OpenID Connect 1.0 Guide

/ ForgeRock Access Management 7.1.4

Latest update: 7.1.4

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

Abstract

Guide showing you how to use OpenID Connect 1.0 with ForgeRock® Access Management (AM). ForgeRock Access Management provides intelligent authentication, authorization, federation, and single sign-on functionality.

Table of Contents

Overview iv
1. AM as the OpenID Provider 1
    OpenID Connect Discovery 4
    Security Considerations 5
    About Token Storage Location 5
2. OpenID Provider Configuration 7
    Encrypting ID Tokens and Backchannel Logout Tokens 9
3. About Claims 12
    Scripting OpenID Connect 1.0 Claims 15
4. Dynamic Client Registration 19
    Dynamic Client Registration Management 29
5. OpenID Connect Client Authentication 35
6. OpenID Connect Grant Flows 36
    Authorization Code Grant 37
    Authorization Code Grant with PKCE 46
    Backchannel Request Grant 56
    Implicit Grant 65
    Hybrid Grant 72
7. Managing OpenID Connect User Sessions 81
    Checking the State of a Session, and Invalidating Sessions 81
    Informing Relying Parties that a Session has Expired 85
8. Adding Authentication Requirements to ID Tokens 94
    The Authentication Context Class Reference (acr) Claim 94
    The Authentication Method Reference (amr) Claim 100
9. GSMA Mobile Connect 104
10. Additional Use Cases for ID Tokens 107
    Using ID Tokens as Session Tokens 107
    Using ID Tokens as Subjects in Policy Decision 108
11. OpenID Connect 1.0 Endpoints 109
    /oauth2/userinfo 110
    /oauth2/idtokeninfo 111
    /oauth2/connect/checkSession 114
    /oauth2/connect/endSession 115
    /oauth2/register 116
    /.well-known/webfinger 117
    /oauth2/.well-known/openid-configuration 118
    /oauth2/connect/jwk_uri 119
    /oauth2/connect/rp/jwk_uri 126
Glossary 130

Overview

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

Quick Start

Configuration Configure the OAuth 2.0 authorization server to double as OpenID provider.	Dynamic Client Registration Discover how clients can register and manage their information dynamically.	OpenID Connect Flows Discover the OpenID Connect flows and how to implement them in AM.
Authentication Requirements Require users to satisfy different rules or conditions when authenticating during OpenID Connect flows.	Scopes and Claims Learn about OpenID claims, scope-derived claims, and how to request them to AM.	OpenID Connect Endpoints Learn about the endpoints that AM exposes when acting as an OpenID Connect Provider.

About ForgeRock Identity Platform™ Software

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

Chapter 1

AM as the OpenID Provider

In its role as OpenID provider, AM returns ID tokens to relying parties. Since OpenID Connect builds on top of OAuth 2.0, when AM is configured as an OpenID provider it can also return access and refresh tokens to the relying parties, if needed.

Tip

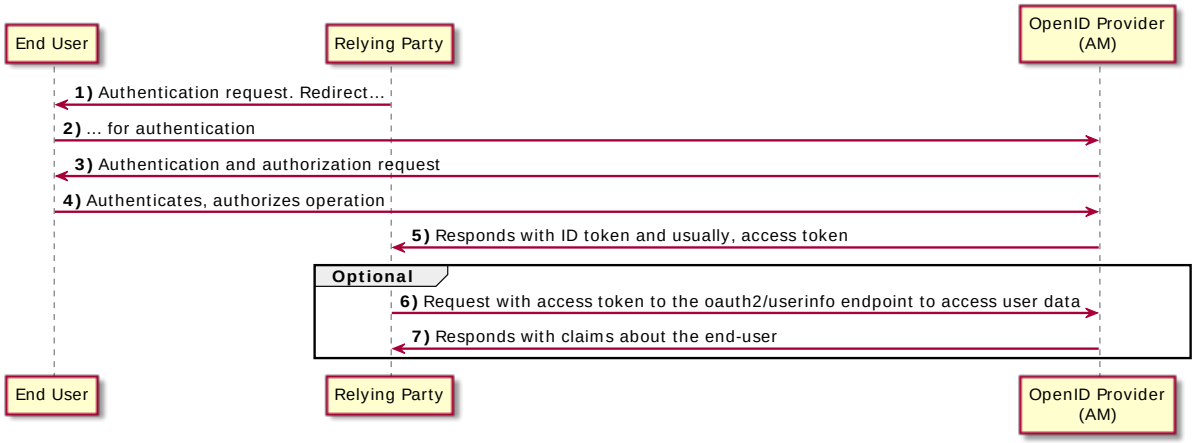
Before configuring OpenID Connect in your environment, ensure you are familiar with the OAuth 2.0 standards and AM's implementation of OAuth 2.0.

+ *OpenID Connect Concepts*

OpenID Connect 1.0 is an identity layer built on OAuth 2.0. It enables clients to verify the identity of users based on the authentication performed by OAuth 2.0 authorization servers, as well as to obtain profile information about the user using REST.

The following sequence diagram demonstrates the basic OpenID Connect flow:

OpenID Connect 1.0 Protocol Flow



OpenID Connect clients can register to the provider and manage their client data dynamically.

To let clients discover the end user's OpenID Connect provider, its endpoints, and how to interact with it, AM supports OpenID Connect Discovery 1.0.

+ *OAuth 2.0 or OpenID Connect?*

Both standards were created under the premise of users having the need to interact with a third party service, but aim to solve different problems:

OAuth 2.0 and OpenID Connect Comparison

	OAuth 2.0	**OpenID Connect**
Purpose	To provide users with a mechanism to **authorize** a service to access and use a subset of their data on their behalf, in a secure way. Users must agree to provide access under the service's term and conditions (for example, for how long the service has access to their data, and the purpose that data would be used for).	To provide users with a mechanism to **authenticate** to a service by providing it with a subset of their data in a secure way. Since OpenID Connect builds on top of OAuth 2.0, users authorize a relying party to collect a subset of their data (usually information stored in the end user's profile) from a third party. The service then uses this data to authenticate the user and provide its services. This way, the user can employ the relying party's services even if they have never created an account on it.
Use Cases	Use-cases are generic and can be tailored to many needs, but an example is a user allowing a photo print service access to a third-party server hosting their pictures, so the photo print service can print them.	The most common scenario is using social media credentials to log in to a third-party service provider.
Tokens	Access and refresh tokens	ID tokens
Regarding Scopes	Concept to limit the information to share with service or the actions the service can do with the data. For example, the `print` scope may allow a photo print service to access photos, but not to edit them. Scopes are not data, nor are related to user data in any way.	Concept that can be mapped to specific user data. For example, AM maps the `profile` scope to a series of user profile attributes. Since different identity managers may present the information in different attributes, the profile attributes are mapped to OpenID Connect *claims*. Claims are returned as part of the ID token. In some circumstances, additional claims can be requested in a call to the `oauth2/userinfo` endpoint. For more information about how AM maps user profile attributes to claims, see "*About Claims*".

Another difference between the standards is the name of the actors. The names of the actors in OpenID Connect 1.0 relate to those used in OAuth 2.0 as follows:

OAuth 2.0 and OpenID Connect Actors Comparison

OIDC Actor	OAuth 2.0 Actor	Description
End User	Resource Owner (RO)	The owner of the information the application needs to access. The end user that wants to use an application through existing identity provider account without signing up to and creating credentials for yet another web service.
Relying Party (RP)	Client	The third-party that needs to know the identity of the end user to provide their services. For example, a delivery company or a shopping site.
OpenID Provider (OP)	Authorization Server (AS) Resource Server (RS)	A service that has the end user's consent to provide 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 an online mail application, this key could be used to access the mailboxes and related account information. In the case of an 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. AM can act as the OpenID Connect provider to authenticate end users and provide RPs with information about the users in the form of an OpenID Connect ID token.

AM supports the following OpenID Connect grant types and standards:

Grant Types

- Authorization Code
- Authorization Code with PKCE
- Back Channel Request
- Implicit
- Hybrid
- Hybrid with PKCE

For more information, see "*OpenID Connect Grant Flows*".

Standards

- Session Management and Logout

  Relying parties can:

  - Track whether end users are logged in at the provider using an invisible iframe and the HTML 5 postMessage API.
  - Initiate end user logout at the provider using an endpoint.

  AM can also send *logout tokens* to relying parties when end user sessions linked to ID tokens become invalid.

  For more information, see "*Managing OpenID Connect User Sessions*".

- Discovery and Dynamic Client Registration

  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.

  For more information, see "OpenID Connect Discovery" and "*Dynamic Client Registration*".

- Mobile Connect

  Mobile Connect builds on top of OpenID Connect to facilitate the use of mobile phones as authentication devices, offering a way for mobile network operators to act as identity providers.

  For more information, see "*GSMA Mobile Connect*".

Tip

See the complete list of supported OpenID Connect in the *Reference* and OAuth 2.0 in the *Reference* standards.

OpenID Connect Discovery

In order to let relying parties (or clients) discover the OpenID Connect Provider for an end user, AM 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.

AM exposes the following REST endpoints for discovering the URL of the provider and its configuration:

- "/oauth2/.well-known/openid-configuration"
- "/.well-known/webfinger"

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.

Relying parties need to find the right *host:port/deployment-uri* combination to locate the well-known endpoints. You must manage the redirection to AM using your proxies, load balancers, and others, such that a request to `http://www.example.com/.well-known/webfinger` reaches, for example, `https://openam.example.com:8443/openam/.well-known/webfinger`.

Once the relying party has discovered the URL of the provider, it can register with it dynamically. For test purposes, or if it suits your environment better, you can also register them manually in the *OAuth 2.0 Guide*.

The `/.well-known/webfinger` endpoint is disabled by default. To enable it, perform the following steps:

1. In the AM console, go to Realms > *Realm Name* > Services > OAuth2 Provider > OpenID Connect.
2. Enable OIDC Provider Discovery.
3. Save your changes.

   The discovery endpoint now allows searches for users within this realm only. Repeat this procedure in as many realms as necessary.

Security Considerations

AM 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 AM runs, see "*Configuring Secrets, Certificates, and Keys*" in the *Security Guide*.

For additional security considerations related to the use of OAuth 2.0, see "Security Considerations" in the *OAuth 2.0 Guide*.

About Token Storage Location

AM OpenID Connect and OAuth 2.0-related services are stateless unless otherwise indicated; they do not hold any token information local to the AM instances.

Access and refresh tokens can be stored in the CTS token store or presented to the clients as JWTs. However, OpenID Connect tokens and session information are managed in the following way:

- ID tokens are always presented as JWTs.
- OpenID Connect sessions are always stored in the CTS token store.

For more information about how to configure access and refresh token storage, see "About Token Storage Location" in the *OAuth 2.0 Guide* in the OAuth 2.0 Guide.

Chapter 2

OpenID Provider Configuration

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

To configure the OAuth 2.0 provider, follow the steps in "To Configure the OAuth 2.0 Provider Service" in the *OAuth 2.0 Guide*. When you are finished, see "Additional Configuration" for OpenID Connect-specific configuration.

Additional Configuration

The OpenID provider is highly configurable:

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

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

OpenID Connect Configuration Options

Task	Resources
Configure the public keys for the provider OpenID providers sign ID tokens so that clients can ensure their authenticity. AM exposes the URI where clients can check the signing public keys to verify the ID token signatures. By default, AM exposes an internal endpoint with keys, but you can configure the URI of your secrets API instead.	"/oauth2/connect/jwk_uri"
Enable the OpenID Connect Discovery endpoint The discovery endpoint is disabled by default when you configure the OAuth 2.0 provider. Enable the endpoint if your clients need to discover the URL of the provider for a given user.	"OpenID Connect Discovery"
Configure pairwise subject types for dynamic registration To provide different values to the `sub` claim in the ID token for different clients (see Subject Identifier Types), configure the Subject Types Supported map.	N/A

Task	Resources
Also, ensure that the value of the Subject Identifier Hash Salt field is different from `changeme`.	
Configure whether AM must return scope-derived claims in the ID token Scope-derived claims, such as those returned when requesting the `profile` scope, are not returned in the ID token by default.	"*About Claims*"
Configure how AM maps scopes to claims and user profile attributes AM lets you map different user profile attributes to claims and scopes by using the AM scripting engine.	"*About Claims*"
Configure the OpenID provider for discovery and dynamic client registration/management AM supports several methods of dynamic registration that offer different security measures. You can also register the clients manually in the *OAuth 2.0 Guide*.	"OpenID Connect Discovery"
Add authentication requirements to ID tokens Require the end users to satisfy different authentication rules or conditions when authenticating to the OpenID provider, such as using a specific authentication tree.	"*Adding Authentication Requirements to ID Tokens*"
Configure AM as part of a GSMA Mobile Connect deployment Configure the OAuth 2.0 authorization server to double as a Mobile Connect provider.	"To Configure AM for Mobile Connect"
Configure the secret AM uses to sign ID tokens and logout tokens ID tokens and backchannel logout tokens are always signed. By default, AM uses a test secret to sign them; change it in production environments.	Secret ID Mappings for Signing OpenID Connect Tokens in the *Security Guide*
Configure the OpenID provider to encrypt ID tokens and logout tokens ID tokens and backchannel logout tokens are only signed by default. Consider encryption to protect them if they carry sensitive information about your end users.	"Encrypting ID Tokens and Backchannel Logout Tokens"
Configure the methods and algorithms available for signed or encrypted JWTs in the `request` parameter If your clients send request parameters to the authorization endpoint as a JWT instead of as HTTP parameters, configure the Request parameter signing algorithm field in the OpenID provider.	Secret ID Mappings for Decrypting OpenID Connect Request Parameters in the *Security Guide*

Task	Resources
Note that the alias mapped to the encryption algorithms are defined in the secret stores.	

Encrypting ID Tokens and Backchannel Logout Tokens

AM supports encrypting ID tokens and backchannel logout tokens to protect them against tampering attacks, which is outlined in the JSON Web Encryption specification (RFC 7516).

ID tokens and backchannel logout tokens share the same encryption configuration. In other words, you encrypt neither, or both.

To Configure ID Token and Backchannel Logout Token Encryption

Perform the following steps to enable and configure encryption:

1. Go to Realms > *Realm Name* > Applications > OAuth 2.0 > *Client Name*.
2. On the Signing and Encryption tab, select Enable ID Token Encryption.
3. In the Id Token Encryption Algorithm field, enter the algorithm AM will use to encrypt ID tokens and backchannel logout tokens:

   + *Supported Encryption Algorithms*

   - `A128KW` - AES Key Wrapping with 128-bit key derived from the client secret.
   - `A192KW` - AES Key Wrapping with 192-bit key derived from the client secret.
   - `A256KW` - AES Key Wrapping with 256-bit key derived from the client secret.
   - `RSA-OAEP` - RSA with Optimal Asymmetric Encryption Padding (OAEP) with SHA-1 and MGF-1.
   - `RSA-OAEP-256` - RSA with OAEP with SHA-256 and MGF-1.
   - `RSA1_5` - RSA with PKCS#1 v1.5 padding (not recommended).
   - `dir` - Direct encryption with AES using the hashed client secret.
   - `ECDH-ES` - Elliptic Curve Diffie-Hellman
   - `ECDH-ES+A128KW` - Elliptic Curve Diffie-Hellman + AES Key Wrapping with 128-bit key.
   - `ECDH-ES+A192KW` - Elliptic Curve Diffie-Hellman + AES Key Wrapping with 192-bit key.
   - `ECDH-ES+A256KW` - Elliptic Curve Diffie-Hellman + AES Key Wrapping with 256-bit key.
   - `X25519` - Elliptic Curve Diffie-Hellman with Curve25519.

- `X448` - Elliptic Curve Diffie-Hellman with Curve448.

Only the `P-256`, `P-384`, and `P-521` curves are supported.

4. In the ID Token Encryption Method field, enter the method AM will use to encrypt ID tokens and backchannel logout tokens:

   + *Supported Encryption Methods*

   - `A128CBC-HS256` - AES 128-bit in CBC mode using HMAC-SHA-256-128 hash (HS256 truncated to 128 bits)
   - `A192CBC-HS384` - AES 192-bit in CBC mode using HMAC-SHA-384-192 hash (HS384 truncated to 192 bits)
   - `A256CBC-HS512` - AES 256-bit in CBC mode using HMAC-SHA-512-256 hash (HS512 truncated to 256 bits)
   - `A128GCM` - AES 128-bit in GCM mode
   - `A192GCM` - AES 192-bit in GCM mode
   - `A256GCM` - AES 256-bit in GCM mode

5. (Optional) If you selected an RSA encryption algorithm, perform one of the following actions:
   - Enter the public key in the Client ID Token Public Encryption Key field.
   - Enter a JWK set in the Json Web Key field.
   - Enter a URI containing the public key in the Json Web Key URI field.
6. (Optional) If you selected an ECDH-ES encryption algorithm, perform one of the following actions:
   - Enter a JWK set in the Json Web Key field.
   - Enter a URI containing the public key in the Json Web Key URI field.
7. (Optional) If you selected an algorithm different from RSA or ECDH-ES, go to the Core tab and store the private key/secret in the Client Secret field.

Caution

Several features of OAuth 2.0 use the string stored in the Client Secret field to sign/encrypt tokens or parameters when you configure specific algorithms. For example, signing ID tokens with HMAC

algorithms, encrypting ID tokens with AES or direct algorithms, or encrypting OpenID Connect parameters with AES or direct algorithms.

In this case, these features must share the key/secret stored in the Client Secret field, and you must ensure that they are configured with the same algorithm.

Chapter 3

About Claims

OpenID Connect relies on claims to provide information about the end user to the relying parties.

+ *What Are Claims?*

A claim is a piece of information about the end user that the relying party or client can use to provide them a service.

Consider a page that lets the end user register using their Google account information instead of providing the information themselves. The page requests Google a set of claims about the end user, and uses the information on the claims to set up the account without user interaction.

If the end user agrees to share access to their claims, OpenID providers can return them in two ways: either as key pairs in the ID token, or by making them available at the `userinfo` endpoint. Part of implementing OpenID Connect in your environment is deciding which claims are safe to travel in the ID token, and which ones require the client to access the endpoint.

ID tokens contain additional claims that are not related to user information directly, but that are relevant to the flow, the relying party, or the authorization server. These are similar to those contained in access tokens; for example, `iss`, `aud`, `exp`, and others.

Read more:

- Section 2 of the OpenID Connect specification.
- Section 5 of the OpenID Connect specification

Note

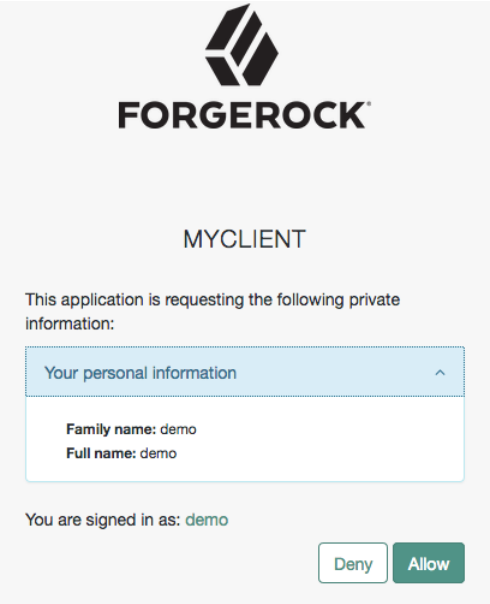
AM supports *Normal Claims*, as specified in section 5.6 of the specification. The optional *Aggregated Claims* and *Distributed Claims* representations are not supported by AM.

When AM is configured as an authorization server, a scope is a concept. For example, Facebook has an OAuth 2.0 scope named `read_stream`. AM returns allowed scopes in the access token, but it does not associate any data with them.

When AM is configured as an OpenID provider, scopes can relate to data in a user profile by making use of one or more claims.

As each claim represents a piece of information from the user profile, AM displays the actual data the relying party will receive if the end user consents to sharing it:

OpenID Connect Consent Page



AM maps scopes and profile data to claims using a script configured in the OAuth2 provider service. By default, the script maps several user profile attributes to the `profile` scope:

OpenID Connect Scope Default Claim Mappings

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

After a successful flow, the OpenID provider returns an ID token with the relevant claims. However, for security reasons, AM does not return scope-derived claims in the ID token by default.

+ *Requesting Claims in ID Tokens*

Sometimes you may need the provider to return scope-derived claims in the ID token. For example, when claims are related to authentication conditions or rules the end user needs to satisfy before being redirected to particular resources.

You can configure AM to either return all scope-derived claims in the ID token, or just the ones specified in the request:

- To configure the provider to always return scope-derived claims in the ID token, enable Always Return Claims in ID Tokens (Realms > *Realm Name* > Services > OAuth2 Provider > Advanced OpenID Connect).

  This option is disabled by default because of the security concerns of returning claims that may contain sensitive user information.

- To request that the provider only include certain scope-derived claims in the ID token, enable the property `Enable "claims_parameter_supported"` (Realms > *Realm Name* > Services > OAuth2 Provider > Advanced OpenID Connect) and request said claims in the `claims` parameter.

  + *Voluntary and Essential Claims in the claims Parameter*

  Claims specified using the `claims` parameter can be voluntary or essential:

  - **Essential**. The relying party specifies a number of claims that are necessary to ensure a good experience to the end user.

    For example, to provide personalized services, the relying party may require the end user's phone number to send them an SMS.

  - **Voluntary**. The relying party specifies a number of claims that are useful but not required to provide services to the end user.

  For an example on requesting voluntary and essential claims, see "Requesting acr Claims Example".

  Clients can still retrieve additional claims from the `/oauth2/userinfo` endpoint.

Tip

The OAuth 2.0 provider's Supported Claims field restricts the claims that can be granted in ID tokens, but not the claims a client can register with during dynamic client registration.

You can also use this field to configure how AM presents the claims in the AM consent screen. By default, scope-derived claims are not configured to display in the consent screen. You can either disable the consent pages in the *OAuth 2.0 Guide*, or manually configure the claims for display.

+ *How to Configure Claims in the AM Consent Screen*

Configure how claims appear in the consent screen by client or by realm (in the OAuth 2.0 provider service). For examples, see the Supported Claims field in the provider's "Advanced" in the *Reference* reference section or the Claim(s) field in Core Properties in the *OAuth 2.0 Guide*.

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`.

If the description is omitted, the claim is not displayed in the consent page. This may be useful when the client requires claims that are not meaningful for the end user.

Client-level configuration overrides that at provider level.

Scripting OpenID Connect 1.0 Claims

The script is configured in the OAuth 2.0 provider. To configure a different script of the type `OIDC Claims`, go to Realms > *Realm Name* > Services > OAuth 2.0 Provider > OpenID Connect, and then select it in the OIDC Claims Script drop-down menu.

To examine the contents of the default OIDC claims script, go to Realms > *Realm Name* > Scripts, and then select the OIDC Claims Script.

For more information about scripting in AM, see Getting Started with Scripting.

Tip

For examples of customizing the claims script, see:

- How do I add custom claims to the OIDC Claims Script in AM (All versions)?
- How do I override claims in the OIDC ID token in Identity Cloud or AM 7.1.x?
- How do I add a session property claim to the OIDC Claims Script in AM (All versions)?

OpenID Connect Claims Scripting API

The following properties are available to scripts:

`claims`

Contains a map of the claims the server provides by default. For example:

```
{
  "sub": "248289761001",
  "updated_at": "1450368765"
}
```

`clientProperties`

A map of properties configured in the relevant client profile. Only present if the client was correctly identified.

The keys in the map are as follows:

`clientId`

The URI of the client.

`allowedGrantTypes`

The list of the allowed grant types (`org.forgerock.oauth2.core.GrantType`) for the client.

`allowedResponseTypes`

The list of the allowed response types for the client.

`allowedScopes`

The list of the allowed scopes for the client.

`customProperties`

A map of any custom properties added to the client.

Lists or maps are included as sub-maps.

For example, a custom property of

`customMap[Key1]=Value1`

is returned as

`customMap` > `Key1` > `Value1`.

To add custom properties to a client, go to OAuth 2.0 > Clients > *Client ID* > Advanced, and update the Custom Properties field. The custom properties can be added in the format shown in these examples:

```
customproperty=custom-value1
customList[0]=customList-value-0
customList[1]=customList-value-1
customMap[key1]=customMap-value-1
customMap[key2]=customMap-value-2
```

From within the script, you can then access the custom properties in the following way:

```
var customProperties = clientProperties.get("customProperties");
var property = customProperties.get(PROPERTY_KEY);
```

`identity`

Contains a representation of the identity of the resource owner.

For more details, see the `com.sun.identity.idm.AMIdentity` class in the ForgeRock Access Management Javadoc.

`requestedClaims`

Contains requested claims if the `claims` query parameter is used in the request, and Enable "claims_parameter_supported" is checked in the OAuth 2.0 provider service configuration; otherwise, this property is empty.

For more information see Requesting Claims using the "claims" Request Parameter in the *OpenID Connect Core 1.0* specification.

Example:

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

`requestProperties`

A map of the properties present in the request. Always present.

The keys in the map are as follows:

`requestUri`

The URI of the request.

`realm`

The realm to which the request was made.

`requestParams`

The request parameters, and/or posted data. Each value in this map is a list of one, or more, properties.

> **Important**
>
> To mitigate the risk of reflection-type attacks, use OWASP best practices when handling these properties. For example, see Unsafe use of Reflection.

`scopes`

Contains a set of the requested scopes. For example:

```
[
  "profile",
  "openid"
]
```

`scriptName`

The display name of the script. Always present.

`session`

Contains a representation of the user's session object if the request contained a session cookie.

For more details, see the `com.iplanet.sso.SSOToken` class in the ForgeRock Access Management Javadoc.

Chapter 4

Dynamic Client Registration

AM supports dynamic registration, as defined in RFC 7591 (*OAuth 2.0 Dynamic Client Registration Protocol*), and in the *OpenID Connect Dynamic Client Registration 1.0* specification. The specifications describe how OAuth 2.0 and OpenID Connect clients can register:

+ *Client Registration Methods*

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

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

- Providing either a self-signed or a CA-signed X.509 certificate as authentication (OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens internet-draft) and its client metadata as JSON.

- By gaining authorization using an OAuth 2.0 access token, and with their client metadata.

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

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

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

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

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

  In AM, you store software publisher details as an agent profile. The software publisher profile identifies the issuer included in software statements, and holds information required to decrypt software statement JWTs and to verify their signatures. When the client presents a software statement as part of the dynamic registration data, AM uses the software publisher profile to determine whether it can trust the software statement.

> The protocol specification does not describe how the client obtains the software statement JWT. AM expects the software publisher to construct the JWT according to the settings in its agent profile. Note, however, that AM ignores keys specified in JWT headers, such as `jku` and `jwe`.

To Configure AM for Dynamic Client Registration

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

1. Configure an authorization service.

   For details, see "To Configure the OAuth 2.0 Provider Service" in the *OAuth 2.0 Guide*.

2. Navigate to Realms > *Realm* > Services > OAuth2 Provider.

3. On the Client Dynamic Registration tab, consider configuring the following settings:

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

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

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

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

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

   For additional details, see "Client Dynamic Registration" in the *Reference*.

4. (Optional) If the clients will authenticate using mTLS with CA-signed (PKI) certificates, configure AM to hold the certificates belonging to the certificate authorities you want the instance of AM to trust. For more information, see "Mutual TLS Using Public Key Infrastructure" in the *OAuth 2.0 Guide*.

5. (Optional) If you enabled Require Software Statement for Dynamic Client Registration, you must register a software publisher:

   a. In the AM console, go to *Realm* > Applications > OAuth 2.0 > Software Publisher, and add a new software publisher agent.

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

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

Important

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

6. (Optional) On the Advanced tab, in the Client Registration Scope Whitelist field, add the scopes clients can register with.

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

+ *Open Dynamic Client Registration*

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

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

```
$ curl \
 --request POST \
 --header "Content-Type: application/json" \
 --data '{
 "redirect_uris": ["https://client.example.com:8443/callback"],
 "client_name#en": "My Client",
 "client_name#ja-Jpan-JP": "\u30AF\u30E9\u30A4\u30A2\u30F3\u30C8\u540D",
 "client_uri": "https://client.example.com/"
 }' \
 "https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/register"
{
 "request_object_encryption_alg": "",
 "default_max_age": 1,
 "application_type": "web",
 "client_name#en": "My Client",
 "registration_client_uri": "https://openam.example.com:8443/openam/realms/root/realms/alpha/
oauth2/register?client_id=2aeff083-83d7-4ba1-ab16-444ced02b535",
 "client_type": "Confidential",
```

```
"userinfo_encrypted_response_alg": "",
"registration_access_token": "4637ee46-51df-4901-af39-fec5c3a1054c",
"client_id": "2aeff083-83d7-4ba1-ab16-444ced02b535",
"token_endpoint_auth_method": "client_secret_basic",
"userinfo_signed_response_alg": "",
"public_key_selector": "x509",
"authorization_code_lifetime": 0,
"client_secret": "6efb5636-6537-4573-b05c-6031cc54af27",
"user_info_response_format_selector": "JSON",
"id_token_signed_response_alg": "HS256",
"default_max_age_enabled": false,
"subject_type": "public",
"jwt_token_lifetime": 0,
"id_token_encryption_enabled": false,
"redirect_uris": ["https://client.example.com:8443/callback"],
"client_name#ja-jpan-jp": "#######",
"id_token_encrypted_response_alg": "RSA1_5",
"id_token_encrypted_response_enc": "A128CBC_HS256",
"client_secret_expires_at": 0,
"access_token_lifetime": 0,
"refresh_token_lifetime": 0,
"request_object_signing_alg": "",
"response_types": ["code"]
}
```

Tip

OpenID Connect clients must ensure that the following information is present in the JSON:

- The `openid` scope. For example, `"scopes": ["profile", "openid"]`.
- The `id_token` response type. For example, `"response_types": ["code", "id_token code"]`.

+ *Dynamic Client Registration with Mutual TLS Authentication*

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

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

To use CA-signed certificates (PKI), you must configure AM to hold the certificates belonging to the certificate authorities you want the instance of AM to trust. For more information, see "Mutual TLS Using Public Key Infrastructure" in the *OAuth 2.0 Guide*.

Include the following properties as part of the client metadata:

- **token_endpoint_auth_method**, which must be set to `tls_client_auth`.
- **tls_client_auth_subject_dn**, which must be set to the distinguished name as it appears in the subject field of the certificate. For example, `CN=myOauth2Client`.

Clients using self-signed X.509 certificates

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

- As a JWKS.

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

  Include the following properties as part of the client metadata:

  - **token_endpoint_auth_method**, which must be set to `self_signed_tls_client_auth`.
  - **A JWKS, configured as per RFC 7517**, which includes certificate information.

- As a JWKS URI, which AM will check periodically to retrieve the certificates from.

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

  Include the following properties as part of the client metadata:

  - **token_endpoint_auth_method**, which must be set to `self_signed_tls_client_auth`.
  - **jwks_uri**, which must be set to the URI from where AM will retrieve the certificates. For example, `https://www.example.com/mysecureapps/certs`.

- As an X.509 certificate.

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

  Include the following properties as part of the client metadata:

  - **token_endpoint_auth_method**, which must be set to `self_signed_tls_client_auth`.
  - **tls_client_auth_x509_cert**, which must be set to an X.509 certificate in PEM format. You can choose to include or exclude the `-----BEGIN CERTIFICATE-----` and `-----END CERTIFICATE-----` labels.

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

```
$ curl \
 --request POST \
 --header "Content-Type: application/json" \
 --data '{
    "jwks": {
        "keys": [{
            "kty": "EC",
```

```
            "crv": "P-256",
            "x": "9BmRru-6AYQ8U_9tUFhMGVG-BvC4vRthzLJTntfSdBA",
            "y": "MqPzVSeVNzzgcR-zZeLGog3GJ4d-doRE9eiGkCKrB48",
            "kid": "a4:68:90:1c:f6:c1:43:c0",
            "x5c": [
                "MIIBZTCCAQugAwIB.....xgASSpAQC83FVBawjmbv6k4CN95G8zHsA=="
            ]
        }]
    },
    "client_type": "Confidential",
    "grant_types": ["authorization_code", "client_credentials"],
    "response_types": ["code", "token"],
    "redirect_uris": ["https://client.example.com:8443/callback"],
    "token_endpoint_auth_method": "self_signed_tls_client_auth",
    "tls_client_auth_subject_dn": "CN=myOauth2Client",
    "tls_client_certificate_bound_access_tokens": true
}' \
 "https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/register"
{
  "request_object_encryption_alg": "",
  "default_max_age": 1,
  "jwks": {
    "keys": [
      {
        "kty": "EC",
        "crv": "P-256",
        "x": "9BmRru-6AYQ8U_9tUFhMGVG-BvC4vRthzLJTntfSdBA",
        "y": "MqPzVSeVNzzgcR-zZeLGog3GJ4d-doRE9eiGkCKrB48",
        "kid": "a4:68:90:1c:f6:c1:43:c0",
        "x5c": [
          "MIIBZTCCAQugAwIB.....xgASSpAQC83FVBawjmbv6k4CN95G8zHsA=="
        ]
      }
    ]
  },
  "application_type": "web",
  "tls_client_auth_subject_dn": "CN=myOauth2Client",
  "registration_client_uri": "https://openam.example.com:8443/openam/oauth2/realms/root/realms/
alpha/register?client_id=83635999-2794-4fcd-b6b3-67e2d86c1952",
  "client_type": "Confidential",
  "userinfo_encrypted_response_alg": "",
  "registration_access_token": "tu4KR0jO3iGn0ubOOY0YCSfyPmk",
  "client_id": "83635999-2794-4fcd-b6b3-67e2d86c1952",
  "token_endpoint_auth_method": "self_signed_tls_client_auth",
  "userinfo_signed_response_alg": "",
  "public_key_selector": "jwks",
...
}
```

Note that the example sets `tls_client_certificate_bound_access_tokens` to `true` to let the client obtain certificate-bound access tokens. For more information, see "Certificate-Bound Proof-of-Possession" in the *OAuth 2.0 Guide*.

Tip

OpenID Connect clients must ensure that the following information is present in the JSON:

- The `openid` scope. For example, `"scopes": ["profile", "openid"]`.
- The `id_token` response type. For example, `"response_types": ["code", "id_token code"]`.

+ *Dynamic Client Registration With Access Token*

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

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

Client ID

`masterClient`

Client secret

`password`

Scope(s)

`dynamic_client_registration`

Prior to registration, obtain an access token:

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

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

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "Authorization: Bearer 5e7d1019-b752-43f1-af97-0d6fe2753105" \
--data '{
    "redirect_uris": ["https://client.example.com/callback"],
    "client_name#en": "My Client",
    "client_name#ja-Jpan-JP": "\u30AF\u30E9\u30A4\u30A2\u30F3\u30C8\u540D",
```

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

Tip

OpenID Connect clients must ensure that the following information is present in the JSON:

- The `openid` scope. For example, `"scopes": ["profile", "openid"]`.
- The `id_token` response type. For example, `"response_types": ["code", "id_token code"]`.

+ *Dynamic Client Registration With Software Statement*

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

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

Name

`My Software Publisher`

Software publisher secret

`secret`

Software publisher issuer

`https://client.example.com`

Software statement signing Algorithm

`HS256`

Public key selector

`JWKs`

Json Web Key

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

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

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


```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
eyJpc3MiOiJodHRwczovL2NsaWVudC5leGFtcGxlLmNvbSIsImlhdCI6MTUwNjY3MTg1MSwiZX
hwIjoxNTM4MjA3ODUxLCJhdWQiOiJvcGVuYW0uZXhhbXBsZS5jb20iLCJzdWIiOiI0TlJCMS0w
WFpBQlpJOUU2LTVTTTNSIiwicmVkaXJlY3RfdXJpcyI6WyJodHRwczovL2NsaWVudC5leGFtcG
xlLmNvbS9jYWxsYmFjayJdfQ.
IOxZaWTOzSPkEkrXC9nj8RDrpulzzMuZ-4R7_Ol_jhw
```


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


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


Prior to registration, obtain an access token:

```
$  curl --request POST \
--data "grant_type=password" \
--data "username=demo" \
--data "password=Ch4ng31t" \
--data "scope=dynamic_client_registration" \
--data "client_id=masterClient" \
--data "client_secret=password" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token"
{
   "access_token":"06bfc193-1f7b-49a1-9926-ffe19e2f5f70",
   "scope":"dynamic_client_registration",
   "token_type":"Bearer",
   "expires_in":3599
}
```

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

```
$ curl \
 --request POST \
 --header "Content-Type: application/json" \
 --header "Authorization: Bearer 06bfc193-1f7b-49a1-9926-ffe19e2f5f70" \
 --data '{
 "redirect_uris": ["https://client.example.com:8443/callback"],
 "client_name#en": "My Client",
 "client_name#ja-Jpan-JP": "\u30AF\u30E9\u30A4\u30A2\u30F3\u30C8\u540D",
 "client_uri": "https://client.example.com/",
 "software_statement": "eyJ0eXAiOiJKV1QiLCJ6W....9nj8RDrpulzzMuZ-4R7_Ol_jhw"
 }' \
 "https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/register"
{
 "request_object_encryption_alg": "",
 "default_max_age": 1,
 "application_type": "web",
 "client_name#en": "My Client",
 "registration_client_uri": "https://openam.example.com:8443/openam/oauth2/realms/root/realms/
alpha/register?client_id=086658c1-0517-4667-bc2d-6786224eb126",
 "client_type": "Confidential",
 "userinfo_encrypted_response_alg": "",
 "registration_access_token": "06bfc193-1f7b-49a1-9926-ffe19e2f5f70",
 "client_id": "086658c1-0517-4667-bc2d-6786224eb126",
 "token_endpoint_auth_method": "client_secret_basic",
 "userinfo_signed_response_alg": "",
 "software_statement": "eyJ0eXAiOiJKV1QiLCJ6W....9nj8RDrpulzzMuZ-4R7_Ol_jhw",
 "public_key_selector": "x509",
 "authorization_code_lifetime": 0,
 "client_secret": "272e26a4-b4ea-4033-bfd3-8b1be2c9aa22",
 "user_info_response_format_selector": "JSON",
 "id_token_signed_response_alg": "HS256",
 "default_max_age_enabled": false,
 "subject_type": "public",
 "jwt_token_lifetime": 0,
 "id_token_encryption_enabled": false,
 "redirect_uris": ["https://client.example.com:8443/callback"],
 "client_name#ja-jpan-jp": "#######",
 "id_token_encrypted_response_alg": "RSA1_5",
```

```
 "id_token_encrypted_response_enc": "A128CBC_HS256",
 "client_secret_expires_at": 0,
 "access_token_lifetime": 0,
 "refresh_token_lifetime": 0,
 "request_object_signing_alg": "",
 "response_types": ["code"]
}
```

Tip

OpenID Connect clients must ensure that the following information is present in the JSON:

- The `openid` scope. For example, `"scopes": ["profile", "openid"]`.
- The `id_token` response type. For example, `"response_types": ["code", "id_token code"]`.

Note

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

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

Dynamic Client Registration Management

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

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

- *registration_client_uri*. The FQDN of the client configuration endpoint the client can use to update their data. This endpoint always contains the client ID as a query parameter.

  Note that you cannot provide a client ID when dynamically registering a new client as this is a generated value.

- *registration_access_token*. The token clients must use to authenticate to the client configuration endpoint.

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

To Configure Client Dynamic Registration Management

1. Configure AM for dynamic client registration.

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

2. In the AM console, go to Realms > *Realm Name* > Services > OAuth2 Provider > Client Dynamic Registration, and ensure that Generate Registration Access Tokens is enabled.

3. Save your changes, if required.

4. (Optional) Review the following examples:

   + *Client Read Request*

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

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

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

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

   ```
   ...
   "registration_client_uri":"https://openam.example.com:8443/openam/oauth2/register?
   client_id=77f296b3-5293-4219-981d-128322c1e173",
   "registration_access_token":"o5IlBXOxC7RGPTIe8is_zzz6Yqg",
   ...
   ```

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

```
$ curl \
--request GET \
--header "Authorization: Bearer o5IlBXOxC7RGPTIe8is_zzz6Yqg" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/register?
client_id=77f296b3-5293-4219-981d-128322c1e173"
{
   "request_object_encryption_alg":"",
   "default_max_age":1,
   "application_type":"web",
   "client_name#en":"My Client",
   "userinfo_encrypted_response_enc":"A128CBC_HS256",
   "client_type":"Confidential",
   "userinfo_encrypted_response_alg":"",
   "token_endpoint_auth_method":"client_secret_basic",
   "userinfo_signed_response_alg":"",
   "client_id":"77f296b3-5293-4219-981d-128322c1e173",
   "public_key_selector":"x509",
   "client_secret":"vXxY3HJ7_...84qfRQHYW3QbZfDSXieAgIVa2tg",
   ....
}
```

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

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

+ *Client Update Request*

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

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

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

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

```
...
"registration_client_uri":"https://openam.example.com:8443/openam/oauth2/register?
client_id=77f296b3-5293-4219-981d-128322c1e173",
"registration_access_token":"o5IlBXOxC7RGPTIe8is_zzz6Yqg",
"grant_types":["authorization_code"],
...
```

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

When updating the client's metadata, AM:

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

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

Note that to update the client scopes, you must ensure that the scopes are set in the Client Registration Scope Whitelist field in the provider's "Advanced" in the *Reference* configuration.

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

```
$ curl \
--request PUT \
--header "Authorization: Bearer o5IlBXOxC7RGPTIe8is_zzz6Yqg" \
--data '{
    "request_object_encryption_alg":"",
    "default_max_age":1,
    "application_type":"web",
    "client_name#en":"My Client",
    "userinfo_encrypted_response_enc":"",
    "client_type":"Confidential",
    "userinfo_encrypted_response_alg":"",
    "token_endpoint_auth_method":"client_secret_basic",
    "userinfo_signed_response_alg":"",
    "client_id":"77f296b3-5293-4219-981d-128322c1e173",
```

```
    "public_key_selector":"x509",
    "scope":"write",
    "authorization_code_lifetime":0,
    "client_secret":"vXxY3HJ7_...84qfRQHYW3QbZfDSXieAgIVa2tg",
    "user_info_response_format_selector":"JSON",
    "tls_client_certificate_bound_access_tokens":false,
    "id_token_signed_response_alg":"RS256",
    "default_max_age_enabled":false,
    "subject_type":"public",
    "grant_types":["authorization_code","implicit"],
    "jwt_token_lifetime":0,
    "id_token_encryption_enabled":false,
    "redirect_uris":["https://client.example.com:8443/callback"],
    "client_name#ja-jpan-jp":"#######",
    "id_token_encrypted_response_alg":"RSA-OAEP-256",
    "id_token_encrypted_response_enc":"A128CBC-HS256",
    "access_token_lifetime":0,
    "refresh_token_lifetime":0,
    "scopes":["write"],
    "request_object_signing_alg":"",
    "response_types":["code"]
}' \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/register?
client_id=77f296b3-5293-4219-981d-128322c1e173"
{
    "request_object_encryption_alg":"",
    "default_max_age":1,
    "application_type":"web",
    "client_name#en":"My Client",
    "userinfo_encrypted_response_enc":"",
    "client_type":"Confidential",
    "userinfo_encrypted_response_alg":"",
    "registration_access_token":"NrvX2bqydMgr...EGI32YuvyrkxDpD_xJVHtHo6fXQ",
    "client_id":"77f296b3-5293-4219-981d-128322c1e173",
    ...
}
```

Tip

To update the client profile with custom attributes, specify the properties as part of the request, for example:

```
--data '{
    "customProperties":["customProperty1=1",customProperty2=2"]
}'
```

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

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

+ *Client Deletion Request*

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

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

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

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

```
...
"registration_client_uri":"https://openam.example.com:8443/openam/oauth2/register?
client_id=77f296b3-5293-4219-981d-128322c1e173",
"registration_access_token":"o5IlBXOxC7RGPTIe8is_zzz6Yqg",
...
```

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

```
$ curl \
--request DELETE \
--header "Authorization: Bearer o5IlBXOxC7RGPTIe8is_zzz6Yqg" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/register?
client_id=77f296b3-5293-4219-981d-128322c1e173"
```

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

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

Chapter 5

OpenID Connect Client Authentication

OAuth 2.0 and OpenID Connect clients can use the same authentication methods described in "*OAuth 2.0 Client Authentication*" in the *OAuth 2.0 Guide*.

However, when using OpenID Connect, you must specify in the client profile the type of authentication the client is using. To configure the authentication method, go to Realms > *Realm Name* > Applications > OAuth 2.0 > Advanced, and select one of the following options in the Token Endpoint Authentication Method drop down:

- **client_secret_post**, if the client sends its credentials as form parameters.
- **client_secret_basic**, if the client sends its credentials in a basic authorization header.
- **private_key_jwt**, if the client sends its credentials as a JWT.
- **tls_client_auth**, if the client uses a CA-signed certificate for mutual TLS authentication.
- **self_signed_tls_client_auth**, if the client uses a self-signed certificate for mutual TLS authentication.
- **none**, if the client is public.

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

Chapter 6

OpenID Connect Grant Flows

This chapter describes the OpenID Connect flows that AM supports as per OpenID Connect Core 1.0 incorporating errata set 1, and also provides the information required to implement them. All the examples assume the realm is configured for CTS-based tokens, but the examples also apply to client-based tokens.

You should decide which flow is best for your environment based on the application that would be the relying party. The following table provides an overview of the flows AM supports when they should be used:

Deciding Which Flow to Use Depending on the Relying Party

Relying Party	Which Grant to use?	Description
The relying party is a web application running on a server. For example, a `.war` application.	**Authorization Code**	The OpenID Connect provider uses the user-agent, for example, the end user's browser, to transport a code that is later exchanged for an ID token (and/or an access token). For security purposes, you should use the Authorization Code grant with PKCE when possible.
The relying party is a native application or a single-page application (SPA). For example, a desktop, a mobile application, or a JavaScript application.	**Authorization Code with PKCE**	Since the relying party does not communicate securely with the OpenID Connect provider, the code may be intercepted by malicious users. The implementation of the Proof Key for Code Exchange (PKCE) standard mitigates against those attacks.
The relying party knows the user's identifier, and wishes to gain consent for an operation from the user by means of a separate authentication device.	**Backchannel Request Grant**	The relying party does not require that the user interacts directly with it; instead it can initiate a backchannel request to the user's authentication device, such as a mobile phone with an authenticator app installed, to authenticate the user and consent to the operation. For example, a smart speaker wants to authenticate and gain consent from its registered user after receiving a voice request to transfer money to a third-party.
The relying party is an SPA. For example, a JavaScript application.	**Implicit**	The OpenID Connect provider uses the user-agent, for example, the end user's browser, to transport an ID token (and maybe an access token) to the relying party.

Relying Party	Which Grant to use?	Description
		Therefore, the tokens might be exposed to the end user and other applications. For security purposes, you should use the Authorization Code grant with PKCE when possible.
The relying party is an application that can use the ID token immediately, and then request an access token and/or a refresh token.	**Hybrid**	AM uses the user-agent, for example, the end user's browser, to transport any combination of ID token, access token, and authorization code to the relying party. The relying party uses the ID token immediately. Later on, it can use either the access token to request a refresh token, or the authorization code to request an access token. For security purposes, you should implement the PKCE specification when using the Hybrid flow when possible.

Tip

ForgeRock provides a Postman collection to try out the flows. See "ForgeRock Grant Flows Collection" in the *OAuth 2.0 Guide*.

Authorization Code Grant

Endpoints

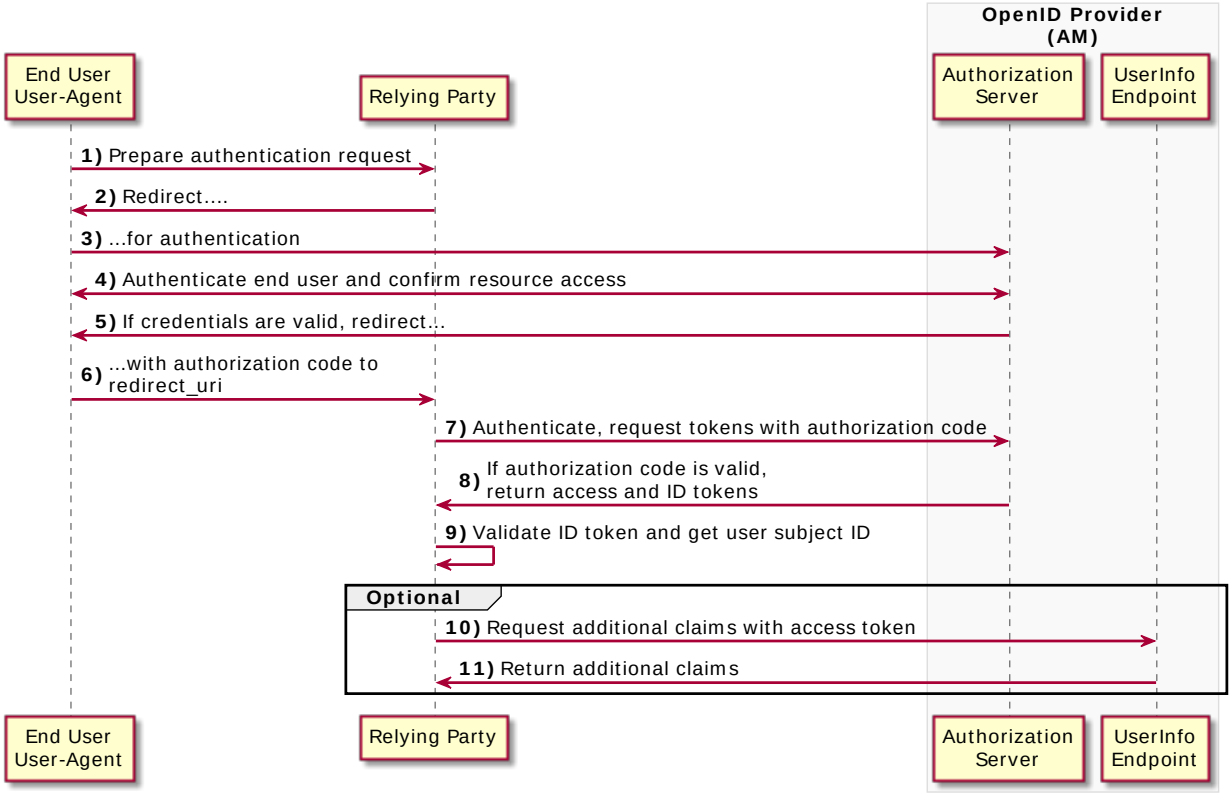
- /oauth2/authorize in the *OAuth 2.0 Guide*
- /oauth2/access_token in the *OAuth 2.0 Guide*
- "/oauth2/userinfo"

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

The Authorization Code grant is the most secure of all the OAuth 2.0/OpenID Connect grants for the following reasons:

- It is a two-step process. The user must authenticate and authorize the client to see the resources and the OpenID provider must validate the code again before issuing the access/ID tokens.
- The OpenID provider delivers the tokens directly to the client, usually over HTTPS. The client secret is never exposed publicly, which protects confidential clients.

OpenID Connect Authorization Code Grant Flow



+ *Authorization Code Grant Flow Explained*

The steps in the diagram are described below:

1. The end user wants to use the services provided by the relying party. The relying party, usually a web-based service, requires an account to provide those services.

   The end user issues a request to the relying party to access their information, which is stored in an OpenID provider.

2. To access the end user's information in the provider, the relying party requires authorization from the end user. Therefore, the relying party redirects the end user's user-agent...

3. ... to the OpenID provider.

4. The OpenID provider authenticates the end user, confirms resource access, and gathers consent if not previously saved.
5. The OpenID provider redirects the end user's user agent to the relying party.
6. During the redirection process, the OpenID provider appends an authorization code.
7. The relying party receives the authorization code and authenticates to the OpenID provider to exchange the code for an access token and an ID token (and a refresh token, if applicable).

   Note that this example assumes a confidential client. Public clients are not required to authenticate.
8. If the authorization code is valid, the OpenID provider returns an access token and an ID token (and a refresh token, if applicable) to the relying party.
9. The relying party validates the ID token and its claims.

   Now, the relying party can use the ID token subject ID claim as the end user's identity.
10. The relying party may require more claims than those included in the ID token. In this case, it makes a request to the OpenID provider's `oauth2/userinfo` endpoint with the access token.
11. If the access token is valid, the `oauth2/userinfo` endpoint returns additional claims, if any.

    Now, the relying party can use the subject ID and the additional retrieved claims as the end user's identity.

Perform the steps in the following procedures to obtain an authorization code and exchange it for an ID token and an access token:

- "To Obtain an Authorization Code Using a Browser in the Authorization Code Grant Flow"
- "To Obtain an Authorization Code Without Using a Browser in the Authorization Code Grant Flow"
- "To Exchange an Authorization Code for an ID/Access Token in the Authorization Code Grant Flow"

To Obtain an Authorization Code Using a Browser in the Authorization Code Grant Flow

This example shows how to obtain an ID token and an access token. It adds notes on how to obtain an ID token only, as well.

This procedure assumes the following configuration:

- AM is configured as an OAuth 2.0/OpenID provider. Ensure that:
  - The `code` plugin is configured in the Response Type Plugins field.
  - The `Authorization Code` grant type is configured in the Grant Types field.
- A *confidential* client called `myClient` is registered in AM with the following configuration:

- **Client secret**: `forgerock`
- **Scopes**: `openid profile`
- **Response Types**: `code`
- **Grant Types**: `Authorization Code`
- **Token Endpoint Authentication Method**: `client_secret_post`

  Confidential OpenID Connect clients can use several methods to authenticate. For more information, see "*OpenID Connect Client Authentication*".

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

Perform the steps in this procedure to obtain an authorization code using a browser:

1. The client redirects the end user's user-agent to the authorization server's authorization endpoint specifying, at least, the following form parameters:

   - **client_id**=*your_client_id*
   - **response_type**=code
   - **redirect_uri**=*your_redirect_uri*
   - **scope**=openid profile

   For information about the parameters supported by the `/oauth2/authorize` endpoint, see "/oauth2/authorize" in the *OAuth 2.0 Guide*.

   If the OAuth 2.0/OpenID provider is configured for a subrealm rather than the Top Level Realm, you must specify it in the endpoint. For example, if the OAuth 2.0/OpenID provider is configured for the `/alpha` realm, then use `/oauth2/realms/root/realms/alpha/authorize`.

   For example:

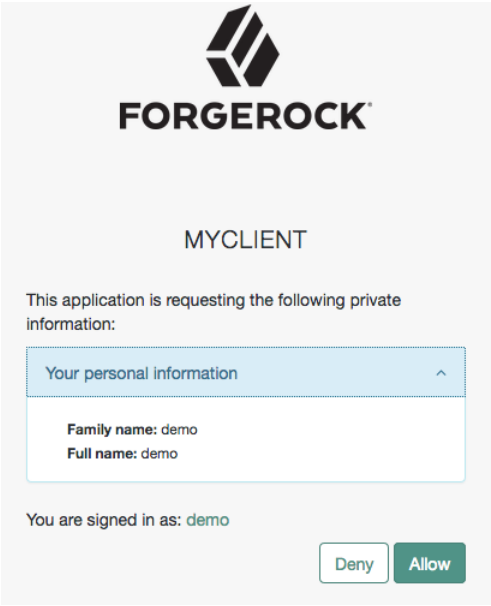
   ```
   https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/authorize \
   ?client_id=myClient \
   &response_type=code \
   &scope=openid%20profile \
   &state=abc123 \
   &nonce=123abc \
   &redirect_uri=https://www.example.com:443/callback
   ```

   Note that the URL is split and spaces have been added for readability purposes. The `state` and `nonce` parameters have been included to protect against CSRF and replay attacks.

2. The end user authenticates to AM, for example, using the credentials of the `demo` user. In this case, they log in using the default chain or tree configured for the realm.

After logging in, AM presents its consent screen:

OpenID Connect Consent Screen



Note that requesting the `profile` scope translates into requesting access to several claims. For more information about the special `profile` scope, see "*About Claims*".

3. The end user selects the `Allow` button to grant consent for the `profile` scope.

   AM redirects the end user to the URL specified in the `redirect_uri` parameter.

4. Inspect the URL in the browser. It contains a `code` parameter with the authorization code AM has issued. For example:

   ```
   https://www.example.com:443/callback?code=g5B3qZ8rWzKIU2xodV_kkSIk0F4&iss=https://
   openam.example.com:8443/openam/oauth2&state=abc123&client_id=myClient
   ```

5. The client performs the steps in "To Exchange an Authorization Code for an ID/Access Token in the Authorization Code Grant Flow" to exchange the authorization code for an access token and an ID token.

To Obtain an Authorization Code Without Using a Browser in the Authorization Code Grant Flow

This example shows how to obtain an ID token and an access token. It adds notes on how to obtain an ID token only, as well.

This procedure assumes the following configuration:

- AM is configured as an OAuth 2.0/OpenID provider. Ensure that:
  - The `code` plugin is configured in the Response Type Plugins field.
  - The `Authorization Code` grant type is configured in the Grant Types field.
- A *confidential* client called `myClient` is registered in AM with the following configuration:
  - **Client secret**: `forgerock`
  - **Scopes**: `openid profile`
  - **Response Types**: `code`
  - **Grant Types**: `Authorization Code`
  - **Token Endpoint Authentication Method**: `client_secret_post`

    Confidential OpenID Connect clients can use several methods to authenticate. For more information, see "*OpenID Connect Client Authentication*".

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

Perform the steps in this procedure to obtain an authorization code without using a browser:

1. The end user logs in to AM, for example, using the credentials of the `demo` user. For example:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "X-OpenAM-Username: demo" \
--header "X-OpenAM-Password: Ch4ng31t" \
--header "Accept-API-Version: resource=2.0, protocol=1.0" \
'https://openam.example.com:8443/openam/json/realms/root/realms/alpha/authenticate'
{
    "tokenId":"AQIC5wM...TU3OQ*",
    "successUrl":"/openam/console",
    "realm":"/alpha"
}
```

2. The client makes a POST call to AM's authorization endpoint, specifying the SSO token of the `demo` in a cookie and, at least, the following parameters:
   - **client_id**=*your_client_id*

- **response_type**=code
- **redirect_uri**=*your_redirect_uri*
- **scope**=openid profile

  You can configure the `openid` scope as a default scope in the client profile or the OAuth 2.0/OpenID provider to avoid including the scope parameter in your calls, if required.

  However, since the `openid` scope is required in OpenID Connect flows, the example specifies it.
- **decision**=allow
- **csrf**=*demo_user_SSO_token*

For information about the parameters supported by the `/oauth2/authorize` endpoint, see "/oauth2/authorize" in the *OAuth 2.0 Guide*.

If the OAuth 2.0/OpenID provider is configured for a subrealm rather than the Top Level Realm, you must specify it in the endpoint. For example, if the OAuth 2.0/OpenID provider is configured for the `/alpha` realm, then use `/oauth2/realms/root/realms/alpha/authorize`.

For example:

```
$ curl --dump-header - \
--request POST \
--Cookie "iPlanetDirectoryPro=AQIC5wM...TU3OQ*" \
--data "scope=openid profile" \
--data "response_type=code" \
--data "client_id=myClient" \
--data "csrf=AQIC5wM...TU3OQ*" \
--data "redirect_uri=https://www.example.com:443/callback" \
--data "state=abc123" \
--data "nonce=123abc" \
--data "decision=allow" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/authorize"
```

Note that the `state` and `nonce` parameters have been included to protect against CSRF and replay attacks.

If AM is able to authenticate the user and the client, it returns an HTTP 302 response with the authorization code appended to the redirection URL:

```
HTTP/1.1 302 Found
Server: Apache-Coyote/1.1
X-Frame-Options: SAMEORIGIN
Pragma: no-cache
Cache-Control: no-store
Date: Mon, 30 Jul 2018 11:42:37 GMT
Accept-Ranges: bytes
Location: https://www.example.com:443/callback?code=g5B3qZ8rWzKIU2xodV_kkSIk0F4&iss=https%3A%2F
%2Fopenam.example.com%3A8443%2Fopenam%2Foauth2&state=abc123&client_id=myClient
Vary: Accept-Charset, Accept-Encoding, Accept-Language, Accept
Content-Length: 0
```

3. Perform the steps in "To Exchange an Authorization Code for an ID/Access Token in the Authorization Code Grant Flow" to exchange the authorization code for an ID/access token.

To Exchange an Authorization Code for an ID/Access Token in the Authorization Code Grant Flow

Perform the steps in the following procedure to exchange an authorization code for an ID/access token:

1. Ensure the relying party has obtained an authorization code by performing the steps in either "To Obtain an Authorization Code Using a Browser in the Authorization Code Grant Flow" or "To Obtain an Authorization Code Without Using a Browser in the Authorization Code Grant Flow".

2. The relying party makes an HTTP POST request to the token endpoint in the OpenID provider specifying, at least, the following parameters:

   - **grant_type**=authorization_code
   - **code**=*your_authorization_code*
   - **redirect_uri**=*your_redirect_uri*

   For information about the parameters supported by the `/oauth2/access_token` endpoint, see "/oauth2/access_token" in the *OAuth 2.0 Guide*.

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

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

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

   If the OAuth 2.0/OpenID provider is configured for a subrealm rather than the Top Level Realm, you must specify it in the endpoint. For example, if the OAuth 2.0/OpenID provider is configured for the `/alpha` realm, then use `/oauth2/realms/root/realms/alpha/access_token`.

   For example:

```
$ curl --request POST \
--data "grant_type=authorization_code" \
--data "code=g5B3qZ8rWzKIU2xodV_kkSIk0F4" \
--data "client_id=myClient" \
--data "client_secret=forgerock" \
--data "redirect_uri=https://www.example.com:443/callback" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token"
```

The `client_id` and the `redirection_uri` parameters specified in this call must match those used as part of the authorization code request, or AM will not validate the code.

AM returns an ID and an access token. For example:

```
{
   "access_token":"cnM3nSpF5ckCFZOaDem2vANUdqQ",
   "scope":"openid profile",
   "id_token":"eyJ0eXAiOiJKV1QiLCJra...7r8soMCk8A7QdQpg",
   "token_type":"Bearer",
   "expires_in":3599
}
```

If the client does not require the access token, revoke it in the *OAuth 2.0 Guide*.

Tip

AM can also issue refresh tokens at the same time the access tokens are issued. For more information, see "*Refresh Tokens*" in the *OAuth 2.0 Guide*.

3. (Optional) The relying party can request additional claims about the end user from AM.

   For more information, see "/oauth2/userinfo".

Tip

For access to a sample JavaScript-based relying party to test the Authorization Code grant flow, see How do I access and build the sample code provided for AM (All versions)? in the *ForgeRock Knowledge Base*.

Clone the example project to deploy it in the same web container as AM. 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 "*Dynamic Client Registration*", and browse the deployment URL to see the initial page.

The example relying party uses an 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.

Authorization Code Grant with PKCE

Endpoints

- /oauth2/authorize in the *OAuth 2.0 Guide*
- /oauth2/access_token in the *OAuth 2.0 Guide*
- "/oauth2/userinfo"

The Authorization Code grant, when combined with the PKCE standard (*RFC 7636*), is used when the client, usually a mobile or a JavaScript application, requires access to protected resources.

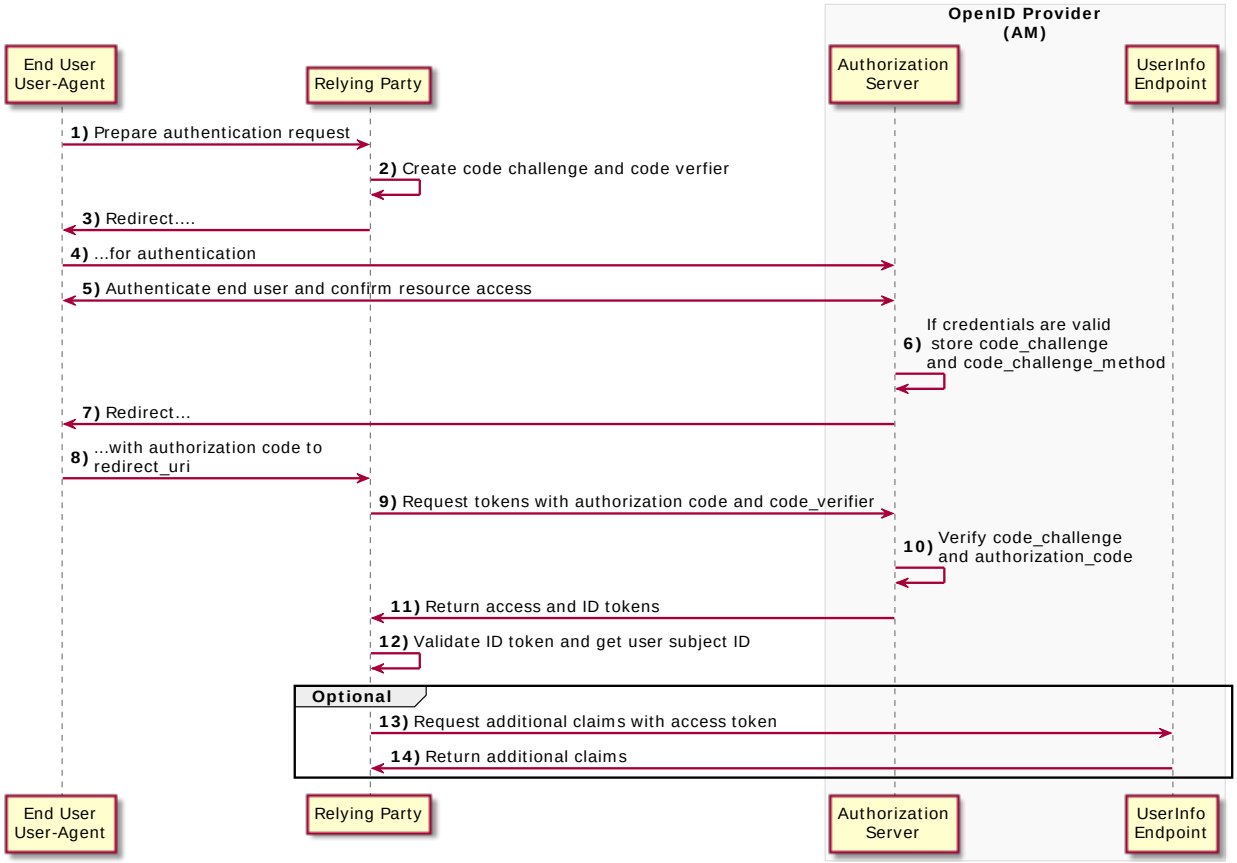
The flow is similar to the regular Authorization Code grant type, but the client must generate a code that will be part of the communication between the client and the OpenID provider. This code mitigates against interception attacks performed by malicious users.

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

The PKCE flow adds three parameters on top of those used for the Authorization code grant:

- **code_verifier** (form parameter). Contains a random string that correlates the authorization request to the token request.
- **code_challenge** (query parameter). Contains a string derived from the code verifier that is sent in the authorization request and that needs to be verified later with the code verifier.
- **code_challenge_method** (query parameter). Contains the method used to derive the code challenge.

OpenID Connect Authorization Code Grant with PKCE Flow



+ *Authorization Code Grant with PKCE Flow Explained*

The steps in the diagram are described below:

1. The end user wants to use the services provided by the relying party. The relying party, usually a web-based service, requires an account to provide those services.

   The end user issues a request to the relying party to access their information which is stored in an OpenID provider.

2. To access the end user's information in the provider, the relying party requires authorization from the end user. When using the PKCE standard, the relying party must generate a unique code and a way to verify it, and append the code to the request for the authorization code.

3. The relying party redirects the end user's user-agent with `code_challenge` and `code_challenge_method`...
4. ... to the OpenID provider.
5. The OpenID provider authenticates the end user, confirms resource access, and gathers consent if not previously saved.
6. If the end user's credentials are valid and they consent to provide their data to the relying party, the OpenID provider stores the code challenge and its method.
7. The OpenID provider redirects the end user's user agent to the redirection URI (usually the relying party).
8. During the redirection process, the OpenID provider appends an authorization code.
9. The relying party receives the authorization code and authenticates to the OpenID provider to exchange the code for an access token and an ID token (and a refresh token, if applicable), appending the verification code to the request.
10. The OpenID provider verifies the code challenge stored in memory using the validation code. It also verifies the authorization code.
11. If both codes are valid, the OpenID provider returns an access and an ID token (and a refresh token, if applicable) to the relying party.
12. The relying party validates the ID token and its claims.

    Now, the relying party can use the ID token subject ID claim as the end user's identity.
13. The relying party may require more claims than those included in the ID token. In this case, it makes a request to AM's `oauth2/userinfo` endpoint with the access token.
14. If the access token is valid, the `oauth2/userinfo` endpoint returns additional claims, if any.

    Now, the relying party can use the subject ID and the additional retrieved claims as the end user's identity.

Perform the steps in the following procedures to obtain an authorization code and exchange it for an access token:

- "To Generate a Code Verifier and a Code Challenge"
- "To Obtain an Authorization Code Using a Browser in the Authorization Code Grant with PKCE Flow"
- "To Obtain an Authorization Code Without Using a Browser in the Authorization Code Grant with PKCE Flow"
- "To Exchange an Authorization Code for an ID/Access Token in the Authorization Code Grant with PKCE Flow"

To Generate a Code Verifier and a Code Challenge

The relying party (the client) must be able to generate a code verifier and a code challenge. For details, see the PKCE standard (*RFC 7636*). The information contained in this procedure is for example purposes only:

1. The client generates the code challenge and the code verifier. Creating the challenge using a SHA-256 algorithm is mandatory if the client supports it, as per the RFC 7636 standard.

   The following is an example of a code verifier and code challenge written in JavaScript:

   ```
   function base64URLEncode(words) {
      return CryptoJS.enc.Base64.stringify(words)
      .replace(/\+/g, '-')
      .replace(/\//g, '_')
      .replace(/=/g, '');
   }
   var verifier = base64URLEncode(CryptoJS.lib.WordArray.random(50));
   var challenge = base64URLEncode(CryptoJS.SHA256(verifier));
   ```

   This example generates values such as `ZpJiIM_G0SE9WlxzS69Cq0mQh8uyFaeEbILlW8tHs62SmEE6n7Nke0XJGx_F4OduTI4` for the code verifier and `j3wKnK2Fa_mc2tgdqa6GtUfCYjdWSA5S23JKTTtPF8Y` for the code challenge. These values will be used in subsequent procedures.

   The relying party is now ready to request an authorization code.

2. The relying party performs the steps in one of the following procedures to request an authorization code:

   - "To Obtain an Authorization Code Using a Browser in the Authorization Code Grant with PKCE Flow"
   - "To Obtain an Authorization Code Without Using a Browser in the Authorization Code Grant with PKCE Flow"

To Obtain an Authorization Code Using a Browser in the Authorization Code Grant with PKCE Flow

This procedure assumes the following configuration:

- AM is configured as an OAuth 2.0/OpenID provider. Ensure that:
  - The `code` plugin is configured in the Response Type Plugins field.
  - The `Authorization Code` grant type is configured in the Grant Types field.

  The Code Verifier Parameter Required drop-down menu (Realms > *Realm Name* > Services > OAuth2 Provider > Advanced) specifies whether AM requires clients to include a code verifier in their calls.

However, if a client makes a call to AM with the `code_challenge` parameter, AM will honor the code exchange regardless of the configuration of the Code Verifier Parameter Required drop-down menu.

- A *public* client called `myClient` is registered in AM with the following configuration:
  - **Scopes**: `openid profile`
  - **Response Types**: `code`
  - **Grant Types**: `Authorization Code`
  - **Token Endpoint Authentication Method**: `none`

    If you were using a confidential OpenID Connect client, you must specify a method to authenticate. For more information, see "*OpenID Connect Client Authentication*".

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

Perform the steps in this procedure to obtain an authorization code using a browser:

1. The relying party redirects the end user's user-agent to the AM's authorization endpoint specifying, at least, the following query parameters:
   - **client_id**=*your_client_id*
   - **response_type**=code
   - **redirect_uri**=*your_redirect_uri*
   - **code_challenge**=*your_code_challenge*
   - **code_challenge_method**=S256
   - **scope**=openid profile

   For information about the parameters supported by the `/oauth2/authorize` endpoint, see "/oauth2/authorize" in the *OAuth 2.0 Guide*.

   If the OAuth 2.0/OpenID provider is configured for a subrealm rather than the Top Level Realm, you must specify it in the endpoint. For example, if the OAuth 2.0/OpenID provider is configured for the `/alpha` realm, then use `/oauth2/realms/root/realms/alpha/authorize`.

   For example:

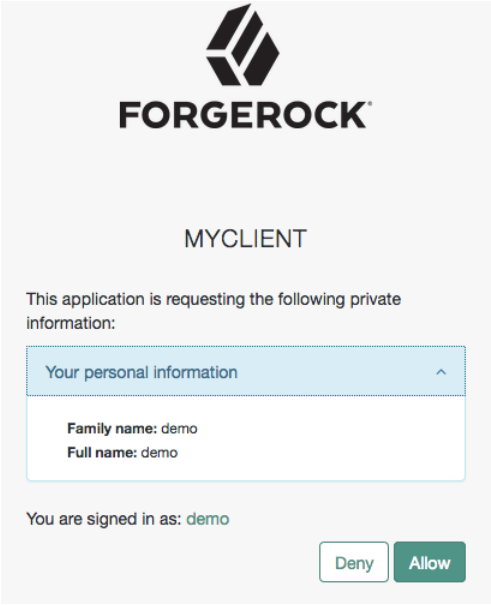
```
https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/authorize \
?client_id=myClient \
&response_type=code \
&scope=openid%20profile \
&redirect_uri=https://www.example.com:443/callback \
&code_challenge=j3wKnK2Fa_mc2tgdqa6GtUfCYjdWSA5S23JKTTtPF8Y \
&code_challenge_method=S256 \
&nonce=123abc \
&state=abc123
```

Note that the URL is split and spaces have been added for readability purposes. The `state` and `nonce` parameters have been included to protect against CSRF and replay attacks.

2. The end user authenticates to AM, for example, using the credentials of the `demo` user. In this case, they log in using the default chain or tree configured for the realm.

   After logging in, AM presents its consent screen:

OpenID Connect Consent Screen



Note that requesting the `profile` scope translates into requesting access to several claims. For more information about the special `profile` scope, see "*About Claims*".

3. The end user selects the `Allow` button to grant consent for the `profile` scope.

   AM redirects the end user to the URL specified in the `redirect_uri` parameter.

4. Inspect the URL in the browser. It contains a `code` parameter with the authorization code AM has issued. For example:

```
https://www.example.com:443/callback?code=g5B3qZ8rWzKIU2xodV_kkSIk0F4&iss=https://
openam.example.com:8443/openam/oauth2&state=abc123&client_id=myClient
```

5. The client performs the steps in "To Exchange an Authorization Code for an ID/Access Token in the Authorization Code Grant with PKCE Flow" to exchange the authorization code for an ID/access token.

To Obtain an Authorization Code Without Using a Browser in the Authorization Code Grant with PKCE Flow

This procedure assumes the following configuration:

- AM is configured as an OAuth 2.0/OpenID provider. Ensure that:
  - The `code` plugin is configured in the Response Type Plugins field.
  - The `Authorization Code` grant type is configured in the Grant Types field.

  The Code Verifier Parameter Required drop-down menu (Realms > *Realm Name* > Services > OAuth2 Provider > Advanced) specifies whether AM require clients to include a code verifier in their calls.

  However, if a client makes a call to AM with the `code_challenge` parameter, AM will honor the code exchange regardless of the configuration of the Code Verifier Parameter Required drop-down menu.

- A *public* client called `myClient` is registered in AM with the following configuration:
  - **Scopes**: `openid profile`
  - **Response Types**: `code`
  - **Grant Types**: `Authorization Code`
  - **Redirection URIs**: `https://www.example.com:443/callback`
  - **Token Endpoint Authentication Method**: `none`

    Confidential OpenID Connect clients can use several methods to authenticate. For more information, see "*OpenID Connect Client Authentication*".

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

Perform the steps in this procedure to obtain an authorization code:

1. The end user logs in to AM, for example, using the credentials of the `demo` user. For example:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "X-OpenAM-Username: demo" \
--header "X-OpenAM-Password: Ch4ng31t" \
--header "Accept-API-Version: resource=2.0, protocol=1.0" \
'https://openam.example.com:8443/openam/json/realms/root/realms/alpha/authenticate'
{
    "tokenId":"AQIC5wM...TU3OQ*",
    "successUrl":"/openam/console",
    "realm":"/alpha"
}
```

2. The client makes an HTTP POST request to AM's authorization endpoint, specifying in a cookie the SSO token of the `demo` and, at least, the following parameters:

   - **client_id**=*your_client_id*
   - **response_type**=code
   - **redirect_uri**=*your_redirect_uri*
   - **decision**=allow
   - **csrf**=*demo_user_SSO_token*
   - **code_challenge**=*your_code_challenge*
   - **code_challenge_method**=S256
   - **scope**=openid profile

     You can configure the `openid` scope as a default scope in the client profile or the OAuth 2.0/OpenID provider to avoid including the scope parameter in your calls, if required.

     However, since the `openid` scope is required in OpenID Connect flows, the example specifies it.

   For information about the parameters supported by the `/oauth2/authorize` endpoint, see "/oauth2/authorize" in the *OAuth 2.0 Guide*.

   If the OAuth 2.0/OpenID provider is configured for a subrealm rather than the Top Level Realm, you must specify it in the endpoint. For example, if the OAuth 2.0/OpenID provider is configured for the `/alpha` realm, then use `/oauth2/realms/root/realms/alpha/authorize`.

   For example:

```
$ curl --dump-header - \
--request POST \
--Cookie "iPlanetDirectoryPro=AQIC5wM...TU3OQ*" \
--data "redirect_uri=https://www.example.com:443/callback" \
--data "scope=openid profile" \
--data "response_type=code" \
--data "client_id=myClient" \
--data "csrf=AQIC5wM...TU3OQ*" \
--data "state=abc123" \
--data "nonce=123abc" \
--data "decision=allow" \
--data "code_challenge=j3wKnK2Fa_mc2tgdqa6GtUfCYjdWSA5S23JKTTtPF8Y" \
--data "code_challenge_method=S256" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/authorize"
```

Note that the `state` and `nonce` parameters have been included to protect against CSRF and replay attacks.

If AM is able to authenticate the user and the client, it returns an HTTP 302 response with the authorization code appended to the redirection URL:

```
HTTP/1.1 302 Found
Server: Apache-Coyote/1.1
X-Frame-Options: SAMEORIGIN
Pragma: no-cache
Cache-Control: no-store
Date: Mon, 30 Jul 2018 11:42:37 GMT
Accept-Ranges: bytes
Location: https://www.example.com:443/callback?code=g5B3qZ8rWzKIU2xodV_kkSIk0F4&iss=https%3A%2F
%2Fopenam.example.com%3A8443%2Fopenam%2Foauth2&state=abc123&client_id=myClient
Vary: Accept-Charset, Accept-Encoding, Accept-Language, Accept
Content-Length: 0
```

3. Perform the steps in "To Exchange an Authorization Code for an ID/Access Token in the Authorization Code Grant with PKCE Flow" to exchange the authorization code for an ID/access token.

To Exchange an Authorization Code for an ID/Access Token in the Authorization Code Grant with PKCE Flow

Perform the steps in the following procedure to exchange an authorization code for an ID/access token:

1. Ensure the client has obtained an authorization code by performing the steps in either "To Obtain an Authorization Code Using a Browser in the Authorization Code Grant with PKCE Flow" or "To Obtain an Authorization Code Without Using a Browser in the Authorization Code Grant with PKCE Flow".

2. The client creates a POST request to the token endpoint in the authorization server specifying, at least, the following parameters:

   - **grant_type**=authorization_code

- **code**=*your_authorization_code*
- **client_id**=*your_client_id*
- **redirect_uri**=*your_redirect_uri*
- **code_verifier**=*your_code_verifier*

For information about the parameters supported by the `/oauth2/access_token` endpoint, see "/oauth2/access_token" in the *OAuth 2.0 Guide*.

For example:

```
$ curl --request POST \
--data "grant_type=authorization_code" \
--data "code=g5B3qZ8rWzKIU2xodV_kkSIk0F4" \
--data "client_id=myClient" \
--data "redirect_uri=https://www.example.com:443/callback" \
--data "code_verifier=ZpJiIM_G0SE9WlxzS69Cq0mQh8uyFaeEbILlW8tHs62SmEE6n7Nke0XJGx_F4OduTI4" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token"
```

The `client_id` and the `redirection_uri` parameters specified in this call must match those used as part of the authorization code request, or AM will not validate the code.

AM returns an ID and an access token. For example:

```
{
   "access_token":"cnM3nSpF5ckCFZOaDem2vANUdqQ",
   "scope":"openid profile",
   "id_token":"eyJ0eXAiOiJKV1QiLCJra...7r8soMCk8A7QdQpg",
   "token_type":"Bearer",
   "expires_in":3599
}
```

If the client does not require the access token, revoke it in the *OAuth 2.0 Guide*.

Tip

AM can also issue refresh tokens at the same time the access tokens are issued. For more information, see "*Refresh Tokens*" in the *OAuth 2.0 Guide*.

3. (Optional) The relying party can request additional claims about the end user from AM.

   For more information, see "/oauth2/userinfo".

Backchannel Request Grant

Endpoints

- /oauth2/bc-authorize in the *OAuth 2.0 Guide*
- /oauth2/access_token in the *OAuth 2.0 Guide*
- "/oauth2/userinfo"

The Backchannel Request grant is used when performing Client Initiated Backchannel Authentication (CIBA).

CIBA allows a client application, known as the *consumption device*, to obtain authentication and consent from a user, without requiring the user to interact with the client directly.

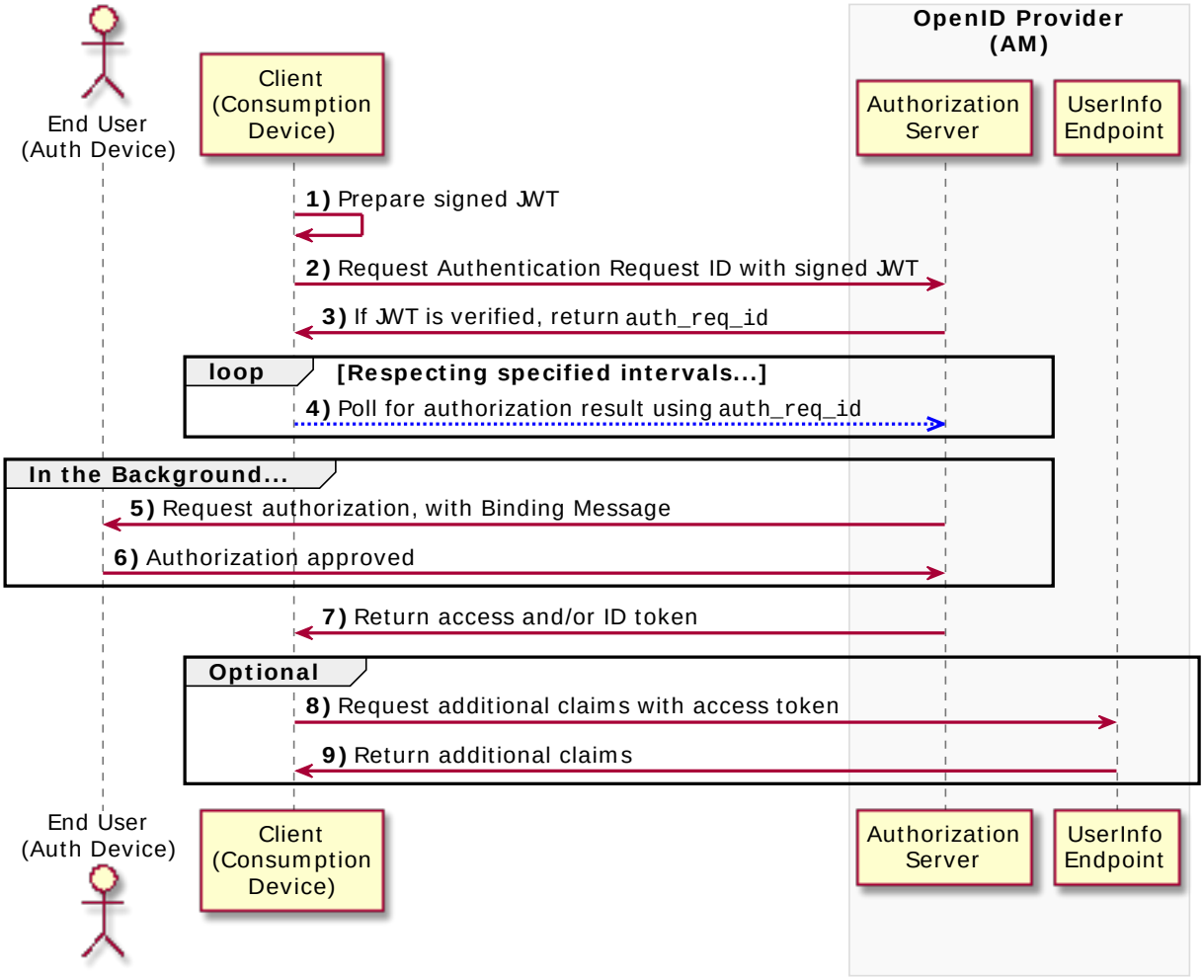
Instead, the user authenticates and consents to the operation using a separate, "decoupled" device, known as the *authentication device*. For example, an authenticator application, or a mobile banking application on their mobile phone.

Note

AM applies the guidelines suggested by the OpenID Financial-grade API (FAPI) Working Group to the implementation of CIBA.

For more information, see OpenID Connect 1.0 in the *Reference*.

OpenID Connect Backchannel Request Grant Flow



+ *Backchannel (CIBA) Request Grant Flow Explained*

The steps in the diagram are described below:

1. The client has a need to authenticate a user. It has a user identifier, and creates a signed JWT.
2. The client creates a POST request containing the signed JWT, and sends it to AM.

3. AM validates the signature using the public key, performs validation checks on the JWT contents, and if verified, returns an `auth_req_id`, as well as a polling interval.
4. The client begins polling AM using the `auth_req_id` to check if the user has authorized the operation. The client must respect the interval returned each time, otherwise an error message is returned.
5. AM sends the user a push notification message, including the contents of the `binding_message`, requesting authorization.
6. The user authorizes the request by performing the required authorization gesture on their authentication device, usually a mobile phone. For example, it may be swiping a slider, or authenticating using facial recognition or a fingerprint sensor.
7. If the authorization is valid, the OpenID provider returns an access token token (and an ID/refresh token, if applicable) to the client.

   Now, the client can use the ID token subject ID claim as the end user's identity.
8. The client may require more claims than those included in the ID token. In this case, it makes a request to the OpenID provider's `oauth2/userinfo` endpoint with the access token.
9. If the access token is valid, the `oauth2/userinfo` endpoint returns additional claims, if any.

   Now, the client can use the subject ID and the additional retrieved claims as the end user's identity.

Perform the steps in the following procedures to obtain an authorization request ID and exchange it for an ID token and an access token:

- "To Configure AM to use the Backchannel Request Grant Flow"
- "To Obtain an Authentication Request ID Using the Backchannel Request Grant Flow"
- "To Exchange an Authorization Request ID for an ID/Access Token in the Backchannel Request Grant Flow"

To Configure AM to use the Backchannel Request Grant Flow

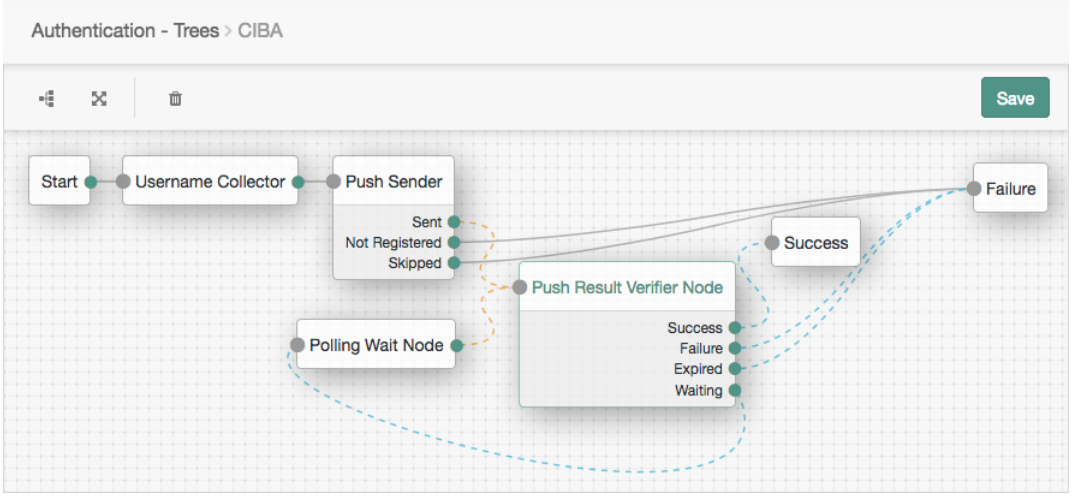
Perform the following steps to prepare AM for the backchannel request grant flow:

1. In AM, configure an OAuth 2.0/OpenID provider. Ensure that:
   - The `Back Channel Request` grant type is configured in the Grant Types field.
2. Associate an authentication tree that performs push authentication with the `acr_values` property contained in the signed JWT.

   The authentication tree must start with a "Username Collector Node" in the *Authentication and Single Sign-On Guide* , and contain a "Push Sender Node" in the *Authentication and Single Sign-*

On Guide and "Push Result Verifier Node" in the *Authentication and Single Sign-On Guide*, and a "Polling Wait Node" in the *Authentication and Single Sign-On Guide*.

The following is an example of a suitable authentication tree:



For more information on creating authentication trees for push authentication, see "Creating Trees for Push Authentication and Registration" in the *Authentication and Single Sign-On Guide*.

To associate a push authentication tree with incoming `acr_values`, perform the following steps:

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

b. In the OpenID Connect acr_values to Auth Chain Mapping box, enter the value of the `acr_values` property in the Key field, and the name of the push authentication tree to use in the Value field, for example `CIBA`, and then click Add.

c. Save your changes.

   For more information, see "The Authentication Context Class Reference (acr) Claim".

For more information, see "*OpenID Provider Configuration*".

3. In AM, create a confidential OAuth 2.0 client with a client ID of `myCIBAClient`. The client ID **must** match the value of the `iss` claim in the signed JWT prepared above.

   The client profile should have the following configuration:

   - **Client secret**: `forgerock`
   - **Scopes**: `openid profile`

- **Grant Types**: `Back Channel Request`
- **Token Endpoint Authentication Method**: `client_secret_basic`

  Confidential OpenID Connect clients can use several methods to authenticate. For more information, see "*OpenID Connect Client Authentication*".

- The client must be provided with the public key of the keypair that will be used to sign the JWT.

  On the Signing and Encryption tab, you must configure *either* the **JWKs URI** or the **JWK Set** fields, as follows:

  - **JWKs URI**: specifies a URI that exposes the public keys AM will use to validate the JWT signature.

    For example, `http://www.example.com/issuer/jwk_uri`.

    > **Note**
    >
    > If you configure this field, ensure the following properties are configured with values suitable for your environment:
    >
    > - JWKs URI content cache timeout in ms
    > - JWKs URI content cache miss cache time

  - **JWK set**: Specifies a JWK set containing the public keys used to validate JWT signatures.

    The following is an example of a public elliptic curve JWK set:

    ```
    {
      "keys": [
        {
          "kty": "EC",
          "use": "sig",
          "crv": "P-256",
          "kid": "myCIBAKey",
          "x": "m0CkpWpZyGu-FLRLjCGBVGC7Fwm5vGt8Lm3HhYU4ylg",
          "y": "U8NMtO-C2c3yhu2I_ApAELttmaittfPNPQaIJxvTCHk",
          "alg": "ES256"
        }
      ]
    }
    ```

    For more information about the contents of the JWK set, see the *JSON Web Key (JWK)* specification.

You can store more than one key in the JWK set. However, it is easier to implement key rotation exposing the validation keys on the URI instead.

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

To Obtain an Authentication Request ID Using the Backchannel Request Grant Flow

Perform the steps in this procedure to obtain an authentication request ID, using CIBA:

1. On the client, prepare a signed JWT. The JWT must contain, at least, the following claims in the payload:

   aud

   Specifies a string or an array of strings that is the intended audience of the JWT. Must be set to the authorization server's OAuth 2.0 endpoint, for example:

   ```
   "aud": "http://openam.example.com:8080/openam/oauth2"
   ```

   exp

   Specifies the expiration time of the JWT in Unix time.

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

   iss

   Specifies the unique identifier of the JWT issuer.

   The identifier must match the client ID of the OAuth 2.0 client in AM, for example *myCIBAClient*.

   login_hint

   Specifies the principal who is the subject of the JWT. It should be a string that identifies the resource owner.

   > **Tip**
   >
   > You can provide a previously obtained ID token in a property named **id_token_hint** as the hint for determining the resource owner, rather than a string.

   scope

   Specifies a space-separated list of the requested scopes. Must include the `openid` scope.

   acr_values

   Specifies an identifier that maps to the authentication mechanism AM uses to obtain authorization from the end user.

binding_message

Specifies a message delivered to the user when obtaining authorization.

Should be a short (100 characters or fewer), description of the operation the end user is authorizing, and should include an identifier to match the authorization request to the client that initiated the request.

> **Note**
>
> If the binding message is sent using push notifications, the following additional limitations apply to the value:
>
> 1. Must begin with a letter, number, or punctuation mark.
> 2. Must **not** include line breaks or control characters.

For example:

```
Allow ExampleBank to transfer £50 from your 'Main' account to your 'Savings' account? (EB-0246326)
```

The following is an example of the payload of a basic JWT:

```
{
  "login_hint": "demo",
  "scope": "openid profile",
  "acr_values": "push",
  "iss": "myCIBAClient",
  "aud": "http://openam.example.com:8080/openam/oauth2",
  "exp": 1559311511,
  "binding_message": "Allow ExampleBank to transfer £50 from your 'Main' account to your 'Savings'
 account? (EB-0246326)"
}
```

For more information about JWTs, see the RFC 7523 standard.

AM ignores keys specified in JWT headers, such as `jku` and `jwe`, and will use the configuration in the client profile to verify the JWT's signature.

2. The client makes a POST request to the authorization server's backchannel authorization endpoint, including the signed JWT, and the client credentials in the authorization header.

   For example:

```
$ curl --request POST \
--header "authorization: Basic bXlDSUJBQ2xpZW50OmZvcmdlcm9jaw==" ❶\
--data "request=eyJhbGci...kfPjAfnBg❷" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/bc-authorize"
```

❶ The basic authorization header is the base64-encoded value of your client ID, a colon character (**:**), and the client secret. For example `myCIBAClient:forgerock`.

For more information about authenticating clients, see "*OAuth 2.0 Client Authentication*" in the *OAuth 2.0 Guide*.

❷ The "request" field should contain the entire signed JWT.

The value in this example has been truncated for display purposes.

AM returns JSON containing the `auth_req_id` value:

```
{
  "auth_req_id": "35Evy3bJXJEnhll2ebacgROYfbU",
  "expires_in": 600,
  "interval": 2
}
```

AM will also send the user a push notification message, containing the contents of the `binding_message`, to request authorization for the operation.

For more information on interacting with push notifications, see "*MFA: Push Authentication*" in the *Authentication and Single Sign-On Guide*.

3. The client performs the steps in "To Exchange an Authorization Request ID for an ID/Access Token in the Backchannel Request Grant Flow" to exchange the authentication request ID for an ID token (and an access/refresh token).

To Exchange an Authorization Request ID for an ID/Access Token in the Backchannel Request Grant Flow

Perform the steps in the following procedure to exchange an authorization request ID for an ID/access token:

1. The client starts to poll the token endpoint in the OpenID provider with HTTP POST requests, with the client credentials in the authorization header, and specifies the following parameters:

   - **grant_type**=urn:openid:params:grant-type:ciba
   - **auth_req_id**=*your_authorization_request_id*

   If the OAuth 2.0/OpenID provider is configured for a subrealm rather than the Top Level Realm, you must specify it in the endpoint. For example, if the OAuth 2.0/OpenID provider is configured for the `/alpha` realm, then use `/oauth2/realms/root/realms/alpha/access_token`.

   For example:

```
$ curl --request POST \
--header "authorization: Basic bXlDSUJBQ2xpZW50OmZvcmdlcm9jaw==" ❶\
--data "grant_type=urn:openid:params:grant-type:ciba" \
--data "auth_req_id=35Evy3bJXJEnhll2ebacgROYfbU" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/access_token"
```

❶ The basic authorization header is the base64-encoded value of your client ID, a colon character (**:**), and the client secret. For example `myCIBAClient:forgerock`.

For more information about authenticating clients, see "*OAuth 2.0 Client Authentication*" in the *OAuth 2.0 Guide*.

- If the user has authenticated and authorized the operation, AM returns an ID token and an access token. For example:

```
{
  "access_token": "z4mWG0cxqwPwgjj7srJ2Jdxe9ag",
  "id_token": "eyJ0eXAiOi...YA9Hoqwew",
  "token_type": "Bearer",
  "expires_in": 3599
}
```

Tip

AM can also issue refresh tokens at the same time the access tokens are issued. For more information, see "*Refresh Tokens*" in the *OAuth 2.0 Guide*.

- If the user has not yet authenticated and authorized the operation, AM returns an HTTP 400 response, as follows:

```
{
  "error_description": "End user has not yet been authenticated",
  "error": "authorization_pending"
}
```

The client should wait the number of seconds specified by the `interval` value that was returned when requesting the `auth_req_id`, and then resend the POST request. The default value for `interval` is two seconds.

- If the client does not wait for the interval before resending the request, AM returns an HTTP 400 response, as follows:

```
{
  "error_description": "The polling interval has not elapsed since the last request",
  "error": "slow_down"
}
```

2. (Optional) The relying party can request additional claims about the end user from AM.

   For more information, see "/oauth2/userinfo".

Implicit Grant

Endpoints

- "/oauth2/authorize" in the *OAuth 2.0 Guide*
- "/oauth2/userinfo"

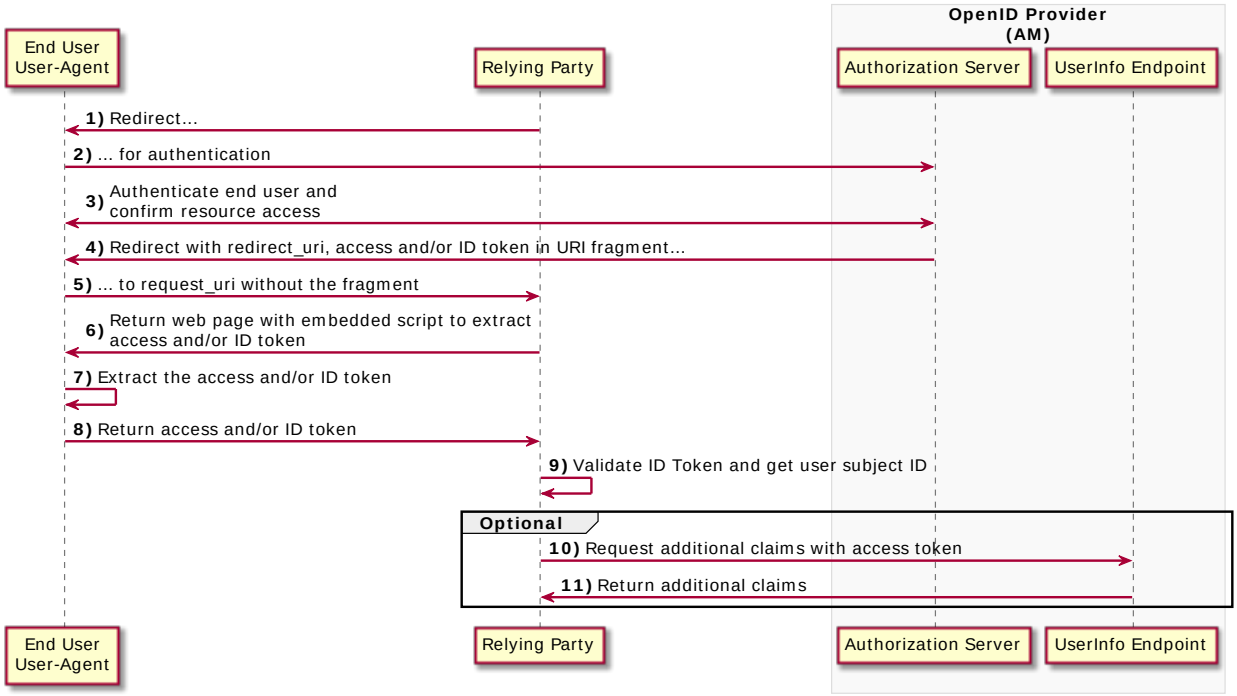
The OpenID Connect implicit grant is designed for public clients that run inside the end user's user-agent. For example, JavaScript applications.

This flow lets the relying party interact directly with the OpenID provider, AM, and receive tokens directly from the authorization endpoint instead of from the token endpoint.

Since applications running in the user-agent are considered less trusted than applications running in servers, the authorization server will never issue refresh tokens in this flow. Also, you must consider the security impact of cross-site scripting (XSS) attacks that could leak the ID and access tokens to other systems, and implement Cross-Origin Resource Sharing (CORS) to make OAuth 2.0/OpenID Connect requests to different domains.

Due to the security implications of this flow, it is recommended to use the Authorization Code grant with PKCE flow whenever possible.

OpenID Connect Implicit Flow



+ *Implicit Flow Explained*

The steps in the diagram are described below:

1. The relying party, usually a single-page application (SPA), receives a request to access user information stored in an OpenID provider. To access this information, the client requires authorization from the end user.

2. The relying party redirects the end user's user-agent or opens a new frame to the AM OpenID provider.

   As part of the OpenID Connect flow, the request contains the `openid` scope and the `nonce` parameter.

3. The OpenID provider authenticates the end user, confirms resource access, and gathers consent if not previously saved.

4. If the end user's credentials are valid, the authorization server returns an ID token (and optionally, an access token) to the user-agent as part of the redirection URI.

5. The user-agent must extract the token(s) from the URI. In this example, the user-agent follows the redirection to the relying party without the token(s)...
6. ... And the relying party returns a web page with an embedded script to extract the token(s) from the URI.

   In another possible scenario, the redirection URI is a dummy URI in the application running in the user-agent which already has the logic in itself to extract the tokens.
7. The user-agent executes the script and retrieves the tokens.
8. The user-agent returns the tokens to the relying party.
9. The relying party validates the ID token and its claims.

   Now, the relying party can use the ID token subject ID claim as the end user's identity.
10. The relying party may require more claims than those included in the ID token. In this case, it makes a request to the OpenID provider's `oauth2/userinfo` endpoint with the access token.
11. If the access token is valid, the `oauth2/userinfo` endpoint returns additional claims, if any.

    Now, the relying party can use the subject ID and the additional retrieved claims as the end user's identity.

Perform the steps in the following procedures to obtain an ID token and an access token:

- "To Obtain an ID/Access Token Using a Browser in the Implicit Grant"
- "To Obtain an ID/Access Token Without Using a Browser in the Implicit Grant"

To Obtain an ID/Access Token Using a Browser in the Implicit Grant

This example shows how to obtain an ID token and an access token. It adds notes on how to obtain the ID token only, as well.

The procedure assumes the following configuration:

- AM is configured as an OAuth 2.0/OpenID provider. Ensure that:
  - The `token` and `id_token` plugins are configured in the Response Type Plugins field.
  - The `Implicit Grant` grant type is configured in the Grant Types field.

  For more information, see "*OpenID Provider Configuration*".
- A *public* client called `myClient` is registered in AM with the following configuration:

- **Scopes**: `openid profile`
- **Response Types**: `token id_token`

  Configure `id_token` to receive an ID token only.
- **Grant Types**: `Implicit`
- **Authentication Method**: `none`
- **Token Endpoint Authentication Method**: `none`

  If you were using a confidential OpenID Connect client, you must specify a method to authenticate. For more information, see "*OpenID Connect Client Authentication*".

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

Perform the steps in this procedure to obtain an ID token and an access token using the Implicit grant:

1. The client makes a GET call to AM's authorization endpoint specifying, at least, the following parameters:

   - **client_id**=*your_client_id*
   - **response_type**=token id_token

     To obtain only an ID token, use `response_type=id_token` instead.
   - **redirect_uri**=*your_redirect_uri*
   - **nonce**=*your_nonce*
   - **scope**=openid profile

   For information about the parameters supported by the `/oauth2/authorize` endpoint, see "/oauth2/authorize" in the *OAuth 2.0 Guide*.

   If the OAuth 2.0/OpenID provider is configured for a subrealm rather than the Top Level Realm, you must specify it in the endpoint. For example, if the OAuth 2.0/OpenID provider is configured for the `/alpha` realm, then use `/oauth2/realms/root/realms/alpha/authorize`.

   For example:

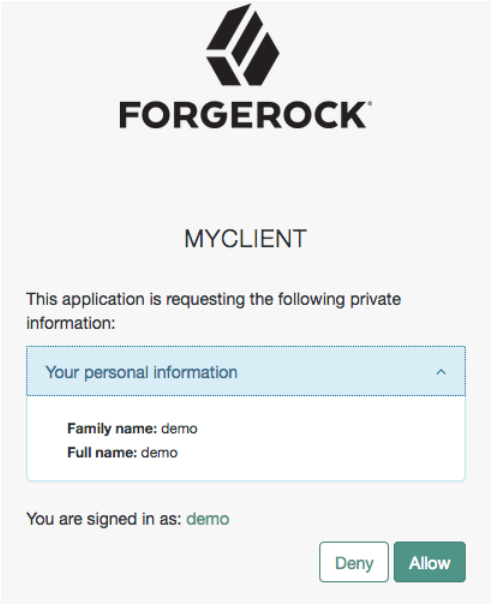
   ```
   https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/authorize \
   ?client_id=myClient \
   &response_type=token%20id_token \
   &scope=openid%20profile \
   &redirect_uri=https://www.example.com:443/callback \
   &state=abc123 \
   &nonce=123abc
   ```

Note that the URL is split for readability purposes and that the `state` parameter has been included to protect against CSRF attacks.

2. The end user logs in to AM, for example, using the credentials of the `demo` user. In this case, they log in using the default chain or tree configured for the realm.

   After logging in, AM presents the AM user interface consent screen:

Consent Screen



   Note that requesting the `profile` scope translates into requesting access to several claims. For more information about the special `profile` scope, see "*About Claims*".

3. The end user selects the `Allow` button to grant consent for the `profile` scope.

   AM redirects the resource owner to the URL specified in the `redirect_uri` parameter.

4. Inspect the URL in the browser. It contains an `access_token` and an `id_token` parameter with the tokens AM has issued. For example:

```
https://www.example.com:443/callback/
#access_token=pRbNamsGPv1T7NfAf5Dbx4AHM2c&id_token=eyJ0eXAiOiJKV1QiLCJra...7r8soMCk8A7QdQpg&state=123&token_typ
```

   If you only request an ID token, the response would not include the `access_token` parameter.

5. (Optional) The relying party can request additional claims about the end user from AM.

For more information, see "/oauth2/userinfo".

To Obtain an ID/Access Token Without Using a Browser in the Implicit Grant

This example shows how to obtain an ID token and an access token. It adds notes on how to obtain the ID token only, as well.

The procedure assumes the following configuration:

- AM is configured as an OAuth 2.0/OpenID provider. Ensure that:
  - The `token` plugin is configured in the Response Type Plugins field.
  - The `Implicit Grant` grant type is configured in the Grant Types field.

  For more information, see "*OpenID Provider Configuration*".
- A *public* client called `myClient` is registered in AM with the following configuration:
  - **Scopes**: `openid profile`
  - **Response Types**: `token id_token`
  - **Grant Types**: `Implicit`
  - **Authentication Method**: `none`

    Confidential OpenID Connect clients can use several methods to authenticate. For more information, see "*OpenID Connect Client Authentication*".

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

Perform the steps in this procedure to obtain an ID token and an access token using the Implicit grant:

1. The end user authenticates to AM, for example, using the credentials of the `demo` user. For example:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "X-OpenAM-Username: demo" \
--header "X-OpenAM-Password: Ch4ng31t" \
--header "Accept-API-Version: resource=2.0, protocol=1.0" \
'https://openam.example.com:8443/openam/json/realms/root/realms/alpha/authenticate'
{
    "tokenId":"AQIC5wM...TU30Q*",
    "successUrl":"/openam/console",
    "realm":"/alpha"
}
```

2. The client makes a POST call to the AM's authorization endpoint, specifying the SSO token of the `demo` in a cookie and, at least, the following parameters:

   - **client_id**=*your_client_id*
   - **response_type**=token id_token

     To obtain only an ID token, use `response_type=id_token` instead.
   - **redirect_uri**=*your_redirect_uri*
   - **nonce**=*your_nonce*
   - **scope**=openid profile
   - **decision**=allow
   - **csrf**=*demo_user_SSO_token*

   For information about the parameters supported by the `/oauth2/authorize` endpoint, see "/oauth2/authorize" in the *OAuth 2.0 Guide*.

   If the OAuth 2.0/OpenID provider is configured for a subrealm rather than the Top Level Realm, you must specify it in the endpoint. For example, if the OAuth 2.0/OpenID provider is configured for the `/alpha` realm, then use `/oauth2/realms/root/realms/alpha/authorize`.

   For example:

```
curl --dump-header - \
--Cookie "iPlanetDirectoryPro=AQIC5wM...TU3OQ*" \
--request POST \
--data "client_id=myClient" \
--data "response_type=token id_token" \
--data "scope=openid profile" \
--data "state=123abc" \
--data "nonce=abc123" \
--data "decision=allow" \
--data "csrf=AQIC5wM...TU3OQ*" \
--data "redirect_uri=https://www.example.com:443/callback" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/authorize"
```

   Note that the `state` parameter has been included to protect against CSRF attacks.

   If the authorization server is able to authenticate the user, it returns an HTTP 302 response with the access and ID tokens appended to the redirection URI:

```
HTTP/1.1 302 Found
Server: Apache-Coyote/1.1
X-Frame-Options: SAMEORIGIN
Pragma: no-cache
Cache-Control: no-store
Date: Mon, 04 Mar 2019 16:56:46 GMT
Accept-Ranges: bytes
Location: https://www.example.com:443/callback#access_token=az91IvnIQ-
uP3Eqw5QqaXXY_DCo&id_token=eyJ0eXAiOiJKV1QiLCJra...7r8soMCk8A7QdQpg&state=123abc&token_type=Bearer&expires_in=3
Vary: Accept-Charset, Accept-Encoding, Accept-Language, Accept
Content-Length: 0
```

If you only request an ID token, the response would not include the `access_token` parameter.

3. (Optional) The relying party can request additional claims about the end user from AM.

   For more information, see "/oauth2/userinfo".

Tip

For access to a sample JavaScript-based relying party to test the Implicit grant flow, see How do I access and build the sample code provided for AM (All versions)? in the *ForgeRock Knowledge Base*.

Clone the example project to deploy it in the same web container as AM. 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 "*Dynamic Client Registration*", and browse the deployment URL to see the initial page.

The example relying party validates the ID token signature using the default (HS256) algorithm, decodes de ID token to validate its contents and shows 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.

Hybrid Grant

Endpoints

- /oauth2/authorize in the *OAuth 2.0 Guide*
- "/oauth2/userinfo"

The OpenID Connect Hybrid grant is designed for clients that require flexibility when requesting ID, access, and refresh tokens.

Similar to the Authorization Code grant flow, the Hybrid grant flow is a two-step process:

1. The relying party makes a first request for tokens or codes. For example, a request for an ID token and an access code. AM returns them in the redirection fragment, as it does during the Implicit grant flow.

   The client relying party usually starts using these tokens immediately.

2. Some time after the first request has happened, the relying party makes a second request for additional tokens. For example, a request for an access token using the access code, or a request for a refresh token.

Important

Consider the following security tips when implementing this flow:

- Requesting an access token during the first step exposes the token in the redirection fragment, just like during the Implicit grant flow.

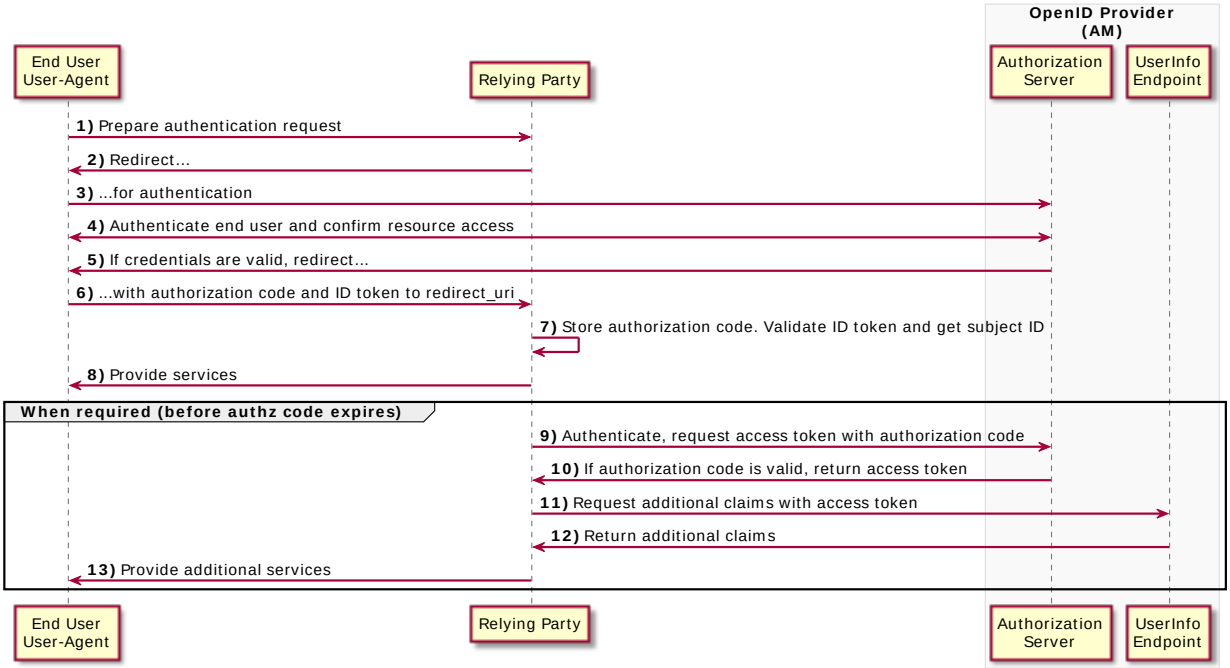
  Also, you must consider the security impact of cross-site scripting (XSS) attacks that could leak the ID and access tokens to other systems, and implement Cross-Origin Resource Sharing (CORS) to make OAuth 2.0/ OpenID Connect requests to different domains.

  Due to the security implications, ForgeRock recommends not to request access tokens during the first step of this flow.

- If the relying party is a public client, you can use the PKCE specification to mitigate against interception attacks performed by malicious users.

A common use case is the relying party requesting an ID token which can be used to, for example, pre-register the end user so they can start shopping. Only later and, if required, the relying party requests an access token to inquire the OpenID provider about additional claims. For example, during the check out, the relying party requests from AM the end user's address details.

OpenID Connect Hybrid Flow



+ *Hybrid Flow Explained*

The steps in the diagram are described below:

1. The end user wants to use the services provided by the relying party. The relying party, usually a web-based service, requires an account to provide those services.

   The end user issues a request to the relying party to access their information, which is stored in an OpenID provider.

2. To access the end user's information in the provider, the relying party requires authorization from the end user. Therefore, the relying party redirects the end user's user agent...

3. ... to the OpenID provider.

4. The OpenID provider authenticates the end user, confirms resource access, and gathers consent if not previously saved.

5. If the end user's credentials are valid, the OpenID provider redirects the end user to the relying party.

6. During the redirection process, the OpenID provider appends an authorization code and an ID token to the URL.

   Note that AM can return any combination of access token, ID token, and authorization code depending on the request. In this example, the access token is not requested at this time due to security concerns.

7. The relying party stores the authorization code for future use. It also validates the ID token and gets the subject ID.

8. With the ID token, the relying party starts providing services to the end user.

9. Later, but always before the authorization code has expired, the relying party requests an access token from the OpenID provider so it can access more information about the end user.

   A use case would be the end user requiring services from the relying party that requires additional (usually more sensitive) information. For example, the end user requests the relying party to compare their electricity usage and supplier information against offers in the market.

   If required, the relying party could also request a refresh token.

10. If the relying party credentials and the authorization code are valid, AM returns an access token.

11. The relying party makes a request to AM's `/oauth2/userinfo` endpoint with the access token to access the end user's additional claims.

12. If the access token is valid, the `/oauth2/userinfo` endpoint returns additional claims, if any.

13. The relying party can now use the subject ID in the ID token and the additional claims as the end user's identity to provide them with more services.

Perform the steps in the following procedure to obtain an authorization code and an ID token, and later an access token:

To Obtain an Authorization Code and an ID Token Using a Browser in the Hybrid Flow

This procedure assumes the following configuration:

- AM is configured as an OAuth 2.0/OpenID provider. Ensure that:

  - The `token`, `code`, and `id_token` plugins are configured in the Response Type Plugins field.

  - The `Authorization Code` grant type is configured in the Grant Types field.

  For more information, see "*OpenID Provider Configuration*".

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

- **Client secret**: `forgerock`
- **Scopes**: `openid profile`
- **Response Types**: `code id_token token`
- **Grant Types**: `Authorization Code`
- **Token Endpoint Authentication Method**: `client_secret_post`

  Confidential OpenID Connect clients can use several methods to authenticate. For more information, see "*OpenID Connect Client Authentication*".

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

Perform the steps in the following procedure to obtain an ID token and an authorization code that will later be exchanged for an access token:

1. The client redirects the end user's user-agent to the authorization server's authorization endpoint specifying, at least, the following form parameters:

   - **client_id**=*your_client_id*
   - **response_type**=code id_token

     As per the specification, you can request the following response types:

     - `code id_token`
     - `code token`
     - `code id_token token`

     Since AM returns the tokens in the redirection URL, requesting access tokens in this way poses a security risk.

   - **redirect_uri**=*your_redirect_uri*
   - **scope**=openid profile

   For information about the parameters supported by the `/oauth2/authorize` endpoint, see "/oauth2/authorize" in the *OAuth 2.0 Guide*.

   If the OAuth 2.0/OpenID provider is configured for a subrealm rather than the Top Level Realm, you must specify it in the endpoint. For example, if the OAuth 2.0/OpenID provider is configured for the `/alpha` realm, then use `/oauth2/realms/root/realms/alpha/authorize`.

   For example:

```
https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/authorize \
?client_id=myClient \
&response_type=code%20id_token \
&scope=openid%20profile \
&state=abc123 \
&nonce=123abc \
&redirect_uri=https://www.example.com:443/callback
```

Note that the URL is split and spaces have been added for readability purposes. The `state` and `nonce` parameters have been included to protect against CSRF and replay attacks.

Tip

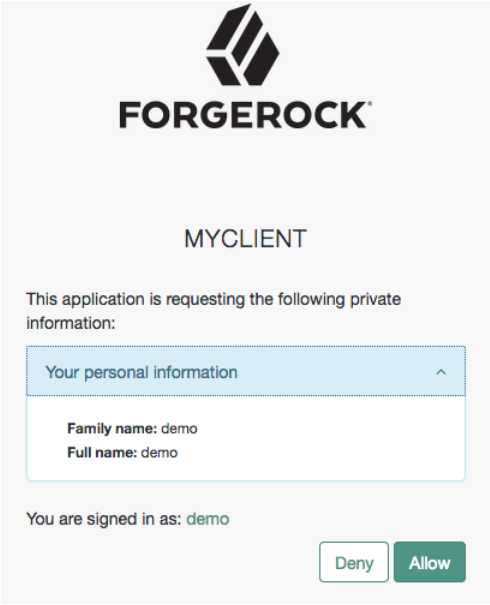
Implement the PKCE specification to mitigate against interception attacks performed by malicious users.

For more information about the required additional parameters and an example, see "Authorization Code Grant with PKCE".

2. The end user authenticates to AM, for example, using the credentials of the `demo` user. In this case, they log in using the default chain or tree configured for the realm.

   After logging in, AM presents its consent screen:

OpenID Connect Consent Screen



Note that requesting the `profile` scope translates into requesting access to several claims. For more information about the special `profile` scope, see "*About Claims*".

3. The end user selects the `Allow` button to grant consent for the `profile` scope.

   AM redirects the end user to the URL specified in the `redirect_uri` parameter.

4. Inspect the URL in the browser. It contains a `code` parameter with the authorization code and an `id_token` parameter with the ID token AM has issued. For example:

   ```
   https://www.example.com:443/
   callback#code=bOrAijEerd_YdNCUC1piL5VfNO4&id_token=eyJ0eXAiOiJKV1QiLCJra...7r8soMCk8A7QdQpg
   ```

   The client relying party can now use the ID token as the end user's identity and store the access code for later use.

5. (Optional) The client exchanges the authorization code for an access token (and maybe, an refresh token). Perform the steps in one of the following procedures:

   - "To Exchange an Authorization Code for an ID/Access Token in the Authorization Code Grant Flow"
   - "To Exchange an Authorization Code for an ID/Access Token in the Authorization Code Grant with PKCE Flow"

To Obtain an Authorization Code and an ID Token Without Using a Browser in the Hybrid Flow

This procedure assumes the following configuration:

- AM is configured as an OAuth 2.0/OpenID provider.

  For more information, see "*OpenID Provider Configuration*".

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

  - **Client secret**: `forgerock`
  - **Scopes**: `openid profile`
  - **Response Types**: `code id_token token`
  - **Grant Types**: `Authorization Code`
  - **Token Endpoint Authentication Method**: `client_secret_post`

    Confidential OpenID Connect clients can use several methods to authenticate. For more information, see "*OpenID Connect Client Authentication*".

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

Perform the steps in the following procedure to obtain an ID token and an authorization code that will later be exchanged for an access token:

1. The end user logs in to AM, for example, using the credentials of the `demo` user. For example:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "X-OpenAM-Username: demo" \
--header "X-OpenAM-Password: Ch4ng31t" \
--header "Accept-API-Version: resource=2.0, protocol=1.0" \
'https://openam.example.com:8443/openam/json/realms/root/realms/alpha/authenticate'
{
    "tokenId":"AQIC5wM...TU3OQ*",
    "successUrl":"/openam/console",
    "realm":"/alpha"
}
```

2. The client makes a POST call to AM's authorization endpoint, specifying the SSO token of the `demo` in a cookie and, at least, the following parameters:

   - **client_id**=*your_client_id*
   - **response_type**=code id_token

     As per the specification, you can request the following response types:

     - `code id_token`
     - `code token`
     - `code id_token token`

     Since AM returns the tokens in the redirection URL, requesting access tokens in this way poses a security risk.

   - **redirect_uri**=*your_redirect_uri*
   - **scope**=openid profile
   - **decision**=allow
   - **csrf**=*demo_user_SSO_token*

   For information about the parameters supported by the `/oauth2/authorize` endpoint, see "/oauth2/authorize" in the *OAuth 2.0 Guide*.

   If the OAuth 2.0/OpenID provider is configured for a subrealm rather than the Top Level Realm, you must specify it in the endpoint. For example, if the OAuth 2.0/OpenID provider is configured for the `/alpha` realm, then use `/oauth2/realms/root/realms/alpha/authorize`.

   For example:

```
$ curl --dump-header - \
--request POST \
--Cookie "iPlanetDirectoryPro=AQIC5wM...TU3OQ*" \
--data "scope=openid profile" \
--data "response_type=code id_token" \
--data "client_id=myClient" \
--data "csrf=AQIC5wM...TU3OQ*" \
--data "redirect_uri=https://www.example.com:443/callback" \
--data "state=abc123" \
--data "nonce=123abc" \
--data "decision=allow" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/authorize"
```

Note that the `state` and `nonce` parameters have been included to protect against CSRF and replay attacks.

Tip

Implement the PKCE specification to mitigate against interception attacks performed by malicious users.

For more information about the required additional parameters and an example, see "Authorization Code Grant with PKCE".

If AM is able to authenticate the user and the client, it returns an HTTP 302 response with the authorization code appended to the redirection URL:

```
HTTP/1.1 302 Found
Server: Apache-Coyote/1.1
X-Frame-Options: SAMEORIGIN
Pragma: no-cache
Cache-Control: no-store
Date: Mon, 30 Jul 2018 11:42:37 GMT
Accept-Ranges: bytes
Location: https://www.example.com:443/
callback#code=bOrAijEerd_YdNCUC1piL5VfNO4&id_token=eyJ0eXAiOiJKV1QiLCJra...7r8soMCk8A7QdQpg
Vary: Accept-Charset, Accept-Encoding, Accept-Language, Accept
Content-Length: 0
```

The client relying party can now use the ID token as the end user's identity and store the access code for later use.

3. (Optional) The client exchanges the authorization code for an access token (and maybe, a refresh token). Perform the steps in one of the following procedures:

   - "To Exchange an Authorization Code for an ID/Access Token in the Authorization Code Grant Flow"
   - "To Exchange an Authorization Code for an ID/Access Token in the Authorization Code Grant with PKCE Flow"

Chapter 7

Managing OpenID Connect User Sessions

Logging in to the OpenID provider and obtaining tokens are well-stabilized processes in the OpenID specification. However, keeping the relying party informed of the session's validity is not as straightforward. The end user's session in AM is unavailable to the relying party, and therefore, the only information the relying party has is the expiration time of the ID token, which may be undesirable.

To solve this problem, AM supports different OpenID Connect specifications:

OpenID Connect Session Management Relying parties can request session information from AM, and act on it. For example, they can request the user to log in. Relying parties can also request AM to log out a user.	OpenID Connect Backchannel Logout AM sends a logout token to the relevant relying parties when a user session linked to an ID token has become invalid.

Checking the State of a Session, and Invalidating Sessions

The OpenID Connect Session Management 1.0 draft series defines a mechanism for relying parties to:

- Request the providers to confirm if a specific OpenID Connect session is still valid or not when presented with an ID token.
- Request session termination for a user in the provider. For example, if the user decides to log out.

To keep the process transparent to the end user, relying parties use hidden *iframes* to request session status from the providers and take actions based on it.

To link ID tokens to sessions and, therefore, let relying parties query AM for session status, AM creates a token in the CTS store with the link information. This also ensures that any AM instance can retrieve session information for a particular token.

Session management is enabled by default. To disable it, go to Realms > *Realm Name* > Services > OAuth2 Provider > Advanced OpenID Connect, and disable OIDC Session Management. Note that this will disable backchannel logout as well.

AM supports Draft 05 and Draft 10 of the specification. They use different endpoints and to achieve the same end result, as you will see next.

- "Session Management Draft 10"
- "Session Management Draft 05"

Session Management Draft 10

Draft 10 does not specify or mandate any session-related endpoint. Therefore, AM's implementation of Draft 10 is as follows:

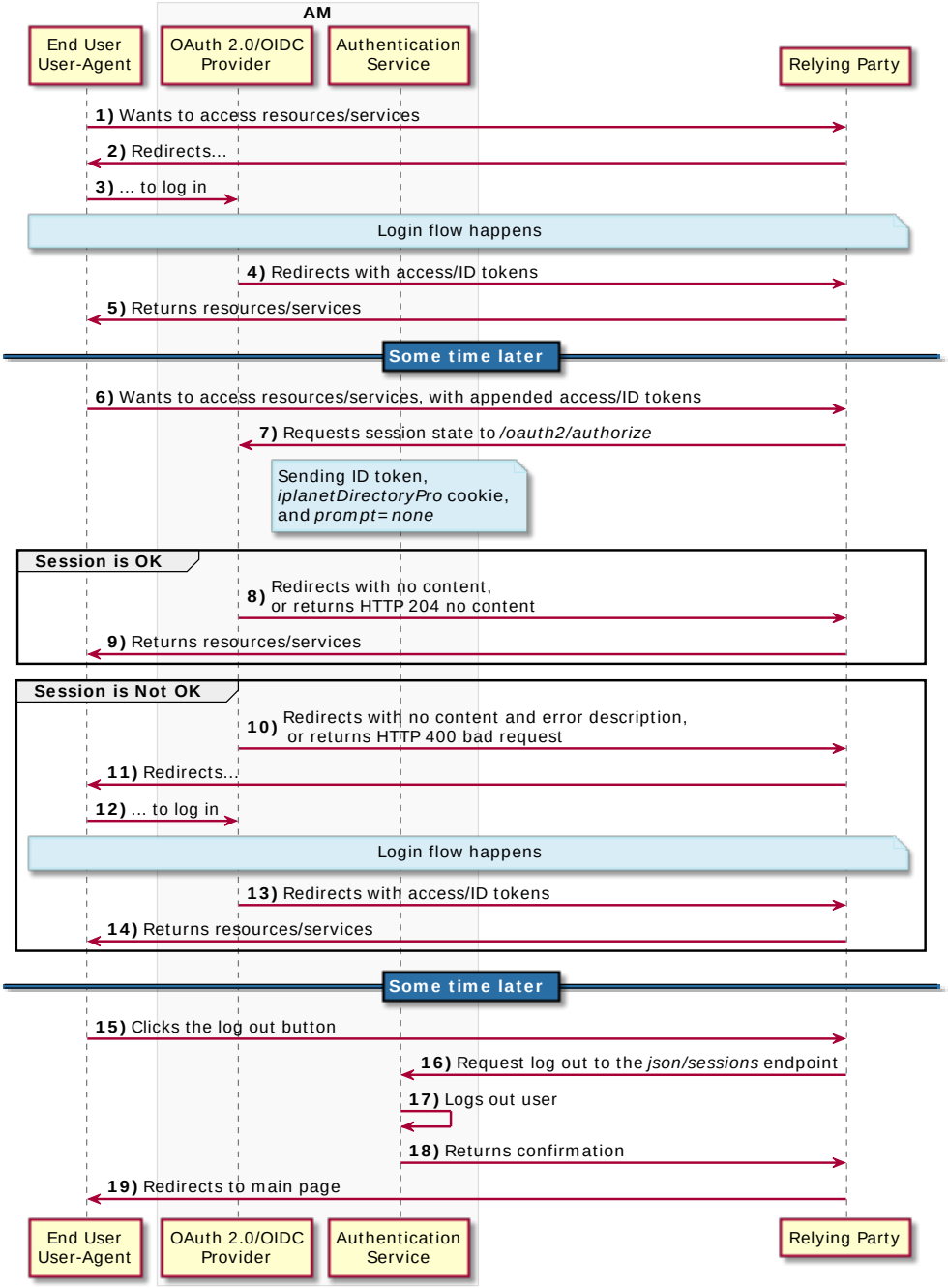
- `/oauth2/authorization`. Retrieves session state for requests.
- `/json/sessions`. (AM-specific). Logs out end users.

  For more information, see "Logging out of AM Using REST" in the *Authentication and Single Sign-On Guide*.

The simplest strategy to check session state using the authorization endpoint is to create an iframe whose `src` attribute is AM's `/oauth2/authorize` endpoint with the required parameters. Note that you must also include any other parameter required in your environment, such as client authentication methods.

For AM to validate an end user session against an ID token, the user-agent must provide the SSO token of the end user's session as the `iplanetDirectoryPro` cookie. Therefore, the flow would resemble the following:

Session Management Flow



The following is an example of a public client requesting session state:

```
https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/authorize \
?client_id=myClient \
&response_type=none \
&id_token_hint=eyJ0eXAiOiJKV1QiLCJra...7r8soMCk8A7QdQpg \
&redirect_uri=https://www.example.com:443/callback \
&prompt=none
```

- `prompt=none`. Specifies that the request is a repeated authentication request for a specific end user. AM will not display any user interaction pages to the end user.
- `id_token_hint=`*`your_ID_token`*. Specifies the ID token associated to an end user. AM validates the ID token against the user's session.
- `response_type=none`. Specifies that AM should not issue any token as response.

Note that the URL is split for readability purposes and that the end user's SSO token must be set in a cookie in order to validate the session.

If the session is valid and the request contains a redirection URI, AM redirects to the specified URI with no content. If the request does not contain a redirection URI, AM returns an HTTP 204 no content message.

If the session is invalid and the request contains a redirection URI, AM redirects to the specified URI with no content and appends an `error_description` parameter to the URL. For example:

```
https://www.example.com:443/callback?error_description=The%20request%20requires
%20login.&error=login_required
```

If the request does not contain a redirection URI, AM returns an HTTP 400 error message and redirects to an AM console page showing a message, such as `login required. The request requires login`.

The relying party's iframe, or the redirection page, should be able to retrieve the error messages and act in consequence. For example, redirecting the end user to a login page.

Enabling Session Management Draft 10

To let clients use session management draft 10 with AM, perform the following steps:

1. Configure the provider:
   a. Go to Realms > *Realm Name* > Services > OAuth2 Provider.
   b. On the Core tab, add the `none|org.forgerock.oauth2.core.NoneResponseTypeHandler` plugin in the Response Type Plugins field, if it is not already present.
   c. Save your changes.
   d. On the Advanced OpenID Connect tab, enable OIDC Session Management, if it is not already enabled.

   e. Save your changes.

2. Configure the clients:

   a. Go to Realms > *Realm Name* > Applications > OAuth 2.0 > Clients > *Client Name* > OpenID Connect > Advanced.

   b. In the Response Types field, add the `none` type.

   c. Save your changes.

Session Management Draft 05

Draft 05 of the of the Session Management specification defined two endpoints for managing OpenID sessions. These endpoints have been removed from version 10 of the draft, but AM still supports them:

- `/oauth2/connect/checkSession`. It lets clients retrieve session state. This endpoint serves the `check_session_iframe` URL that allows the relying party to interact with the endpoint.

  When checking session state with the check session endpoint, you must configure the Client Session URI field in the client profile as the iframe URL in the relying party.

  For an alternative method of checking session state compliant with version 10 of the session management draft, see "Session Management Draft 10".

- `/oauth2/connect/endSession`. It lets clients terminate end user sessions and redirect end users to a particular page after logout.

For more information about the endpoints, see "*OpenID Connect 1.0 Endpoints*".

Informing Relying Parties that a Session has Expired

OpenID Connect Back-Channel Logout 1.0 Draft 06 defines how a provider can send a *logout token* to the relevant relying parties when an end user session linked to an ID token becomes invalid.

When back-channel logout is enabled, AM sends a logout token to a URL configured in the relying party's client profile. This URL must be a page or application in the relying party that is capable of dealing with the token.

AM stores a list of logged in clients in the end user's session so that, when it becomes invalid, AM has a list of URLs to which it needs to send the logout token to.

This is particularly important in scenarios where different relying parties use the same user session to obtain ID tokens. By storing the URLs in the session itself, AM ensures that all the related relying parties receive a logout token.

Next, the relying party validates the logout token, and clears any state associated with the combination of session ID, user, and issuer. Finally, it sends a response to AM with the outcome of the logout, as explained in *2.8 Back-Channel Logout Response* of the draft.

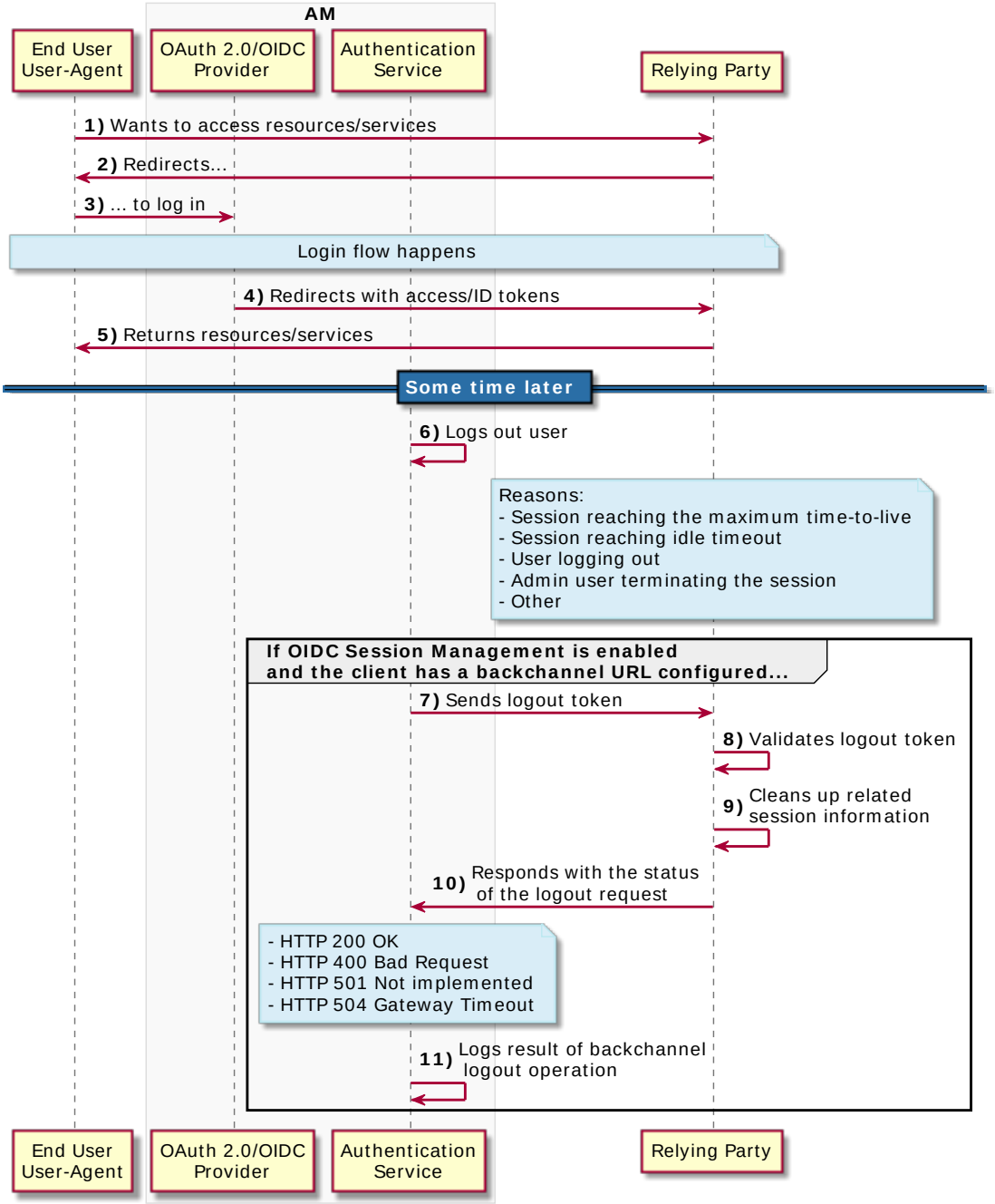
Depending on which status code AM receives, it logs an audit event of the type `AM-BACK-CHANNEL-LOGOUT` in the activity log file, which resembles the following:

```
{
  "_id":"cb52bc45-549d-4a9c-86cc-20d7500e333b-96127",
  "eventName":"AM-BACK-CHANNEL-LOGOUT","transactionId":"cb52bc45-549d-4a9c-86cc-20d7500e333b-94750",
  ...
  ...
  "operation":"Sent logout request to https://rp.example.com:8443/logout, which responded with HTTP code
 200.",
  ...
}
```

AM will log the HTTP code that the relying party returns, or an error if there is no response before the request times out.

The following simplified diagram illustrates a possible back-channel logout flow:

Back-Channel Logout Flow



Back-Channel Logout Limitations

The current implementation of back-channel logout in AM has the following limitations:

- It is only supported for CTS-based sessions.
- AM currently only supports back-channel logout when acting as the provider.

The Logout Token

The logout token is defined in section *2.4 Logout Token* of the draft. The following is an example logout token issued by AM, and the description of its claims:

```
{
  "aud": "backchannelConfidentialClient",  ❶
  "sub": "(usr!ForgerockDemo)",  ❷
  "auditTrackingId": "cb52bc45-549d-4a9c-86cc-20d7500e333b-91288",  ❸
  "iss": "https://openam.example.com:8443/openam/oauth2/backchannelSubRealm",  ❹
  "cause": "CLIENT_LOGOUT",  ❺
  "iat": 1614005410,  ❻
  "jti": "1cd8805d-6fc0-4699-a33f-a75d45b24e9e",❼
  "events": {  ❽
    "http://schemas.openid.net/event/backchannel-logout": {}
  },
  "sid": "mTNo042FCiPkgAJKjdjgCvBWvVYTB1d+zreDBnZAqvM="  ❾
}
```

❶ Specifies the audience of the logout token. In this case, the client that requested the ID token(s) related to the user that has been logged out.

❷ Specifies the subject of the logout token. In this case, the user that has been logged out.

The subject of the logout token matches the subject of the ID token(s).

+ *About the Subject Claim*

The subject claim is in the format `(type!subject)`, where:

- `subject` is the identifier of the user/identity, or the name of the OAuth 2.0/OpenID Connect client that is the subject of the token.
- `type` can be one of the following:
  - `age`. Specifies that the *subject* is an OAuth 2.0/OpenID Connect-related user-agent or client. For example, an OAuth 2.0 client, a Remote Consent Service agent, and a Web and Java Agent internal client.
  - `usr`. Specifies that the *subject* is a user/identity.

For example, `(usr!demo)`, or `(age!myOAuth2Client)`.

❸ (AM-specific) Determines the unique audit identifier for this token.
❹ Specifies the authorization server that issued the logout token.
❺ (AM-specific) Documents the reason why the user was logged out, if known. Possible values are:

- `CLIENT_LOGOUT`. AM has received a call to the sessions endpoint in the *Authentication and Single Sign-On Guide* to end the session.
- `SESSION_TERMINATION`. An administrative user has terminated the session.
- `SESSION_MAX_TIMEOUT`. AM terminated the session because it reached its maximum time-to-live.
- `SESSION_IDLE_TIMEOUT`. AM terminated the session because it reached its maximum idle time.

If the reason is not known, the claim does not show in the token.
❻ Specifies the creation time of the logout token.
❼ Specifies the unique identifier for the logout token.
❽ Specifies a JSON object that contains the `http://schemas.openid.net/event/backchannel-logout` URL, which marks the JWT as a logout token. The value of the object is always an empty JSON object (**{}**).
❾ Specifies a session ID that identifies the relying party's session with the provider.

The `sid` in the logout token matches the `sid` in the related ID token, and therefore, the relying party can match both when doing session clean up operations.

If a relying party requests several ID tokens using the same end user session, they will all share the same `sid`.

However, if several relying parties use the same user session to obtain ID tokens, the `sid` in them will be different. When the end user's session becomes invalid, AM will send logout tokens to all the relying parties involved.

Note that the claim is only populated in the logout token if Backchannel Logout Session Required is enabled in the client profile. See "Enabling Back-Channel Logout".

Enabling Back-Channel Logout

To enable back-channel logout, first configure the OAuth 2.0 provider for the realm, and next configure the clients that will use the feature:

1. Configure the OAuth 2.0 provider:

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

   b. Enable OIDC Session Management, if it is not already enabled.

> **Note**
>
> OIDC Session Management is enabled by default, and it is also required for Session Management.
>
> When enabled, AM will always add a `sid` to the ID tokens, regardless of whether clients have logout URLs configured in them or not.

c. Save your changes.

d. (Optional) Review the logout token signing secret.

   AM signs logout tokens with the same secret it uses to sign ID Tokens. For more information, see Secret ID Mappings for Signing OpenID Connect Tokens in the *Security Guide*.

e. (Optional) Configure AM to encrypt the logout tokens.

   Encryption is disabled by default. To enable it, you must configure ID token encryption as well. For more information, see "Encrypting ID Tokens and Backchannel Logout Tokens".

2. Configure the clients:

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

   b. In the Back Channel Logout URI field, set the URL in the relying party to where AM will send the logout token during back-channel logout.

      This URL can use the http or the https schemes, and may contain a port, a path, or query parameters, depending on the implementation of the relying party. For example, `https://my-rp.example.com:8443/logout`.

   c. (Optional) If the logout token must contain the session ID (`sid`), enable Backchannel Logout Session Required.

   d. Save your changes, and configure as many clients as required.

   > **Tip**
   >
   > Clients registering dynamically can provide their back-channel logout configuration, too.

The Back-Channel Logout Postman Collection

ForgeRock provides a back-channel logout Postman collection to try out the functionality. The source for the REST calls, including the prerequisites needed to run the collection, is provided as a downloadable JSON file collection.

Back-channel logout relies on a relying party that can acknowledge the logout token and send a response back to AM. To emulate this, you can send a mock server in Postman.

Perform the following steps to download, configure, and run the back-channel logout Postman collection:

1. Download and install Postman.

2. Download the ForgeRock OpenID Connect Back-Channel Logout Collection.

3. Import the collection in Postman:

   a. Go to File > Import ... > Upload Files.

   b. Select the collection you downloaded, and click Open. Then, click Import.

4. (Optional) Create a Postman mock server. Follow the instructions in the Setting Up Mock Servers in the *Postman Documentation*.

   The mock server will work as the relying party. Inspect the requests sent to the mock server to see the logout tokens sent by AM.

   When you create a mock server, Postman presents you with some information and the URL you can use to call it. For example, `https://f4a08510-2f95-4990-8cb0-5a4f281a8bac.mock.pstmn.io`.

   Save this URL; you need to configure it in the following step of this procedure.

5. Configure the collection's variables to suit your environment:

   a. In Postman, on the Collections tab, select the ForgeRock OpenID Connect Back-Channel Logout Collection. Click on the **...** button, and then on Edit.

      The Edit Collection page appears.

   b. Click on the Variables tab, and change at least the value of the following variables:

      - `URL_base`

        Configure the URL of your AM environment. For example, `https://openam.example.com:8443/openam`.

      - `admin_password`

        Configure the password of the administrative user, such as `amAdmin`, that the collection will use to create AM configuration objects.

      - `back_channel_logout_uri`

        Configure the back-channel logout URL of your relying party, or the URL of the Postman mock server.

c. Click Update to save your changes.

You are ready to start running the collection.

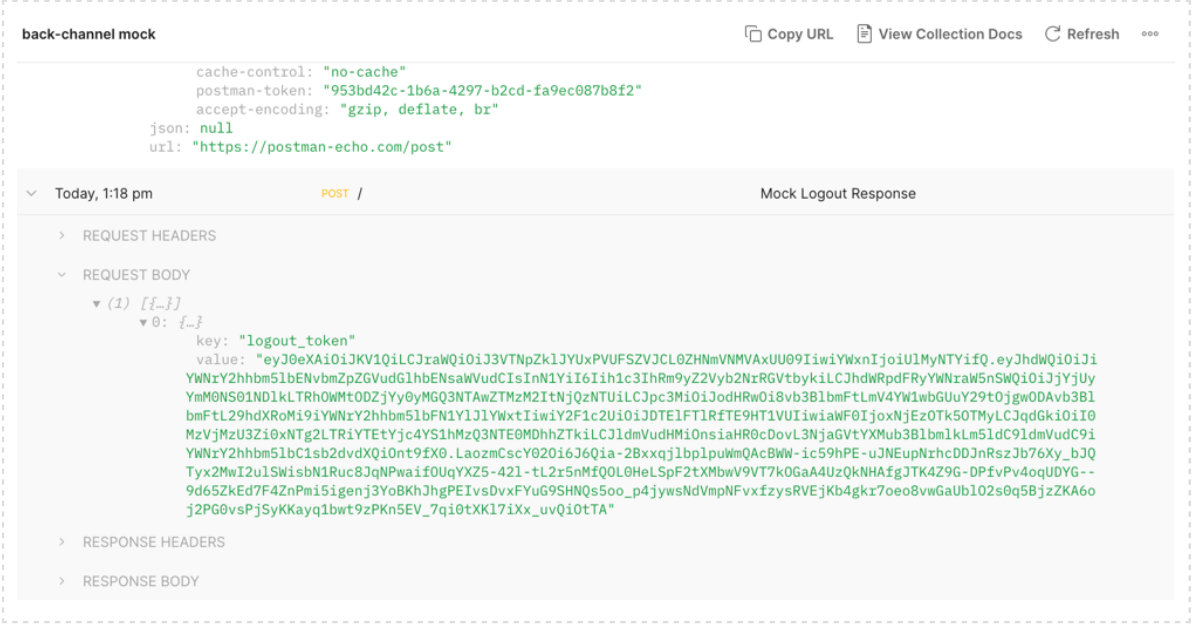
The collection is divided into the following folders:

- `Prerequisites`, containing REST calls to configure a new realm containing an authorization server, and to create the clients and users required to run the collection.
- `OpenID Connect Back-Channel Logout Flow`, containing REST calls for the authorization code grant flow, and to log out the demo user.
- `Mock Response`, containing a REST call to send AM a response when using the mock server.

6. Run the collection.

7. (Optional) (Mock server only) On the mockup server window, check the request body of the `Mock Up Response` to see the logout token.

+ *Example Postman Mock Up Server Window with the Response Body Expanded*



8. Inspect the contents of the logout token.

Tokens created by the collection are not encrypted.

9. Open the AM activity audit log.

Check for an entry with event name `AM-BACK-CHANNEL-LOGOUT` with the logout request, and the relying party's response. For example:

```
{
 "_id":"cb52bc45-549d-4a9c-86cc-20d7500e333b-96127",
 "eventName":"AM-BACK-CHANNEL-LOGOUT","transactionId":"cb52bc45-549d-4a9c-86cc-20d7500e333b-94750",
 ...
 ...
 "operation":"Sent logout request to https://f4a08510-2f95-4990-8cb0-5a4f281a8bac.mock.pstmn.io, which
 responded with HTTP code 200."
 ...
 }
```

Chapter 8

Adding Authentication Requirements to ID Tokens

Relying parties may require end users to satisfy different rules or conditions when authenticating to the provider. Consider the case of a financial services provider. While authenticating with username and password may be acceptable to create an account, accessing the end user's bank account details may require multi-factor authentication.

By specifying an authentication context reference (acr) or an authentication module reference (amr) claim in the request, relying parties can require that AM authenticate users using specific chains, modules, or trees.

The following sections show how AM implements both claims, and how to configure AM to honor them:

- "The Authentication Context Class Reference (acr) Claim"
- "The Authentication Method Reference (amr) Claim"

The Authentication Context Class Reference (acr) Claim

In the OpenID Connect specification, the `acr` claim identifies a set of rules the user must satisfy when authenticating to the OpenID provider. For example, a chain or tree configured in AM.

To avoid exposing the name of authentication trees or chains, AM implements a map that consists of a key (the value that is included in the `acr` claim) and the name of the authentication tree or chain.

The specification indicates that the `acr` claim is a list of authentication contexts; AM honors the first value in the list for which it has a valid mapping. For example, if the relying party requests a list of acr values such as `acr-1 acr-2 acr-3` and only `acr-2` and `acr-3` are mapped, AM will always choose `acr-2` to authenticate the end user.

The `acr` claim is optional and therefore is not added to ID tokens by default, but you can request AM to include it by specifying it as a voluntary or essential claim:

+ *Voluntary Claim*

Request voluntary `acr` claims when the fact that the user has authenticated to a specific chain or tree would improve the user experience in the OpenID Connect flow, but it is not a requisite.

You can request voluntary `acr` claims in the following ways:

- Specifying the authentication chains or trees in the `acr_values` parameter when requesting an ID token to the `/oauth2/authorize` endpoint.
- Specifying the authentication chains or trees in JSON format in the `claims` parameter when requesting an ID token to the `/oauth2/authorize` endpoint.

If the end user is already authenticated to the first value on the list for which AM has a mapping, AM does not force the user to reauthenticate. If they are not already authenticated, or if they are authenticated to any other tree or chain on the list, AM uses the first value for which it has a valid mapping to authenticate them.

Consider an example where the relying party requests a list of acr values, such as `acr-1 acr-2 acr-3`, and AM only has `acr-2` and `acr-3` mapped:

- AM will not force the end user to reauthenticate if they are already authenticated to `acr-2` (which is the first value in the list for which AM has a mapping).
- AM will force the end user to reauthenticate to `acr-2` in the following cases:
  - If the end user has authenticated to `acr-3`.
  - If the end user has authenticated to any other tree or chain.
  - If the end user has not yet authenticated.

If the user reauthenticates to a tree, AM destroys the original session and provides them with a new one that reflects the new authentication journey.

If the user reauthenticates to a chain, AM updates the original session to reflect the new authentication journey.

If the relying party requests authentication chains or trees that are not mapped in AM as valid acr values, AM continues the grant flow. The resulting ID token will contain an `acr` claim with the following values:

- **0 (zero)**, if the client authenticated to AM using a chain or tree that is not mapped to an acr value.
- ***acr_key_string***, if the client authenticated to AM using a chain or tree that is mapped to an acr value.

  If the end user authenticated to more than one chain or tree, AM will use the last chain or tree, provided it is mapped to an acr value.

+ *Essential Claim*

Request essential `acr` claims when the user must authenticate to a specific chain or tree to complete an OpenID Connect flow.

To request essential `acr` claims, specify the required authentication chains or trees in JSON format in the `claims` parameter when requesting an ID token to the `/oauth2/authorize` endpoint.

AM will always force the end user to authenticate to the first value in the list for which AM has a mapping, even if the end user already authenticated using the same chain or tree.

Consider an example where the relying party requests a list of acr values, such as `acr-1 acr-2 acr-3`, and AM only has `acr-2` and `acr-3` mapped:

AM will force the end user to authenticate to `acr-2` in the following cases:

- If the end user has authenticated to either `acr-2` or `acr-3`.
- If the end user has authenticated to any other chain or tree.
- If the end user is not authenticated.

If the user reauthenticates to a tree, AM destroys the original session and provides them with a new one that reflects the new authentication journey.

If the user reauthenticates to a chain, AM updates the original session to reflect the new authentication journey.

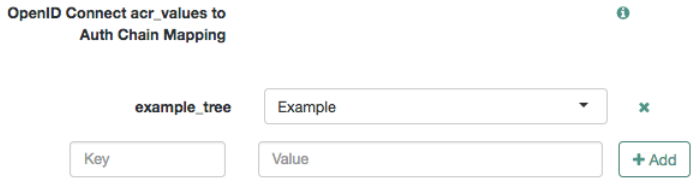
This mechanism can be used to perform step-up authentication, but AM does not consider if, for example, the authentication level of the current session is higher than the one achievable with the requested tree or chain.

If the relying party requests authentication chains or trees that are not mapped in AM as valid acr values, AM returns an error and redirects to the `redirect_uri` value, if available.

Perform the steps in the following procedure to configure AM to honor `acr` claims:

To Configure AM for the acr Claim

1. In the AM console, go to Realms > *Realm Name* > Services > OAuth2 Provider > Advanced OpenID Connect.
2. (Optional) To request `acr` claims using the `claims` parameter, enable "claims_parameter_supported".
3. In the OpenID Connect acr_values to Auth Chain Mapping box, map an identifier (the key in the map) to an authentication chain or tree. For example:



The identifier is the string that AM will add in the `acr` claim.

4. Save your changes.

5. (Optional) In the Default ACR values field, specify the identifiers of the authentication trees or chains AM should use to authenticate end users when acr values are not specified in the request. AM treats acr values specified in this field as voluntary claims.

   Acr values specified in the request will override the default values.

   If a request does not specify acr values and the Default ACR values field is empty, AM authenticates the end user with the default authentication chain or tree defined for the realm where the OAuth 2.0 provider is configured.

   Tip

   Query the `/oauth2/.well-known/openid-configuration` endpoint to determine the acr values supported by the OpenID provider. Mapped acr values are returned in the `acr_values_supported` object.

6. Save your changes.

7. (Optional) Review the "Requesting acr Claims Example".

Requesting acr Claims Example

This example assumes the following configuration:

- An authentication tree called `Example` is configured in AM. For more information, see "Configuring Authentication Trees" in the *Authentication and Single Sign-On Guide*.
- The `Example` tree is mapped to the `example_tree` identifier in the acr_value map of the OAuth 2.0 provider. For more information, see "To Configure AM for the acr Claim".
- A public client called `myClient` is registered in AM with the following configuration:
  - **Scopes**: `openid profile`
  - **Response Types**: `token id_token`
  - **Grant Types**: `Implicit`

- AM is configured as an OAuth 2.0/OpenID Provider.

Perform the following steps to request acr claims:

1. Log in to AM, for example, as the `demo` user. Ensure you are not using the `Example` tree to log in, and note the value of the `iplanetDirectoryPro` cookie.

2. In a new tab of the same browser, request an ID token using, for example, the "Implicit Grant" flow. Perform one of the following steps:

   a. Request **voluntary claims** with the `acr_values` parameter. For example:

   ```
   https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/authorize? \
   client_id=myClient \
   &response_type=id_token \
   &scope=openid%20profile \
   &redirect_uri=https://www.example.com/callback \
   &acr_values=example_chain%20example_tree \
   &nonce=abc123 \
   &prompt=login \
   &state=123abc
   ```

   Note that the URL is split for readability purposes and that the `prompt=login` parameter has been added. In most cases, this parameter is not required with the current implementation of `acr` claims, but it is recommended to add it for compliance with the specification when you need to force the user to re-authenticate.

   For information, see the prompt parameter in the *OAuth 2.0 Guide*.

   b. Request **voluntary claims** with the `claims` parameter.

   The `claims` parameter expects a JSON object, such as:

   ```
   {"id_token":{"acr":{"values":["example_chain","example_tree"]}}}
   ```

   For example:

   ```
   https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/authorize? \
   client_id=myClient \
   &response_type=id_token \
   &scope=openid%20profile \
   &redirect_uri=https://www.example.com/callback \
   &nonce=abc123 \
   &state=123abc \
   &prompt=login \
   &claims=%7B%22id_token%22%3A%7B%22acr%22%3A%7B%22values%22%3A%5B%22example_chain%22%2C
   %22example_tree%22%5D%7D%7D%7D
   ```

   Note that the URL is split for readability purposes, and that the JSON value of the `claims` parameter is URL encoded.

   c. Request **essential claims** with the `claims` parameter.

   The `claims` parameter expects a JSON object, such as:

```
{"id_token":{"acr":{"essential":true,"values":["example_chain","example_tree"]}}}
```

If `"essential":true` is not included in the JSON, AM assumes the acr request is voluntary.

For example:

```
https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/authorize? \
client_id=myClient \
&response_type=id_token \
&scope=openid%20profile \
&redirect_uri=https://www.example.com/callback \
&nonce=abc123 \
&state=123abc \
&prompt=login \
&claims=%7B%22id_token%22%3A%7B%22acr%22%3A%7B%22essential%22%3Atrue%2C%22values%22%3A%5B
%22example_chain%22%2C%22example_tree%22%5D%7D%7D%7D
```

Note that the URL is split for readability purposes, and that the JSON value of the `claims` parameter is URL encoded.

AM redirects to the Example tree. Note that the new URL contains the following parameters:

- `authIndexValue`, the value of which is the `Example` tree.
- `goto`, the value of which is the URL the UI will use to return to the authorization endpoint once the authentication flow has finished.
- `acr_sig`, the value of which is a unique string that identifies this particular `acr` request.

3. Log in again as the `demo` user. Note that the value of the `iplanetDirectoryPro` cookie changes to reflect the new session.

   AM redirects back to the authorization endpoint, and shows you the OAuth 2.0 consent page. Grant consent by selecting `Allow`. AM redirects you now to the URI specified by the `redirect_uri` parameter with an ID token in the fragment.

4. Decode the ID token. It contains the `acr` claim with the value of `example_tree`, which the identifier mapped to the `Example` tree in the acr_value map of the OAuth 2.0 provider:

```
{
   "at_hash": "3WHa52upb5ihwWVDC8a-Tw",
   "sub": "demo",
   "auditTrackingId": "fe330f16-2115-45fe-ae04-f68a9fc2ef92-65191",
   "iss": "https://openam.example.com:8443/openam/oauth2",
   "tokenName": "id_token",
   "nonce": "abc123",
   "sid": "I0GdWDfy1qhahDl1PpEA0v5LDspul+qW70biBhetUCk=",
   "aud": "myClient",
   "acr": "example_tree",
   "org.forgerock.openidconnect.ops": "7r1RiXbjWp1QBjJ8Uys4Ob8cwxY",
   "azp": "myClient",
   "auth_time": 1554724614,
   "realm": "/alpha",
   "exp": 1554728218,
   "tokenType": "JWTToken",
   "iat": 1554724618
}
```

The Authentication Method Reference (amr) Claim

In the OpenID Connect specification, the `amr` claim identifies a family of authentication methods, such as a one-time password or multi-factor authentication.

In AM, you can map authentication modules to specific values that the relying party understands. For example, you could map an amr value called `PWD` to the LDAP module.

Unlike `acr` claims, relying parties do not request `amr` claims. As long as authentication modules are mapped to amr values, and provided that end users log in using one of the mapped modules, AM will return the `amr` claim in the ID token.

Since authentication nodes are not used on their own but as part of a tree context, you cannot map amr values to specific authentication nodes. However, you can map an `AuthType` session property to an amr value using the "Set Session Properties Node" in the *Authentication and Single Sign-On Guide*. AM will add the configured amr claim to the ID token, provided the user's journey on the tree goes through the node.

The following is an example of a decoded ID token that contains both `acr` and `amr` claims:

```
{
   "at_hash": "kP7U-po4xla0OYqJ60p72Q",
   "sub": "demo",
   "auditTrackingId": "ac8ecadc-140f-48a0-b3ec-ccd02d6f9c3d-183361",
   "amr": [
      "PWD"
   ],
   "iss": "https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha",
   "tokenName": "id_token",
   "nonce": "abc123",
   "sid": "I0GdWDfy1qhahDl1PpEA0v5LDspul+qW70biBhetUCk=",
   "aud": "myClient",
   "acr": "0",
   "org.forgerock.openidconnect.ops": "hM7F00xw0kzb7os_S9KdmDphosY",
   "azp": "myClient",
   "auth_time": 1554889732,
   "realm": "/alpha",
   "exp": 1554893403,
   "tokenType": "JWTToken",
   "iat": 1554889803
}
```


In this example, the end user logged in with an authentication chain or tree that is not mapped to an acr value. Therefore, AM returned `"acr": "0"`. However, the relying party at least knows the user logged in with an authentication method of the family `PWD`. The relying party can use this knowledge to take additional actions, such as request the end user to reauthenticate using a particular chain or tree.

See the following procedures to map amr values:

- "To Use a Set Session Properties Node to Map an amr Value"
- "To Map an Authentication Module to an amr Value"

To Use a Set Session Properties Node to Map an amr Value

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

2. In the OpenID Connect id_token amr Values to Auth Module Mappings box, map an identifier (the key in the map) to any authentication module. The authentication module you use is not important; AM will only use its name to map the amr, and it will not show in the ID token.

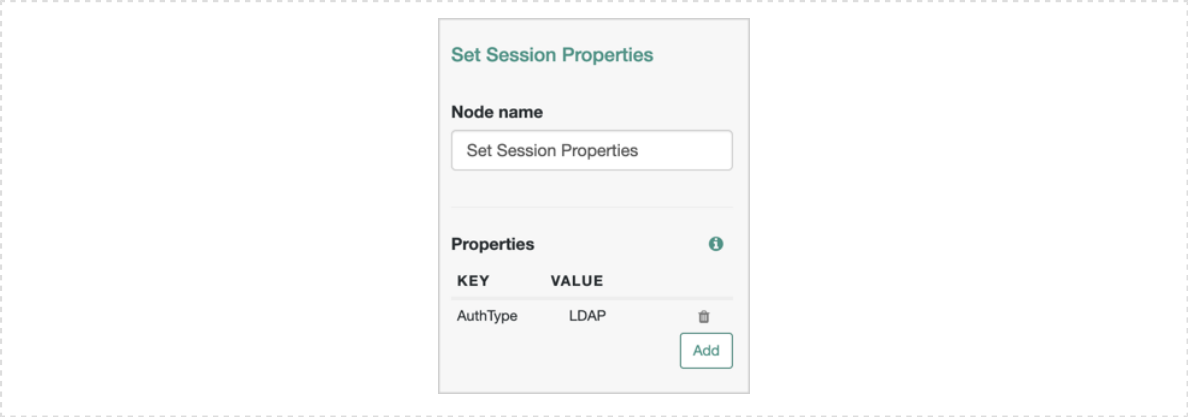
   + *Example: Mapping Authentication Modules to amr Identifiers*

3. Save your changes.

4. Create an authentication tree containing the "Set Session Properties Node" in the *Authentication and Single Sign-On Guide*.

5. On the Set Session Properties node, configure a key called `AuthType`. As its value, set the name of the authentication module you configured with the amr mapping. For example, `LDAP`.

   + *Example: Configuring the AuthType Session Property*



Tip

To reference multiple authentication modules, separate amr values with `|`. For example, if both the LDAP and the DataStore modules are mapped to amr values, set the `AuthType` key to the value `LDAP|DataStore`.

6. Go to Realms > *Realm Name* > Services, and add a Session Property Whitelist service if one is not already available.

7. On the Whitelisted Session Property Names field, add the `AuthType` key. This will ensure that the property can be read, edited, or deleted, from a session.

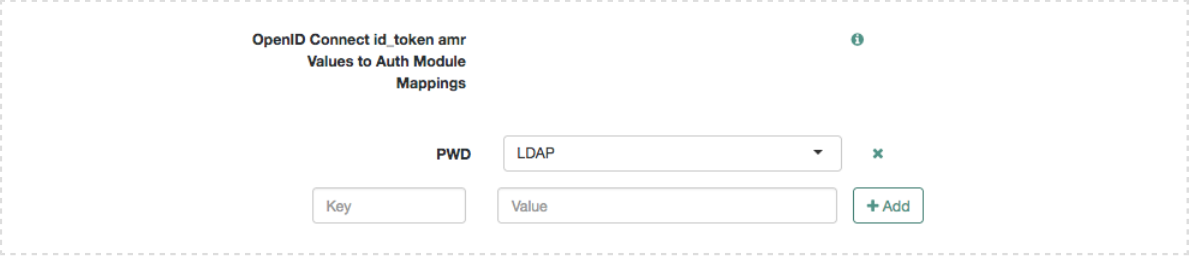
   Save your changes.

To Map an Authentication Module to an amr Value

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

2. In the OpenID Connect id_token amr Values to Auth Module Mappings box, map an identifier (the key in the map) to an authentication module.

   + *Example: Mapping Authentication Modules to amr Identifiers*



3. Save your changes.

Chapter 9

GSMA Mobile Connect

GSMA Mobile Connect is an application of OpenID Connect that facilitates the use of mobile phones as authentication devices independent of the service provided, and independent 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.

In a Mobile Connect deployment, AM can play the following roles:

- **The OpenID provider role**, implementing the Mobile Connect Profile as part of the Service Provider (Identity Gateway interface).

  As OpenID provider, AM responds to a successful authorization request with a response containing all the required fields, and also the optional `expires_in` field. AM 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:

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

- **The Authenticator role**, implementing the Mobile Connect Profile as part of the Identity Gateway (Authenticators interface).

  In the Authenticator role, AM ensures that users authenticate at the appropriate Level of Assurance (LoA). 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.

  + *LoA Support*

    In AM, LoAs map to an authentication mechanism. Service Providers acting as relying parties request an LoA by using the `acr_values` field in an authentication request.

    AM returns the corresponding `acr` claim in the ID token.

    LoA support:

    - `1` (low - little or no confidence)

- `2` (medium - some confidence, as in single-factor authentication)
- `3` (high - high confidence, as in multi-factor authentication)

LoA support does not include support for `4`, which involves digital signatures. Therefore, the `dtbs` authorization parameter is not supported when requesting tokens to the authorization endpoint.

Perform the steps in this procedure to set up the OAuth2 provider service:

To Configure AM for Mobile Connect

1. Configure an OAuth2 provider service in the realm. See "To Configure the OAuth 2.0 Provider Service" in the *OAuth 2.0 Guide*.

   Mobile Connect is an extension of OpenID Connect. Therefore, review the additional configuration options shown in "*OpenID Provider Configuration*".

2. Go to Realms > *Realm Name* > Services > OAuth2 Provider.

3. Configure OpenID Connect authentication context settings for AM to return `acr` and `amr` claims in the ID tokens.

   For information and examples, see "*Adding Authentication Requirements to ID Tokens*".

4. Go to Realms > *Realm Name* > Identity Stores > *Identity Store Name* > User Configuration.

   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.

   When using DS as an identity store, the value is read from the `modifyTimestamp` attribute, or the `createTimestamp` attribute for a profile that has never been modified.

5. Add the relevant attributes to the LDAP User Attributes list, and save your changes.

   You can now use OpenID Connect with Mobile Connect. As per the specification, you must use the Authorization Code flow to request ID tokens.

   + *Supported Authorization 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`

Request Parameter	Support	Description
		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 AM.
`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 that can be set to the ID the user uses to log in. For example, `Bob` or `bob@example.com`, depending on how the authentication node or module is configured to search for users. When provided as part of the OIDC Authentication Request, the `login_hint` is set as the value of a cookie named `oidcLoginHint`, which is an HttpOnly cookie (only sent over HTTPS).
`acr_values`	Supported	Authentication Context class Reference values used to communicate acceptable LoAs that users must satisfy when authenticating to the OpenID provider. For more information, see "The Authentication Context Class Reference (acr) Claim".
`dtbs`	Not supported	Data To Be Signed At present AM does not support LoA 4.

For access to a simple, non-secure GSMA Mobile Connect relying party sample, see How do I access and build the sample code provided for AM (All versions)? in the *ForgeRock Knowledge Base*.

Chapter 10

Additional Use Cases for ID Tokens

In addition to using the ID tokens in OpenID Connect flows, AM supports using ID tokens in place of session tokens when calling REST endpoints and using ID tokens in policy evaluation.

Using ID Tokens as Session Tokens

You can authorize trusted clients to use ID tokens as the value of the `iPlanetDirectoryPro` cookie. This is useful when clients need to make calls to AM endpoints, such as the authorization endpoints, without requesting the end user to log in again.

The ID token *must* be issued using the Authorization Code Grant flow.

To Configure the OAuth 2.0 Service for Authorized Clients

Follow these steps to let clients use ID tokens in the place of session tokens:

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

2. In the Authorized OIDC SSO Clients field, add the name of each client that will be able to use ID tokens in place of session tokens.

   Because these clients will act with the full authority of the end user, grant this permission to trusted clients only.

3. Ensure that Enable Session Management is enabled.

4. Save your changes.

Important

Although the lifetime of the ID token can be extended using refresh tokens, the maximum lifetime is still determined by the session lifetime. If the session expires, the ID token is rejected by AM, even if the token itself is still valid.

The following is an example of a call to the `policies` endpoint using an ID token instead of a session token:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "Accept-API-Version: protocol=1.0,resource=2.0" \
--header "iPlanetDirectoryPro: ID_TOKEN_VALUE" \
--data '{
   "resources":[
      "https://www.france-site.com:8443/index.html"
   ],
   "subject":{
      "ssoToken": "ID_TOKEN_VALUE"
   },
   "application":"iPlanetAMWebAgentService"
}' \
"https://openam.example.com:8443/openam/json/realms/root/policies?_action=evaluate"
```

Note

To access the `policies` endpoint, a user must have the `Entitlement REST Access` privilege.

Using ID Tokens as Subjects in Policy Decision

You can use the ID token as a subject condition during policy evaluation to validate claims within an ID token.

For example, you can validate that the `aud` claim has a value of `myApplication`, which identifies a particular application or group of applications within your environment.

Note that policy evaluation does not validate the ID token, but the claims within. Your applications should validate the ID token before requesting policy evaluation from AM.

For more information about configuring policy evaluation using the OpenID Connect/JWT Claim type, refer to "To Configure a Policy (UI)" in the *Authorization Guide*.

Chapter 11

OpenID Connect 1.0 Endpoints

AM exposes the following OpenID Connect-related endpoints:

AM Acting As...	Endpoint	Description
Provider	`/oauth2/userinfo`	Retrieves information about an authenticated user. It requires a valid token issued with, at least, the `openid` scope (OpenID Connect userinfo endpoint).
Provider	`/oauth2/idtokeninfo`	Validates unencrypted ID tokens (AM-specific endpoint).
Provider	`/oauth2/connect/checkSession`	Retrieves OpenID Connect session information (OpenID Connect Session Management endpoint).
Provider	`/oauth2/connect/endSession`	Invalidates OpenID Connect sessions (OpenID Connect Session Management endpoint).
Provider	`/oauth2/register`	Registers, reads, and deletes OAuth 2.0 clients (RFC7592 and RFC7591)
Provider	`/.well-known/webfinger`	Exposes the URL of the OpenID provider during OpenID Connect discovery.
Provider	`/oauth2/.well-known/openid-configuration`	Exposes provider configuration for OpenID Connect discovery.
Provider	`"/oauth2/connect/jwk_uri"`	Exposes the public keys that clients can use to verify the signature of client-based tokens and to encrypt OpenID Connect requests sent as a JWT.
Relying Party	`"/oauth2/connect/rp/jwk_uri"`	Exposes AM client public keys. Providers can use them to encrypt ID tokens sent to AM, and to verify JWT and object signatures coming from AM.

Tip

When AM acts as an OpenID Connect provider, the OAuth 2.0 endpoints support OpenID Connect specific parameters, such as `prompt` and `ui_locales`.

For a complete list of the endpoints and parameters AM supports as an OAuth 2.0/OpenID Connect provider, see "*OAuth 2.0 Endpoints*" and "*OAuth 2.0 Administration and Supporting REST Endpoints*" in the *OAuth 2.0 Guide*.

/oauth2/userinfo

Endpoint that returns claims about the authenticated end user, as defined in OpenID Connect Core 1.0 incorporating errata set 1.

When requesting claims, provide an access token granted in an OpenID Connect flow as an authorization bearer header. The endpoint will return the claims associated with the scopes granted when the access token was requested.

You must compose the path to the user information endpoint addressing the specific realm where AM logged in the user. For example, `https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/userinfo`.

The following example shows AM returning claims about a user:

```
$ curl \
--request GET \
--header "Authorization: Bearer U-Wjlv-w1jtpuBVWUGFV6PwI_nE" \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/userinfo"
{
    "given_name":"Demo First Name",
    "family_name":"Demo Last Name",
    "name":"demo",
    "sub":"(usr!demo)",
    "subname":"id=demo,ou=user,o=root,ou=services,dc=openam,dc=forgerock,dc=org"
}
```

If the access token validates successfully, the endpoint returns the claims as JSON.

+ *About the Subject and the Subname Claims*

The subject claim is in the format `(type!subject)`, where:

- `subject` is the identifier of the user/identity, or the name of the OAuth 2.0/OpenID Connect client that is the subject of the token.
- `type` can be one of the following:
  - `age`. Specifies that the *subject* is an OAuth 2.0/OpenID Connect-related user-agent or client. For example, an OAuth 2.0 client, a Remote Consent Service agent, and a Web and Java Agent internal client.
  - `usr`. Specifies that the *subject* is a user/identity.

For example, `(usr!demo)`, or `(age!myOAuth2Client)`.

> The value of the `subname` claim matches the value of the *subject* portion of the `sub` claim.

The user information endpoint can return claims as JSON (the default) or as a signed, encrypted, or signed and encrypted JWT. To configure the response type, perform the following steps:

1. In the AM console, go to Realms > *Realm Name* > Applications > OAuth 2.0 > Clients > *Client Name* > Signing and Encryption.
2. In the User info response format drop-down menu, select the type of response required by the client.
3. Configure the signing and/or encryption algorithms AM should use when returning claims to this particular client in the following properties:
   - User info signed response algorithm
   - User info encrypted response algorithm
   - User info encrypted response encryption algorithm

   For more information about these properties, see Signing and Encryption Properties in the *OAuth 2.0 Guide*.

   Note that you can configure the algorithms the OAuth 2.0/OpenID Connect provider service supports by navigating to > *Realm Name* > Services > OAuth2 Provider > Advanced OpenID Connect.

   For more information about the secret IDs mapped to the OAuth 2.0/OpenID Connect provider signing and encrypting algorithms, see "Secret ID Default Mappings" in the *Security Guide*.

Signed, and signed and encrypted JWT responses will include the `iss` and the `aud` objects.

/oauth2/idtokeninfo

AM-specific endpoint that lets relying parties validate *unencrypted* ID tokens and to retrieve claims within the token.

AM validates the tokens based on rules 1-10 in section 3.1.3.7 of the OpenID Connect Core. During token validation, AM performs the following steps:

1. Extracts the first `aud` (audience) claim from the ID token. The `client_id`, which is passed in as an authentication of the request, will be used as the client and validated against the `aud` claim.
2. Extracts the `realm` claim, if present, defaults to the root realm if the token was not issued by AM.
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 ID Token Signing Algorithm and Public key selector settings in the client profile.

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

Tip

By default, the ID token information endpoint requires client authentication. You can configure it by navigating to Realms > *Realm Name* Services > OAuth2 Provider > Advanced OpenID Connect and disabling the Idtokeninfo Endpoint Requires Client Authentication switch.

The ID token information endpoint supports the following parameters:

`id_token`

Specifies the ID token to validate.

Required: Yes.

`client_id`

Specifies the client ID unique to the application making the request.

Required: Yes, when client authentication is enabled.

`client_assertion`

Specifies the signed JWT that the client uses as a credential when using the JWT bearer client authentication method.

Required: A form of password or credentials is required for confidential clients when client authentication is enabled. However, the use of the `client_assertion` parameter depends on the client authentication method used. For more information, see "*OAuth 2.0 Client Authentication*" in the *OAuth 2.0 Guide*

`client_assertion_type`

Specifies the type of assertion when the client is authenticating to the authorization server using JWT bearer client authentication.

Set it to `urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer`.

Required: A form of password or credentials is required for confidential clients when client authentication is enabled. However, the use of the `client_assertion_type` parameter depends on the client authentication method used. For more information, see "*OAuth 2.0 Client Authentication*" in the *OAuth 2.0 Guide*

`client_secret`

Specifies the secret of the client making the request.

Required: A form of password or credentials is required for confidential clients when client authentication is enabled. However, the use of the `client_secret` parameter depends on the client authentication method used. For more information, see "*OAuth 2.0 Client Authentication*" in the *OAuth 2.0 Guide*

`claims`

Comma-separated list of claims to return from the ID token.

Required: No.

The endpoint is always accessed from the root realm. For example, `https://openam.example.com:8443/openam/oauth2/idtokeninfo`.

The following example shows AM returning ID token information:

```
$ curl \
--request POST \
--data "client_id=myClient" \
--data "client_secret=forgerock" \
--data "id_token=eyJ0eXAiOiJKV1QiLCJra...7r8soMCk8A7QdQpg" \
"https://openam.example.com:8443/openam/oauth2/idtokeninfo"
{
   "at_hash":"AvJ0dXLQgFxHn-qnqP9xmQ",
   "sub":"(usr!demo)",
   "auditTrackingId":"b3f48c69-de7f-4afc-ab78-582733e5f025-156621",
   "iss":"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha",
   "tokenName":"id_token",
   "nonce":"123abc",
   "sid": "I0GdWDfy1qhahDl1PpEA0v5LDspul+qW70biBhetUCk=",
   "aud":"myClient",
   "c_hash":"jMXGi-FCjUad2VQukJRcLQ",
   "acr":"0",
   "s_hash":"bKE9UspwyIPg8LsQHkJaiQ",
   "azp":"myClient",
   "auth_time":1553077105,
   "realm":"/myRealm",
   "exp":1553080707,
   "tokenType":"JWTToken",
   "iat":1553077107
}
```

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 specific claims.

For example, you can run the following command to retrieve specific claims:

```
$ curl \
--request POST \
--data "client_id=myClient" \
--data "client_secret=forgerock" \
--data "id_token=eyJ0eXAiOiJKV1QiLCJra...7r8soMCk8A7QdQpg" \
--data "claims=sub,exp,realm" \
"https://openam.example.com:8443/openam/oauth2/idtokeninfo"
{
   "sub":"(usr!demo)",
   "exp":1553080707,
   "realm":"/myRealm"
}
```

If a requested claim does not exist, no error occurs; AM will simply not present it in the response.

+ *About the Subject and the Subname Claims*

The subject claim is in the format `(type!subject)`, where:

- `subject` is the identifier of the user/identity, or the name of the OAuth 2.0/OpenID Connect client that is the subject of the token.
- `type` can be one of the following:
  - `age`. Specifies that the *subject* is an OAuth 2.0/OpenID Connect-related user-agent or client. For example, an OAuth 2.0 client, a Remote Consent Service agent, and a Web and Java Agent internal client.
  - `usr`. Specifies that the *subject* is a user/identity.

For example, `(usr!demo)`, or `(age!myOAuth2Client)`.

The value of the `subname` claim matches the value of the *subject* portion of the `sub` claim.

The `subname` claim is not included in the `OIDC Claims Script`. Therefore, AM does not add it to ID tokens by default.

Note

The ID token information endpoint does not check if a token has been revoked using the `/oauth2/endSession` endpoint.

/oauth2/connect/checkSession

Endpoint to check session state as per OpenID Connect Session Management 1.0 - draft 5.

The relying party client creates an invisible iframe that embeds the URL to the endpoint (by setting it as the `src` attribute of the `iframe` tag).

The endpoint accepts postMessage API requests from the iframe, and it postMessages back with the login status of the user in AM.

The endpoint is always accessed from the root realm. For example, `https://openam.example.com:8443/openam/oauth2/connect/checkSession`.

Tip

Note that this endpoint has been removed in later versions of the OpenID Connect Session Management draft. For an alternative method of checking session state, see "Session Management Draft 10".

/oauth2/connect/endSession

Endpoint to terminate authenticated end-user sessions, as per OpenID Connect Session Management 1.0 - draft 5.

Query the well-known configuration endpoint for the realm to determine the URL of the end session endpoint. For example, `https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/.well-known/openid-configuration`.

The endpoint supports the following query parameters:

`id_token_hint`

The ID token corresponding to the identity of the end user the relying party is requesting to be logged out by AM.

Required: Yes

`client_id`

To support ending sessions when ID tokens are encrypted, AM requires that the request to the end session endpoint includes the client ID to which AM issued the ID token.

Failure to include the client ID will result in error; AM needs the information in the client profile to decrypt the token.

This parameter is `not` compliant with the specification.

Required: Yes, if the ID token is encrypted.

`post_logout_redirect_uri`

The URL AM will redirect to after logout.

For security reasons, the value of this parameter must match one of the values configured in the Post Logout Redirect URIs field of the client profile.

If a logout redirection URL is specified, AM redirects the end user to it after they have been logged out.

If a logout redirection URL is not specified, AM returns an HTTP 204 message to indicate the user has been logged out, and does not perform more actions.

Required: No

This example shows AM deleting a session when an encrypted ID token is provided, and redirecting the end user to the logout redirect URL specified:

```
$ curl --dump-header - \
--request GET \
"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/connect/endSession?
id_token_hint=eyJ0eXAiOiJKV1QiLCJra...&post_logout_redirect_uri=https://www.example.com:443/
logout_callback&client_id=myClient"
HTTP/1.1 302
X-Frame-Options: SAMEORIGIN
X-Content-Type-Options: nosniff
Location: https://www.example.com:443/logout_callback
Content-Length: 0
Date: Mon, 12 Sep 2022 09:33:45 GMT
```

/oauth2/register

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

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

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

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

The endpoint supports the following methods:

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

For usage information and examples, see the following links:

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

/.well-known/webfinger

Lets clients determine the provider URL for an end user, as described in the OpenID Connect Discovery 1.0 incorporating errata set 1 specification.

Tip

The endpoint is disabled by default; to enable it, see "OpenID Connect Discovery".

The discovery endpoint supports the following parameters:

`realm`

Specifies the AM realm that must be queried for user information. Unlike other AM endpoints, the discovery endpoint does not support specifying the realm in the path, because it is always located after the deployment URI. For example, `https://openam.example.com:8443/openam/.well-known/webfinger`.

Required: No

`resource`

Identifies the URL-encoded subject of the request. This parameter can take the following formats, as defined in the specification:

- `acct:`*`user_email`*. For example, `acct%3Ademo%40example.com`.
- `acct:`*`user_email@host`*. For example, `acct%3Ademo%2540example.com%40server.example.com`
- *`http_or_https://host/username`*. For example, `http%3A%2F%2Fserver.example.com%2Fdemo`.
- *`http_or_https://host:port`*. For example, `http%3A%2F%2Fserver.example.com%3A8080`.

The value of `host` is related to the discovery URL exposed to the clients. In the examples, the exposed discovery endpoint would be something similar to `http://server.example.com/.well-known/webfinger`. For more information about exposing the endpoint through a proxy or load balancer, see "OpenID Connect Discovery".

Wildcard (*) characters are not supported.

Required: Yes.

`rel`

Specifies the URL-encoded URI identifying the type of service whose location is requested. The only valid value is `http://openid.net/specs/connect/1.0/issuer`.

Required: Yes.

The following command requests information for the `demo` user in the `example.com` domain to the OAuth 2.0 provider service in the `Engineering` realm:

```
$ curl \
--request GET \
"https://openam.example.com:8443/openam/.well-known/webfinger\
?resource=acct%3Ademo%40example.com\
&realm=Engineering\
&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"
    }
  ]

}
```

/oauth2/.well-known/openid-configuration

Lets relying parties retrieve the 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 realm named `alpha`, the URL would resemble the following: `https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/.well-known/openid-configuration`.

Tip

For more information about OpenID Connect discovery, see "OpenID Connect Discovery".

After the relying party has discovered who the provider for the end user is, they can discover the provider's configuration:

```
$ curl "https://openam.example.com:8443/openam/oauth2/.well-known/openid-configuration"
{
   "request_parameter_supported":true,
   "claims_parameter_supported":false,
   "introspection_endpoint":"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/
introspect",
   "check_session_iframe":"https://openam.example.com:8443/openam/oauth2/connect/checkSession",
   "scopes_supported":[
      "address",
      "phone",
      "openid",
      "profile",
      "email"
   ],
   "userinfo_endpoint":"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/userinfo",
   "jwks_uri":"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/connect/jwk_uri",
   "registration_endpoint":"https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/
register",
  ....
}
```

/oauth2/connect/jwk_uri

Each realm configured for OAuth 2.0 exposes a JSON web key (JWK) URI endpoint that contains public keys that clients can use to:

- Verify the signature of client-based access tokens and OpenID Connect ID tokens.
- Encrypt OpenID Connect ID requests to AM sent as a JWT.

+ *Where Do the Keys Come From?*

By default, the endpoint exposes an internal URI relative to the AM deployment. For example, `openam/oauth2/realms/root/connect/jwk_uri`.

The keys appearing in that URI are those configured in the AM secret stores in the *Security Guide*, regardless of the algorithms configured in the OAuth 2.0 provider. This is to support the process of deprecating keys/algorithms.

Secrets are configured by default; delete the ones you are not planning to use so that they are not exposed on the endpoint.

In environments where secrets are centralized, you may want AM to share the URI of your secrets API instead of the local AM endpoint. To configure it, go to Realms > *Realm name* > Services > OAuth2 Provider, and add the new URI to the Remote JSON Web Key URL field.

Note that HMAC-based algorithms, direct encryption, and AES key wrapping encryption algorithms use the client secret instead of a public key. Therefore, clients do not need to check the JWK URI endpoint for those algorithms.

+ *The Endpoint is Exposed, But I Haven't Configured an OAuth 2.0 Provider Yet*

Web and Java agents use an internal OAuth 2.0 provider to connect to AM. This provider exposes the endpoint so that agents can access the key configured for the `am.global.services.oauth2.oidc.agent.idtoken.signing` secret ID.

Tip

Configure the base URL source service in the *Security Guide* to change the URL used in the `.well-known` endpoints used in OpenID Connect 1.0 and UMA.

The following table summarizes the high-level tasks you need to complete to manage the JWK URI endpoint in your environment:

Task	Resources
Learn where to find and how to query the JWK URI endpoint. Clients need to find the endpoint to, for example, validate tokens signed by AM.	"To Access the Keys Exposed by the JWK URI Endpoint"
Control which keys are displayed. The JWK URI endpoint returns keys based on the secret mappings configured for the relevant OAuth 2.0/OpenID connect functionality. Therefore, to control which keys are displayed, ensure that you only map the secrets required in your environment.	"Configuring ID Token Signatures"
Learn how to deprecate algorithms and how to rotate public keys. You may need to perform these tasks to replace algorithms with more secure ones.	"Deprecating Algorithms and Rotating Public Keys"
Customize the key ID (`kid`) of the exposed keys. By default, AM generates a `kid` for each public key exposed in the `jwk_uri` endpoint when AM is configured as an OAuth 2.0 authorization server. You need to customize AM if any exposed keys in your environment need a specific `kid`.	"Customizing Public Key IDs".
Decide if the JWK URI endpoint should display duplicated key IDs By default, each `kid` exposed by the `jwk_uri` endpoint matches a unique secret, as required by the RFC7517 specification. If you have several algorithms and key types associated with one `kid`, configure AM to display them individually.	"Displaying Every Algorithm and Key Type Associated to a Key ID".

To Access the Keys Exposed by the JWK URI Endpoint

Perform the following steps to access the public keys:

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

```
$ curl https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/.well-known/openid-
configuration
{
  "request_parameter_supported": true,
  "claims_parameter_supported": false,
  "introspection_endpoint": "https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/
introspect",
  "check_session_iframe": "https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/
connect/checkSession",
  "scopes_supported": [],
  "issuer": "https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha",
  "id_token_encryption_enc_values_supported": [
    "A256GCM",
    "A192GCM",
    "A128GCM",
    "A128CBC-HS256",
    "A192CBC-HS384",
    "A256CBC-HS512"
  ],
...
  "jwks_uri": "https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/connect/
jwk_uri",
  "subject_types_supported": [
    "public"
  ],
...
}
```

   By default, AM exposes the JWK URI as an endpoint relative to the deployment URI. For example, `https://openam.example.com:8443/openam/oauth2/realms/root/connect/jwk_uri`.

   In environments where secrets are centralized, you may want AM to share the URI of your secrets API instead of the local AM endpoint.

   To configure it, go to Realms > *Realm name* > Services > OAuth2 Provider, and add the new URI to the Remote JSON Web Key URL field.

2. Perform an HTTP GET at the JWK URI to get the relevant public keys. For example:

```
$ curl https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/connect/jwk_uri
{
"keys":[
      {
         "kty":"EC",
         "kid":"I4x/IijvdDsUZMghwNq2gC/7pYQ=",
         "use":"sig",
         "x5t":"GxQ9K-sxpsH487eSkJ7lE_SQodk",
         "x5c":[
            "MIIB/zCCAYYCCQDS7UWmBdQtETAJ0mN0TZL7/MaY..."
               ],
         "x":"k5wSvW_6JhOuCj-9PdDWdEA4oH90RSmC2GTliiUHAhXj6rmTdE2S-_zGmMFxufuV",
         "y":"XfbR-tRoVcZMCoUrkKtuZUIyfCgAy8b0FWnPZqevwpdoTzGQBOXSNi6uItN_o4tH",
         "crv":"P-384"
      },
      ...
   ]
}
```

Configuring ID Token Signatures

ID tokens are signed by default with a test key configured during installation. Change this key on production-like and production environments.

To Configure ID Token Digital Signatures

Perform the steps in this procedure to configure the signing algorithm AM should use to sign OpenID Connect tokens:

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

2. In the OpenID Connect tab, ensure that the ID Token Signing Algorithms supported list has the signing algorithms your environment requires.

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

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

   + *Secret ID Mappings for Signing OpenID Connect Tokens*

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

Secret ID	Default Alias	Algorithms [a]
am.services.oauth2.oidc.signing.ES256	es256test	ES256
am.services.oauth2.oidc.signing.ES384	es384test	ES384

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

[a] The following applies to confidential clients only:

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

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

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

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

3. (Optional) If the client is configured in AM, go to Realms > *Realm Name* > Applications > OAuth 2.0 > *Client Name*.

   Clients registering dynamically can see the algorithms that the provider supports by making a call to the `/oauth2/.well-known/openid-configuration` endpoint.

4. In the ID Token Signing Algorithm field, enter the signing algorithm that AM will use to sign the ID token for this client.

   Note that the OAuth 2.0 provider must support signing with the chosen algorithm.

5. Save your changes.

   AM is ready to sign ID tokens with the algorithm you configured.

Tip

If you chose a non-HMAC-based algorithm, the client will need to make a request to AM's JWK URI endpoint for the realm to recover the signing public key they can use to validate the ID tokens.

Deprecating Algorithms and Rotating Public Keys

With signing and encryption methods changing so rapidly, during the lifecycle of your OAuth 2.0 environment you will need to deprecate older, less secure signing and/or encrypting algorithms in favor of new ones.

In the same way, you will rotate the keys AM uses for signing and encryption if you suspect they may have been leaked or just due to security policies, such as deprecating algorithms or because they have reached the end of their lifetime.

The keys you expose in the JWK URI endpoint should reflect the algorithms currently supported by AM as well as the deprecated ones. Otherwise, clients still using tokens signed with deprecated keys would not be able to validate them.

This is why deprecating supported algorithms in the OAuth 2.0/OpenID Connect provider is a two-step process:

1. Remove the old algorithm from the OAuth 2.0 provider supported algorithm list. This stops new clients from registering with that algorithm.
2. After a grace period, remove the secret mapping associated to that algorithm. This removes the associated public keys from the JWK URI endpoint.

To Deprecate Supported Algorithms and their Keys

Perform the steps in this procedure to deprecate an algorithm and its related keys. If you only want to deprecate keys or rotate them as part of your environment's security policies, see "Mapping and Rotating Secrets" in the *Security Guide* instead.

1. (Optional) Configure the new algorithm, if required.
   a. Go to Realms > *Realm Name* > Services > OAuth2 Provider > OpenID Connect.
   b. (Optional) In the ID Token Signing Algorithm supported field, add the new signing algorithm, if not already present.
   c. (Optional) In the ID Token Encryption Algorithms supported field, add the new encryption algorithm, if not already present.
   d. Save your changes.
2. (Optional) Configure secret ID mappings for the keys using the new algorithm, if required.

For more information, see "Configuring Secret Stores" in the *Security Guide*.

3. Remove the algorithm to be deprecated from the relevant OAuth 2.0 provider algorithm list:

   a. Go to Realms > *Realm Name* > Services > OAuth2 Provider > OpenID Connect.

   b. (Optional) In the ID Token Signing Algorithm supported field, remove the deprecated signing algorithm.

   c. (Optional) In the ID Token Encryption Algorithms supported field, remove the deprecated encryption algorithm.

   d. Save your changes.

4. Decide on a grace period. For example, a month. During this period both the deprecated and the new algorithms/keys are supported.

   New clients cannot register with the deprecated algorithms and are forced to use a supported algorithm. However, since the deprecated keys are still mapped to secret IDs, existing clients still can use them to validate active tokens and encrypt requests.

   Existing clients must change their configuration during the grace period to use one of the supported algorithms.

5. After the grace period, remove the secret ID mappings relevant to the deprecated algorithm.

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

Customizing Public Key IDs

By default, AM generates a key ID (`kid`) for each public key exposed in the `jwk_uri` URI when AM is configured as an OAuth 2.0 authorization server.

For keys stored in a keystore or HSM secret store, you can customize how key ID values are determined by writing an implementation of the `KeyStoreKeyIdProvider` interface and configuring it in AM:

To Customize Public Key IDs

Perform the following steps:

1. Write your own implementation of the `KeyStoreKeyIdProvider` interface that provides a specific key ID for a provided public key. For more information, see the `KeyStoreKeyIdProvider` interface in the *Access Management 7.1.4 Java API Specification*.

2. In the AM console, configure the OAuth 2.0/OpenID Connect Provider service, if not done already. For more information, see "*Authorization Server Configuration*" in the *OAuth 2.0 Guide*.

3. Go to Configure > Server Defaults > Advanced.

4. Add an advanced server property called `org.forgerock.openam.secrets.keystore.keyid.provider`, whose value is the fully qualified name of the class you wrote in previous steps. For example:

```
org.forgerock.openam.secrets.keystore.keyid.provider =
 com.mycompany.am.secrets.CustomKeyStoreKeyIdProvider
```

5. Restart the AM instance or the container in which it runs.

6. Verify that the customized key IDs are displayed by navigating to the OAuth 2.0 authorization server's `jwk_uri` URI. For example, `https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/connect/jwk_uri`.

Displaying Every Algorithm and Key Type Associated to a Key ID

By default, each key ID (`kid`) exposed by the `jwk_uri` endpoint matches a unique secret, as recommended by the RFC7517 specification. This means that each `kid` is of a particular key type, and uses a particular algorithm.

If you have several algorithms and key types associated with one `kid`, configure the JWK URI endpoint to display them as different keys in the JWK. Note that when including all combinations associated with a `kid`, that `kid` does not uniquely identify a particular secret.

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

2. Enable Include all kty and alg combinations in jwk_uri.

3. Save your changes.

4. Verify that you can now see duplicate `kid` entries for different combinations of algorithms and key types.

   For more information, see "To Access the Keys Exposed by the JWK URI Endpoint".

/oauth2/connect/rp/jwk_uri

As well as acting as the provider, AM can also act as the relying party. To share its client public secrets, AM exposes a JSON web key (JWK) URI endpoint for each realm.

Use this endpoint during ForgeRock Identity Platform social identity registration, where providers can use the exposed secrets to:

- Encrypt ID tokens returned to AM.
- Verify the signature of JWTs coming from AM, such as that of request objects or client authentication JWTs.

- Decrypt client authentication JWTs coming from AM.

Specify the AM realm path in the URI, as follows:

```
/oauth2/realms/root/realms/{realm}/connect/rp/jwk_uri
```

Example:

```
$ curl https://openam.example.com:8443/openam/oauth2/realms/root/realms/alpha/connect/rp/jwk_uri
{
  "keys": [
    {
      "kty": "RSA",
      "kid": "DkKMPE7hFVEn77WWhVuzaoFp4O8=",
      "use": "enc",
      "x5t": "JRxY4hJRL3sI_dAUWUEosCEQJ3A",
      "x5c": [
        "MIIDYTCCAkm...eP4wLr3cM="
      ],
      "n": "i7t6m4d_02dZ8dOe-DFc...zflF8jR9pewTbQ",
      "e": "AQAB"
    },
    {
      "kty": "RSA",
      "kid": "wU3ifIIaLOUAReRB/FG6eM1P1QM=",
      "use": "sig",
      "x5t": "5eOfy1Nn2MMIKVRRkq0OgFAw348",
      "x5c": [
        "MIIDdzCCAl+gAwIBAgIES3eb+zANBgk...s009kbW6inN8zA6"
      ],
      "n": "10iGQ5l5IdqB...AJW4ZSg1PPO2UJSQ",
      "e": "AQAB"
    }
  ]
}
```

Supply the JWK URI to the provider when registering AM as a relying party. Consult the documentation provided by your OpenID provider for more information.

The JWK URI endpoint publishes keys based on secret mappings made either globally, or in the specific realm. The secret IDs to map are as follows:

`am.services.oauth2.oidc.rp.jwt.authenticity.signing`

The OpenID Connect provider obtains the public key from the alias mapped to this secret, and uses it to verify the signature applied to request objects it receives.

All aliases configured for the secret ID are published at the endpoint so that, when you rotate secrets in the *Security Guide*, the provider is still able to validate JWTs with all the secrets.

The active secret is the only one that AM uses for signing, however.

`am.services.oauth2.oidc.rp.idtoken.encryption`

The OpenID Connect provider obtains the public key from the alias mapped to this secret, and uses it to encrypt ID tokens and `userinfo` endpoint data in JWT format before returning it to AM.

Unlike the signing secret ID above, only the alias that is marked as *active* in the mappings is published at the endpoint. Any additional mappings are ignored.

`am.services.oauth2.mtls.client.authentication`

The OpenID Connect provider obtains the public JWK from the alias mapped to this secret, and uses it to verify the mutual TLS self-signed certificate that the client uses to authenticate.

Secrets configured globally will show in the JWK URI for all realms.

In a new installation of AM, these signing and encryption secret IDs are mapped by default, as explained in the table below:

+ *Secret ID Mappings for Decrypting ID Tokens*

The following table shows the secret ID mapping to support decryption of ID tokens and `userinfo` endpoint data in JWT format when AM is configured as a relying party of the Social Identity Provider Service:

Secret ID	Default Alias	Algorithms
am.services.oauth2.oidc.rp.idtoken.encryption	test	—

The public key is exposed in the "/oauth2/connect/rp/jwk_uri".

For more information about the algorithms supported, and how to configure this secret ID mapping, see "*Social Authentication*" in the *Authentication and Single Sign-On Guide*.

+ *Secret ID Mappings for Signing JWTs and Objects*

The following table shows the secret ID mapping that AM uses to sign JWTs and objects when configured as a relying party of the Social Identity Provider Service:

Secret ID	Default Alias	Algorithms
am.services.oauth2.oidc.rp.jwt.authenticity.signing	rsajwtsigningkey	—

The public key is exposed in the "/oauth2/connect/rp/jwk_uri".

For more information about the algorithms supported, and how to configure this secret ID mapping, see "*Social Authentication*" in the *Authentication and Single Sign-On Guide*.

Note

In upgraded AM instances, the secret IDs will not have default aliases mapped, and the JWK URI endpoint returns an empty JWK set.

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

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

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

Glossary

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

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

	request. For browser-based clients, AM sets a cookie in the browser that contains the session information. For browser-based clients, AM sets a cookie in the browser that contains the session state. When the browser transmits the cookie back to AM, AM decodes the session state from the cookie.
Conditions	Defined as part of policies, these determine the circumstances under which which a policy applies. Environmental conditions reflect circumstances like the client IP address, time of day, how the subject authenticated, or the authentication level achieved. Subject conditions reflect characteristics of the subject like whether the subject authenticated, the identity of the subject, or claims in the subject's JWT.
Configuration datastore	LDAP directory service holding AM configuration data.
Cross-domain single sign-on (CDSSO)	AM capability allowing single sign-on across different DNS domains.
CTS-based OAuth 2.0 tokens	After a successful OAuth 2.0 grant flow, AM returns a *reference* to the token to the client, rather than the token itself. This differs from client-based OAuth 2.0 tokens, where AM returns the entire token to the client.
CTS-based sessions	AM sessions that reside in the Core Token Service's token store. CTS-based sessions might also be cached in memory on one or more AM servers. AM tracks these sessions in order to handle events like logout and timeout, to permit session constraints, and to notify applications involved in SSO when a session ends.
Delegation	Granting users administrative privileges with AM.
Entitlement	Decision that defines which resource names can and cannot be accessed for a given identity in the context of a particular application, which actions are allowed and which are denied, and any related advice and attributes.
Extended metadata	Federation configuration information specific to AM.
Extensible Access Control Markup Language (XACML)	Standard, XML-based access control policy language, including a processing model for making authorization decisions based on policies.
Federation	Standardized means for aggregating identities, sharing authentication and authorization data information between trusted providers, and

	allowing principals to access services across different providers without authenticating repeatedly.
Fedlet	Service provider application capable of participating in a circle of trust and allowing federation without installing all of AM on the service provider side; AM lets you create Java Fedlets.
Hot swappable	Refers to configuration properties for which changes can take effect without restarting the container where AM runs.
Identity	Set of data that uniquely describes a person or a thing such as a device or an application.
Identity federation	Linking of a principal's identity across multiple providers.
Identity provider (IDP)	Entity that produces assertions about a principal (such as how and when a principal authenticated, or that the principal's profile has a specified attribute value).
Identity repository	Data store holding user profiles and group information; different identity repositories can be defined for different realms.
Java agent	Java web application installed in a web container that acts as a policy enforcement point, filtering requests to other applications in the container with policies based on application resource URLs.
Metadata	Federation configuration information for a provider.
Policy	Set of rules that define who is granted access to a protected resource when, how, and under what conditions.
Policy agent	Java, web, or custom agent that intercepts requests for resources, directs principals to AM for authentication, and enforces policy decisions from AM.
Policy Administration Point (PAP)	Entity that manages and stores policy definitions.
Policy Decision Point (PDP)	Entity that evaluates access rights and then issues authorization decisions.
Policy Enforcement Point (PEP)	Entity that intercepts a request for a resource and then enforces policy decisions from a PDP.
Policy Information Point (PIP)	Entity that provides extra information, such as user profile attributes that a PDP needs in order to make a decision.
Principal	Represents an entity that has been authenticated (such as a user, a device, or an application), and thus is distinguished from other entities.

	When a Subject successfully authenticates, AM associates the Subject with the Principal.
Privilege	In the context of delegated administration, a set of administrative tasks that can be performed by specified identities in a given realm.
Provider federation	Agreement among providers to participate in a circle of trust.
Realm	AM unit for organizing configuration and identity information. Realms can be used for example when different parts of an organization have different applications and identity stores, and when different organizations use the same AM deployment. Administrators can delegate realm administration. The administrator assigns administrative privileges to users, allowing them to perform administrative tasks within the realm.
Resource	Something a user can access over the network such as a web page. Defined as part of policies, these can include wildcards in order to match multiple actual resources.
Resource owner	In OAuth 2.0, entity who can authorize access to protected web resources, such as an end user.
Resource server	In OAuth 2.0, server hosting protected web resources, capable of handling access tokens to respond to requests for such resources.
Response attributes	Defined as part of policies, these allow AM to return additional information in the form of "attributes" with the response to a policy decision.
Role based access control (RBAC)	Access control that is based on whether a user has been granted a set of permissions (a role).
Security Assertion Markup Language (SAML)	Standard, XML-based language for exchanging authentication and authorization data between identity providers and service providers.
Service provider (SP)	Entity that consumes assertions about a principal (and provides a service that the principal is trying to access).
Authentication Session	The interval while the user or entity is authenticating to AM.
Session	The interval that starts after the user has authenticated and ends when the user logs out, or when their session is terminated. For browser-based clients, AM manages user sessions across one or more applications by setting a session cookie. See also CTS-based sessions and Client-based sessions.

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