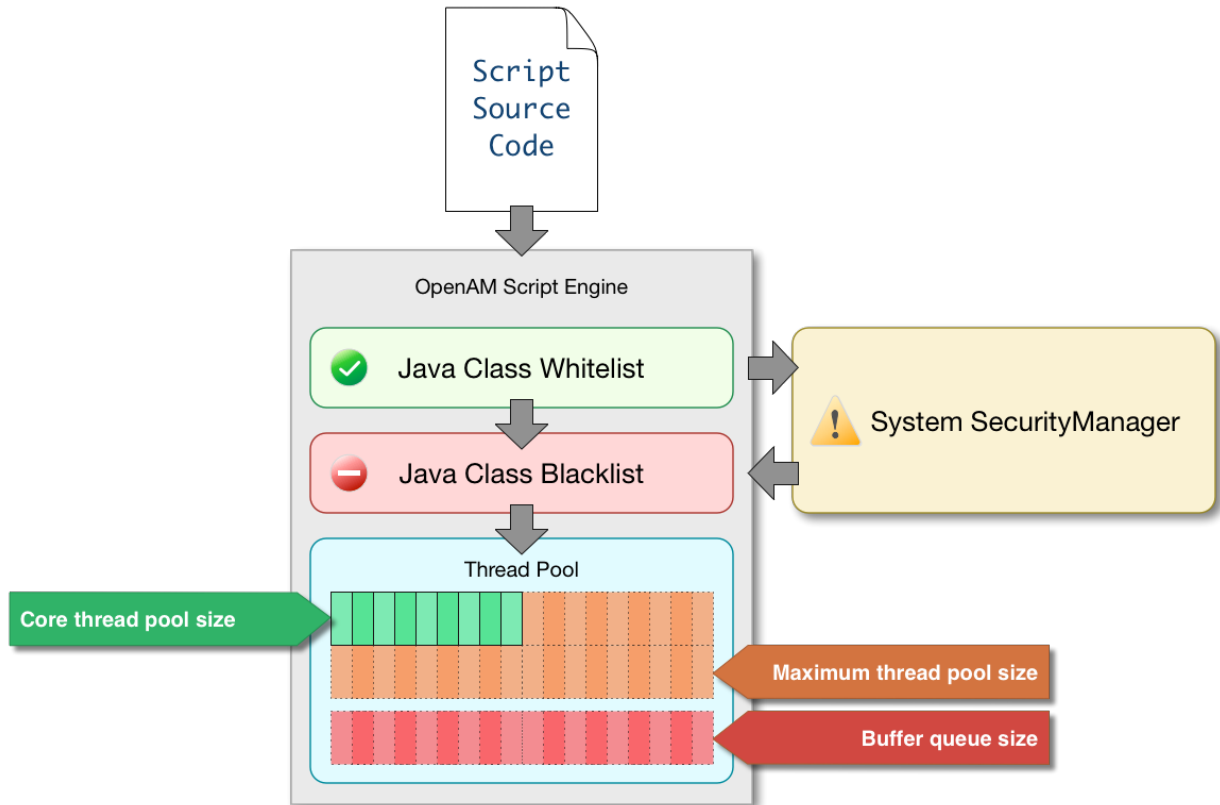
A.1.1. Security

OpenAM 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.

A.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, OpenAM 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, OpenAM terminates the thread, shrinking the pool.

For more information on configuring script engine thread pools, see "Scripting".

A.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

A.2.1. Accessing HTTP Services

OpenAM 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.

A.2.2. Debug Logging

Server-side scripts can write messages to OpenAM debug logs by using the `logger` object.

OpenAM does not log debug messages from scripts by default. You can configure OpenAM 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 OpenAM 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 OpenAM 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 OpenAM debug logs if WARNING level logging is enabled.
`logger.warningEnabled`	`Void`	`Boolean`	Return `true` when WARNING level debug messages are enabled.

A.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.

A.3.1. Managing Scripts With the AM Console
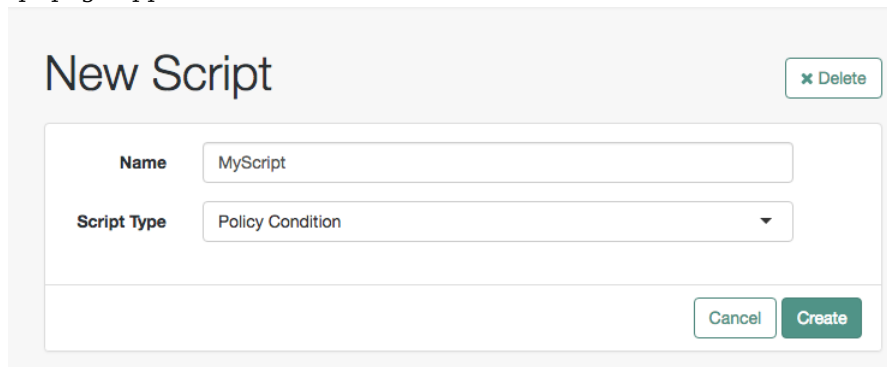
The following procedures describe how to create, modify, and delete scripts using the AM console:

- "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 OpenAM 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 OpenAM 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 OpenAM 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 OpenAM.

4. Click Delete.

A.3.2. Managing Scripts With the ssoadm Command

Use the **ssoadm** command's **create-sub-cfg**, **get-sub-cfg**, and **delete-sub-cfg** subcommands to manage OpenAM scripts.

Create an OpenAM script as follows:

1. Create a script configuration file as follows:

```
script-file=/path/to/script-file
language=JAVASCRIPT|GROOVY
name=myScript
context=AUTHENTICATION_SERVER_SIDE|AUTHENTICATION_CLIENT_SIDE|POLICY_CONDITION|OIDC_CLAIMS
```

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 myScript \
 --datafile /path/to/myScriptConfigurationFile
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/myScript
createdBy=
lastModifiedDate=
lastModifiedBy=
name=myScript
context=POLICY_CONDITION
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/myScript
Sub Configuration scriptConfigurations/myScript was deleted from realm /myRealm
```

A.3.3. 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.

OpenAM 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.

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": "dmFyIFNUQVJUX1RJ...",
  "language": "JAVASCRIPT",
  "context": "AUTHENTICATION_SERVER_SIDE",
  "createdBy": "id=dsameuser,ou=user,dc=openam,dc=forgerock,dc=org",
  "creationDate": 1433147666269,
  "lastModifiedBy": "id=dsameuser,ou=user,dc=openam,dc=forgerock,dc=org",
  "lastModifiedDate": 1433147666269
}
```

The values for the fields shown in the example above are explained below:

_id

The UUID that OpenAM 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:

dmFyIGEgPSAxMjM7IIA0KdmFyIGIgPSA0NTY7

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

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

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.

A.3.4. 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, OpenAM 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..." \
  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": "LyoqCiAqIFRoaXMgaXMg...",
      "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": "dmFyIFNUQVJJ1RJ...",
      "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)

A.3.5. 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..." \
  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
}
```

A.3.6. 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..." \
  --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..." \
  --data '{
    "script": "dmFyIGEgPSAxMjM7dmFyIGIgPSA0NTY7ID1WQUxxJREFUSU90IFNIT1VMRCBGQUlMMPQo=",
    "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"
    }
  ]
}
```

A.3.7. 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, OpenAM 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..." \
  --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
}
```

A.3.8. 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, OpenAM 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" \
  --request PUT \
  --data '{
    "name": "MyUpdatedJavaScript",
    "script": "dmFyIGEgPSAxMjM7CnZhciBiID0gNDU2Ow==",
    "language": "JAVASCRIPT",
    "context": "POLICY_CONDITION",
    "description": "An updated example script configuration"
  }' \
  https://openam.example.com:8443/openam/json/realms/root/realms/myrealm/scripts/0168d494-015a-420f-ae5a
-6a2a5c1126af
{
    "_id": "0168d494-015a-420f-ae5a-6a2a5c1126af",
    "name": "MyUpdatedJavaScript",
    "description": "An updated example script configuration",
    "script": "dmFyIGEgPSAxMjM7CnZhciBiID0gNDU2Ow==",
    "language": "JAVASCRIPT",
    "context": "POLICY_CONDITION",
    "createdBy": "id=amadmin,ou=user,dc=openam,dc=forgerock,dc=org",
    "creationDate": 1436807766258,
    "lastModifiedBy": "id=amadmin,ou=user,dc=openam,dc=forgerock,dc=org",
    "lastModifiedDate": 1436808364681
}
```

A.3.9. 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, OpenAM 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..." \
  https://openam.example.com:8443/openam/json/realms/root/realms/myrealm/scripts/0168d494-015a-420f-ae5a
-6a2a5c1126af
{}
```

A.4. Scripting

amster type ID: `scripting`

A.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
AUTHENTICATION_SERVER_SIDE
AUTHENTICATION_CLIENT_SIDE
OIDC_CLAIMS
```

Default value: `POLICY_CONDITION`

amster data attribute: `defaultContext`

A.4.2. Secondary Configurations

This service has the following Secondary Configurations.

A.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.

amster data 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.

amster data 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.

amster data 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.

amster data 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.

amster data 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.

amster data 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.

amster data 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.

amster data 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.

amster data attribute: `useSecurityManager`

Scripting languages

Select the languages available for scripts on the chosen type. Either `GROOVY` or `JAVASCRIPT`.

amster data attribute: `languages`

Default Script

The source code that is presented as the default when creating a new script of this type.

amster data attribute: `defaultScript`

Appendix B. Getting Support

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

B.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.

B.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.

B.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 on ForgeRock's support offering, including support plans and service level agreements (SLAs), visit https://www.forgerock.com/support.

FORGEROCK

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 subjects 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 policy agent profile configuration information, typically created to delegate policy agent profile creation to the user installing a policy 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.
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.
Delegation	Granting users administrative privileges with AM.
Entitlement	Decision that defines which resource names can and cannot be accessed for a given subject 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 EE policy agent	Java web application installed in a web container that acts as a policy agent, 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	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 subjects 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).
Session	The interval that starts with the user authenticating through AM 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 Stateful session and Stateless session.
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 Stateful session, 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. Use sticky load balancing based on `amlbcookie` values to improve site performance. 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.
Stateful session	An AM session that resides in the Core Token Service's token store. Stateful sessions might also be cached in memory on one or more

AM servers. AM tracks stateful 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.

Stateless session

An AM session for which state information is encoded in AM and stored on the client. The information from the session is not retained in the CTS token store. For browser-based clients, AM sets a cookie in the browser that contains the session information.

Subject

Entity that requests access to a resource

When a subject successfully authenticates, AM associates the subject with the Principal that distinguishes it from other subjects. A subject can be associated with multiple principals.

User data store

Data storage service holding principals' profiles; underlying storage can be an LDAP directory service, a relational database, or a custom `IdRepo` implementation.

Web policy agent

Native library installed in a web server that acts as a policy agent with policies based on web page URLs.