FORGEROCK®

Sessions Guide

/ ForgeRock Access Management 7.0.2

Latest update: 7.0.2

Copyright © 2019-2021 ForgeRock AS.

Abstract

Guide to understanding and configuring sessions in AM.

Table of Contents

Overview

This guide covers concepts and implementation procedures that will help you manage sessions in your AM environment.

This guide is written for administrators that are configuring AM's authentication and authorization components.

Quick Start

### Introducing Sessions	### The Session Cookie
Learn about the different types of sessions in AM and their characteristics.	Learn about the session cookie, and why you must secure it from malicious users.
### Comparing Sessions	### Session Upgrade
Decide where sessions should be stored in each realm based on your needs and the session characteristics.	Discover how AM performs step-up authentication.

About ForgeRock Identity Platform™ Software

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

Chapter 1
Introducing Sessions

A `session` in AM is a token that represents a usually interactive exchange of information between AM and a user or identity.

AM creates an authentication session to track the user's authentication progress through an authentication chain or tree. Once the user has authenticated, AM creates a session to manage the user's or entity's access to resources.

AM session-related services are stateless unless otherwise indicated; they do not hold any session information local to the AM instances. Instead, they store session information either in the CTS token store or on the client. This architecture allows you to scale your AM infrastructure horizontally since any server in the deployment can satisfy any session's request.

Sessions have different characteristics depending on where AM stores the sessions. Session storage location is configured at the realm level. The following table illustrates where AM can store sessions:

Session Storage Location

	In the CTS Token Store	**On the Client**	**In AM's Memory**
Authentication Sessions	✔ [a]	✔ [a] (Default in new installations)	✔ [b] (Default after upgrade)
Sessions	✔ (Default)	✔	✘

[a]Authentication trees only.
[b] Available for authentication trees and authentication chains.

> **Tip**
>
> Session storage location can be heterogeneous within the same AM deployment to suit the requirements of each of your realms.

Related information:

- "CTS-Based Sessions"

- "Client-Based Sessions"

- "In-Memory Sessions"

CTS-Based Sessions

CTS-based sessions reside in the CTS token store and can be cached in memory on one or more AM servers to improve system performance [1] . If the session request is redirected to an AM server that does not have the session cached, that server must retrieve the session from the CTS token store.

AM sends a reference to the session to the client, but the reference does not contain any of the session state information. AM can modify a session during its lifetime without changing the client's reference to the session.

- **CTS-Based Authentication Sessions Specifics**

  CTS-based authentication sessions are *supported for authentication trees only.*

  During authentication, the session reference is returned to the client after a call to the `authenticate` endpoint and stored in the `authId` object of the JSON response.

  AM maintains the authenticating user's session in the CTS token store. After the authentication flow has completed, if the realm to which the user has authenticated is configured for client-based sessions, AM returns session state to the client and deletes the CTS-based session.

  Authentication session whitelisting is an optional feature that maintains a list of in-progress authentication sessions and their progress in the authentication flow to protect against replay attacks. For more information, see "Configuring Authentication Session Whitelisting" in the *Security Guide*.

- **CTS-Based Sessions**

  Once the user is authenticated, the session reference is known as an *SSO token*. For browser clients, AM sets a cookie in the browser that contains the session reference. For REST clients, AM returns the session reference in response to calls to the `authentication` endpoint.

  For more information about session cookies, see "*Session Cookies and Session Security*".

Related information:

- "*Choosing Where to Store Sessions*"

Client-Based Sessions

Client-based sessions are those where AM returns session state to the client after each request, and require it to be passed in with the subsequent request.

---

[1] For information about configuring AM with sticky load balancing, see "*Load Balancers*" in the *Setup Guide*.

> **Important**
>
> Some features are not supported in realms configured for client-based sessions. For more information, see Limitations When Using Client-Based Sessions.

You should configure AM to sign and/or encrypt client-based sessions and authentication sessions for security reasons. As decrypting and verifying the session may be an expensive operation to perform on each request, AM caches [1] the decrypt sequence in memory to improve performance.

For more information about configuring client-based security, see "Configuring Client-Based Session Security" in the *Security Guide*.

- **Client-Based Authentication Sessions Specifics**

  Client-based authentication sessions are *supported for authentication trees only*, and are configured by default in new installations.

  During authentication, authentication session state is returned to the client after each call to the `authenticate` endpoint and stored in the `authId` object of the JSON response.

  After the authentication flow has completed, if the realm to which the user has authenticated is configured for CTS-based sessions, AM creates the user's session in the CTS token store. Then, AM attempts to invalidate the client-based authentication session.

  Storing authentication sessions on the client allows any AM server to handle the authentication flow at any point in time without load balancing requirements.

  Authentication session whitelisting is an optional feature that maintains a list of in-progress authentication sessions and their progress in the authentication flow to protect against replay attacks. For more information, see "Configuring Authentication Session Whitelisting" in the *Security Guide*.

- **Client-Based Sessions Specifics**

  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. For REST-based clients, AM sends the cookie in a header. For more information about session cookies, see "*Session Cookies and Session Security*".

  Session blacklisting is an optional feature that maintains a list of logged out client-based sessions in the CTS token store. For more information about session termination and session blacklisting, see "Understanding Session Termination" in the *Security Guide* and "Configuring Client-Based Session Blacklisting" in the *Security Guide*.

Related information:

- "*Choosing Where to Store Sessions*"

In-Memory Sessions

In-memory sessions reside in AM's memory. AM sends clients a reference to the session, but the reference does not contain any of the session state information.

- **In-Memory Authentication Sessions Specifics**

  In-memory authentication sessions are the *only configuration supported for authentication chains*. They are also configured by default for authentication trees after an upgrade.

  During authentication, the authentication session reference is returned to the client after a call to the `authenticate` endpoint and stored in the `authId` object of the JSON response.

  AM maintains the user's authentication session in its memory. After the authentication flow has completed, AM performs the following tasks:

  - If the realm to which the user has authenticated is configured for CTS-based sessions, AM stores the user's session in the CTS token store and deletes the authentication session from memory.

  - If the realm to which the user has authenticated is configured for client-based sessions, AM stores the user's session in a cookie on the user's browser and deletes the authentication session from memory.

  Authentication session whitelisting is an optional feature that maintains a list of in-progress authentication sessions and their progress in the authentication flow to protect against replay attacks. For more information, see "Configuring Authentication Session Whitelisting" in the *Security Guide*.

  > **Important**
  >
  > Deployments where AM stores authentication sessions in memory require sticky load balancing to route all requests pertaining to a particular authentication flow to the same AM server. If a request reaches a different AM server, the authentication flow will start anew.
  >
  > Authentication chains only support storing authentication sessions in memory. ForgeRock recommends switching to authentication trees with CTS-based or client-based authentication sessions.
  >
  > For information about configuring AM with sticky load balancing, see "*Load Balancers*" in the *Setup Guide*.

- **In-Memory Sessions Specifics**

  AM does not support in-memory sessions for authenticated users.

Related information:

- "*Choosing Where to Store Sessions*"

Chapter 2
Choosing Where to Store Sessions

You can configure authentication session storage location independently from session storage location. For example, you could configure the same realm for client-based authentication sessions and CTS-based sessions if it suits your environment.

AM stores CTS-based sessions in the CTS token store and caches sessions in server memory. If a server with cached sessions fails, or if the load balancer in front of AM servers directs a request to a server that does not have the user's session cached, the AM server retrieves the session from the CTS token store, incurring performance overhead.

Choosing where to store sessions is an important decision you must make by realm. Consider the information in the following tables before configuring sessions:

+ *Advantages of CTS-Based Sessions*

Advantage	Applies to Authentication Sessions?	Applies to Sessions?
Full Feature Support CTS-based sessions support all AM features, such as CDSSO and quotas. Client-based sessions do not. For information about restrictions on AM usage with client-based sessions, see Limitations When Using Client-Based Sessions. This advantage does not apply to authentication sessions, since they do not provide features.	–	✔
Session Information Is Not Resident In Browser Cookies With CTS-based sessions, all the information about the session resides in CTS and might be cached on one or more AM servers. With client-based sessions, session information is held in browser cookies. This information could be very long-lived.	✔	✔

+ *Advantages of Client-Based Sessions*

Advantage	Applies to Authentication Sessions?	Applies to Sessions?
Unlimited Horizontal Scalability for Session Infrastructure	✔	✔

Advantage	Applies to Authentication Sessions?	Applies to Sessions?
Client-based sessions provides unlimited horizontal scalability for your sessions by storing the session state on the client as a signed and encrypted JWT. Overall performance on hosts using client-based sessions can be easily improved by adding more hosts to the AM deployment.		
Replication-Free Deployments Global deployments may struggle to keep their CTS token store replication in sync when distances are long and updates are frequent. Client-based sessions are not constrained by the replication speed of the CTS token store. Therefore, client-based sessions are usually more suitable for deployments where a session can be serviced at any time by any server.	✔	✔

+ *Advantages of In-Memory Sessions*

Advantage	Applies to Authentication Sessions?	Applies to Sessions?
Faster Performance With Equivalent Host AM servers configured for in-memory authentication sessions can validate more sessions per second per host than those configured for client-based or CTS-based authentication sessions.	✔	✘
Session Information Is Not Resident in Browser Cookies Authentication session information resides in AM's memory and it is not accessible to users. With client-based sessions, authentication session information is held in browser cookies.	✔	✘

+ *Impact of Storage Location for Authentication Sessions*

	CTS-Based Authentication Sessions	Client-Based Authentication Sessions	In-Memory Authentication Sessions
Authentication Method	Authentication trees.	Authentication trees.	Authentication trees and authentication chains.
Session Location	Authoritative source: CTS token store. Sessions might also be cached in AM's memory for improved performance.	On the client.	In AM server's memory.

	CTS-Based Authentication Sessions	**Client-Based Authentication Sessions**	**In-Memory Authentication Sessions**
Load Balancer Requirements	None. Session stickiness recommended for performance.	None. Session stickiness recommended for performance.	Session stickiness.
Core Token Service Usage	Authoritative source for user sessions. Session whitelisting, when enabled.	Session whitelisting, when enabled.	None.
Uninterrupted Session Availability	No special configuration required.	No special configuration required.	Not available.
Session Security	Sessions reside in the CTS token store, and are not accessible to users.	Sessions reside on the client and should be signed and encrypted.	Sessions reside in AM's memory, and are not accessible to users.

+ *Impact of Storage Location for Sessions*

	CTS-Based Sessions	**Client-Based Sessions**
Hardware	Higher I/O and memory consumption.	Higher CPU consumption.
Logical Hosts	Variable or large number of hosts.	Variable or large number of hosts.
Session Monitoring	Available.	Not available.
Session Location	Authoritative source: CTS token store. Sessions might also be cached in AM's memory for improved performance.	In a cookie in the client.
Load Balancer Requirements	None. Session stickiness recommended for performance.	None. Session stickiness recommended for performance.
Uninterrupted Session Availability	No special configuration required.	No special configuration required.
Core Token Service Usage	Authoritative source for user sessions.	Provides session blacklisting for logged out sessions.
Core Token Service Demand	Heavier.	Lighter.
Session Security	Sessions reside in the CTS token store, and are not accessible to users.	Sessions should be signed and encrypted. [a]
Cross-Domain Single Sign-On Support	All AM capabilities supported.	Web Agents and Java Agents: Supported without restricted tokens.

[a] Web Agents and Java Agents support either signing or encrypting client-based sessions, but not both. For more information, see "Client-Based Session Security and Agents" in the *Security Guide*.

Chapter 3
Session Cookies and Session Security

Sessions require the user or client to be able to hold on to cookies. Cookies provided by AM's Session Service may contain a JSON Web Token (JWT) with the session or just a reference to where the session is stored.

AM issues a cookie to the user or entity regardless of the session location for client-based and CTS-based sessions. By default, the cookie's name is `iPlanetDirectoryPro`. For sessions stored in the CTS token store, the cookie contains a reference to the session in the CTS token store and several other pieces of information. For sessions stored on the client, the `iPlanetDirectoryPro` cookie contains all the information that would be held in the CTS token store.

Client-based session cookies are comprised of two parts. The first part of the cookie is identical to the cookie used by CTS-based sessions, which ensures the compatibility of the cookies regardless of the session location. The second part is a JSON Web Token (JWT), and it contains session information, as illustrated below:

- `iPlanetDirectoryPro` cookie for CTS-based sessions:

  ```
  AQIC...sswo.*AAJ...MA..*
  ```

- `iPlanetDirectoryPro` cookie for Client-based sessions:

  ```
  AQIC...sswo.*AAJ...MA..*ey...................................fQ.
  ```

Note that the examples are not to scale. The size of the client-based session cookie increases when you customize AM to store additional attributes in users' sessions. You are responsible for ensuring that the size of the cookie does not exceed the maximum cookie size allowed by your end users' browsers.

Since the session cookie is either a pointer to the actual user session or the session itself, you must configure AM to secure the session cookie against hijacking, session tampering, and other security concerns.

For example, terminating a session effectively logs the user or entity out of all realms, but the way AM terminates sessions has security implications depending on where AM stores the sessions. You can also configure the session time-to-live, idle timeout, the number of concurrent sessions for a user, and others.

Related information:

- "*Securing Sessions*" in the *Security Guide*

- "*Securing the Session Cookie*" in the *Security Guide*

- What information is contained in the AM/OpenAM session cookie?

Chapter 4
Configuring CTS-Based Sessions

By default, AM configures the CTS token store schema in the AM configuration store. Before configuring your AM deployment to use CTS-based sessions or authentication sessions, we recommend you install and configure an external CTS token store. For more information, see Core Token Service Guide (CTS).

CTS-based sessions and authentication sessions benefit from configuring sticky load balancing. For more information, see "*Load Balancers*" in the *Setup Guide*.

To configure CTS-based sessions and authentication sessions, see the following procedures:

- To Configure CTS-Based Authentication Sessions

- To Configure CTS-Based Sessions

To Configure CTS-Based Authentication Sessions

> **Important**
>
> Configuring storage location for authentication sessions is only supported for authentication trees. Authentication chains always store authentication sessions in AM's memory. For more information, see "*Introducing Sessions*".

1. Log in to the AM console as an administrative user. For example, `amAdmin`.

2. Navigate to Realms > *Realm Name* > Authentication > Settings > Trees.

3. From the Authentication session state management scheme drop-down list, select `CTS`.

4. In the Max duration (minutes) field, enter the maximum life of the authentication session in minutes.

5. Save your changes.

6. Navigate to Configure > Authentication > Core > Security.

7. In the Organization Authentication Signing Secret field, enter a base64-encoded HMAC secret that AM uses to sign the JWT that is passed back and forth between the client and AM during the authentication process. The secret must be at least 128-bits in length.

8. Save your changes.

To Configure CTS-Based Sessions

1. Log in to the AM console as an administrative user. For example, `amAdmin`.

2. Navigate to Realms > *Realm Name* > Authentication > Settings > General.

3. Ensure the Use Client-based Sessions check box is not selected.

4. Save your changes.

5. Verify that AM creates a CTS-based session when non-administrative users authenticate to the realm. Perform the following steps:

   a. Authenticate to AM as a non-administrative user in the realm you enabled for CTS-based sessions.

   b. In a different browser, authenticate to AM as an administrative user. For example, `amAdmin`.

   c. Navigate to Realms > *Realm Name* > Sessions.

   d. Verify that a session is present for the non-administrative user.

Chapter 5
Configuring Client-Based Sessions

Client-based sessions require additional setup in your environment to keep the sessions safe, and to ensure both the browser and the web server where AM runs can manage large cookies. Additionally, some of the AM features cannot be used with client-based sessions. Review the following list before configuring client-based sessions:

Planning for Client-Based Sessions

- Ensure the trust store used by AM has the necessary certificates installed:

    - A certificate is required for encrypting JWTs containing client-based sessions.

    - If you are using RS256 signing, then a certificate is required to sign JWTs. (HMAC signing uses a shared secret.)

    The same certificates must be stored on all servers participating in an AM site.

    For more information about managing certificates for AM, see "*Configuring Secrets, Certificates, and Keys*" in the *Security Guide*.

- Ensure that your users' browsers can accommodate larger session cookie sizes required by client-based sessions. For more information about session cookie sizes, see "*Session Cookies and Session Security*".

- Ensure that the AM web container can accommodate an HTTP header that is 16K in size or greater. When using Apache Tomcat as the AM web container, configure the `server.xml` file's `maxHttpHeaderSize` property to `16384` or higher.

- Ensure that your deployment does not require any of the capabilities specified in the list of limitations that apply to client-based sessions.

    + *Limitations When Using Client-Based Sessions*

    > Client-based sessions cannot use the following functionality:
    >
    > - **Session quotas**. See "Configuring Session Quotas" in the *Security Guide*.
    >
    > - **Session idle timeout**. See "Understanding Session Termination" in the *Security Guide*.
    >
    > - **Session upgrade with the `ForceAuth` parameter**. See "*Session Upgrade*".

- **Cross-domain single sign-on with restricted tokens (Web Agents and Java Agents)**. See "Enabling Restricted Tokens for CDSSO Session Cookies" in the *Security Guide*.

- **Both session signing and encryption (Web Agents and Java Agents)**. See "Client-Based Session Security and Agents" in the *Security Guide*.

- **Uncompressed sessions (Web Agents and Java Agents)**. See "Client-Based Session Security and Agents" in the *Security Guide*.

- **SAML v2.0 single logout using the SOAP binding**. See "Session State Considerations" in the *SAML v2.0 Guide*.

- **SNMP session monitoring.** See "SNMP Monitoring for Sessions" in the *Maintenance Guide*.

- **Session management by using the AM console.** See "*Managing Sessions (Console)*".

- **Session notification.** See "Session" in the *Reference*.

To configure client-based sessions and authentication sessions, see the following procedures:

- To Configure Client-Based Authentication Sessions

- To Configure Client-Based Sessions

To Configure Client-Based Authentication Sessions

> **Important**
>
> Configuring storage location for authentication sessions is only supported for authentication trees. Authentication chains always store authentication sessions in AM's memory. For more information, see "*Introducing Sessions*".

1. Log in to the AM console as an administrative user. For example, `amAdmin`.

2. Navigate to Realms > *Realm Name* > Authentication > Settings > Trees.

3. From the Authentication session state management scheme drop-down list, select `JWT`.

4. In the Max duration (minutes) field, enter the maximum life of the authentication session in minutes.

5. Save your changes.

6. Navigate to Configure > Authentication > Core > Security.

7. In the Organization Authentication Signing Secret field, enter a base64-encoded HMAC secret that AM uses to sign the JWT that is passed back and forth between the client and AM during the authentication process. The secret must be at least 128-bits in length.

8. Save your changes.

9. Protect your client-based authentication sessions. See "Configuring Client-Based Session Security" in the *Security Guide*.

To Configure Client-Based Sessions

1. Log in to the AM console as an administrative user. For example, `amAdmin`.

2. Navigate to Realms > *Realm Name* > Authentication > Settings > General.

3. Select the Use Client-based Sessions check box.

4. Save your changes.

5. Protect your client-based sessions. See "Configuring Client-Based Session Security" in the *Security Guide*.

6. Verify that AM creates a client-based session when non-administrative users authenticate to the realm. Perform the following steps:

   a. Authenticate to the AM console as the top-level administrator (by default, the `amAdmin` user). Note that sessions for the top-level administrator are always stored in the CTS token store.

   b. Navigate to Realms > *Realm Name* > Sessions.

   c. Verify that a session is present for the `amAdmin` user.

   d. In your browser, examine the AM cookie, named `iPlanetDirectoryPro` by default. Copy and paste the cookie's value into a text file and note its size.

   e. Start up a private browser session that will not have access to the `iPlanetDirectoryPro` cookie for the `amAdmin` user:

      • In Chrome, open an incognito window.

      • In Internet Explorer or Microsoft Edge, start InPrivate browsing.

      • In Firefox, open a new private window.

      • In Safari, open a new private window.

   f. Authenticate to AM as a non-administrative user in the realm for which you enabled client-based sessions. Be sure *not* to authenticate as the `amAdmin` user this time.

   g. In your browser, examine the `iPlanetDirectoryPro` cookie. Copy and paste the cookie's value into a second text file and note its size. The size of the client-based session cookie's value should be considerably larger than the size of the cookie used by the CTS-based session for the `amAdmin` user. If the cookie is not larger, you have not enabled client-based sessions correctly.

h. Return to the original browser window in which the AM console appears.

i. Refresh the window containing the Sessions page.

j. Verify that a session still appears for the `amAdmin` user, but that no session appears for the non-administrative user in the realm with client-based sessions enabled.

Chapter 6
Configuring In-Memory Authentication Sessions

Authentication chains always store authentication sessions in AM's memory. Perform the steps in the following procedure only for realms that configure authentication trees:

To Configure In-Memory Authentication Sessions

1. Ensure you have configured AM for sticky load balancing. For more information, see "*Load Balancers*" in the *Setup Guide*.

2. Log in to the AM console as an administrative user. For example, `amAdmin`.

3. Navigate to Realms > *Realm Name* > Authentication > Settings > Trees.

4. From the Authentication session state management scheme drop-down list, select `In-Memory`.

5. In the Max duration (minutes) field, enter the maximum life of the authentication session in minutes.

6. Save your changes.

7. Navigate to Configure > Authentication > Core > Security.

8. In the Organization Authentication Signing Secret field, enter a base64-encoded HMAC secret that AM uses to sign the JWT that is passed back and forth between the client and AM during the authentication process. The secret must be, at least, 128-bits in length.

9. Save your changes.

Chapter 7

Managing Sessions (Console)

The AM console lets the administrator view and manage active CTS-based user sessions by realm by navigating to Realms > *Realm Name* > Sessions.

Sessions Page

To search for active sessions, enter a username in the search box. AM retrieves the sessions for the user and displays them within a table. If no active CTS-based session is found, AM displays a session not found message.

You can end any sessions—except the current `amAdmin` user's session—by selecting it and clicking the Invalidate Selected button. As a result, the user has to authenticate again.

> **Important**
>
> Deleting a user does not automatically remove any of the user's CTS-based sessions. After deleting a user, check for any sessions for the user and remove them on the Sessions page.

> **Tip**
>
> Use the REST API for advanced functionality regarding sessions.

Chapter 8
Managing Sessions (REST)

AM provides REST APIs under `/json/sessions` for the following use cases:

- "Obtaining Information About Sessions Using REST"

- "Validating Sessions Using REST"

- "Refreshing CTS-Based Sessions Using REST"

- "Invalidating Sessions Using REST"

- "Getting and Setting Session Properties Using REST"

Obtaining Information About Sessions Using REST

To get information about a session, send an HTTP POST to the `/json/sessions/` endpoint, using the `getSessionInfo` action. This endpoint returns information about the session token provided in the `iPlanetDirectoryPro` header by default. To get information about a different session token, include it in the POST body as the value of the `tokenId` parameter.

> **Note**
>
> For information about how to retrieve custom session properties:
>
> - If you are using authentication modules, see "How do I retrieve user attributes from a session using the REST API?" in the *ForgeRock Knowledge Base*.
>
> - For authentication trees, use the Scripted Decision Node to retrieve user attributes and session properties, or the Set Session Properties Node for session properties only.

The following example shows an administrative user passing their session token in the `iPlanetDirectoryPro` header, and the session token of the `demo` user as the `tokenId` parameter:

```
$ curl \
--request POST \
--header "iPlanetDirectoryPro: AQICS...NzEz*" \
--header "Accept-API-Version: resource=4.0" \
--header "Content-Type: application/json" \
--data '{ "tokenId": "BXCCq...NX*1*" }' \
https://openam.example.com:8443/openam/json/realms/root/sessions/?_action=getSessionInfo
{
    "username": "demo",
    "universalId": "id=demo,ou=user,dc=openam,dc=forgerock,dc=org",
    "realm": "/",
    "latestAccessTime": "2020-02-21T14:31:18Z",
    "maxIdleExpirationTime": "2020-02-21T15:01:18Z",
    "maxSessionExpirationTime": "2020-02-21T16:29:56Z",
    "properties": {
        "AMCtxId": "aba7b4f3-16ff-4680-b06a-d7ba237d3730-91932"
    }
}
```

The `getSessionInfo` action does not refresh the session idle timeout. To obtain session information about a CTS-based session and also reset the idle timeout (you cannot reset the idle timeout of client-based sessions), use the `getSessionInfoAndResetIdleTime` endpoint, as follows:

```
$ curl \
--request POST \
--header "iPlanetDirectoryPro: AQICS...NzEz*" \
--header "Accept-API-Version: resource=4.0, protocol=1.0" \
--header "Content-Type: application/json" \
--data '{ "tokenId": "BXCCq...NX*1*" }' \
https://openam.example.com:8443/openam/json/realms/root/sessions/?_action=getSessionInfoAndResetIdleTime
{
    "username": "demo",
    "universalId": "id=demo,ou=user,dc=openam,dc=forgerock,dc=org",
    "realm": "/",
    "latestAccessTime": "2020-02-21T14:32:49Z",
    "maxIdleExpirationTime": "2020-02-21T15:02:49Z",
    "maxSessionExpirationTime": "2020-02-21T16:29:56Z",
    "properties": {
        "AMCtxId": "aba7b4f3-16ff-4680-b06a-d7ba237d3730-91932"
    }
}
```

Note

To return the `AMCtxId` property in the session query response, as in this example, you must set `AMCtxId` in the Session Properties to return to session queries setting, under Realms > *Realm Name* > Services > Session Property Whitelist Service.

Validating Sessions Using REST

To check over REST whether a session token is valid, perform an HTTP POST to the `/json/sessions/` endpoint using the `validate` action. Provide the session token in the POST data as the value of

the `tokenId` parameter. You must also provide the session token of an administrative user in the `iPlanetDirectoryPro` header.

If you don't specify the `tokenId` parameter, the session in the `iPlanetDirectoryPro` header is validated instead.

The following example shows an administrative user, such as `amAdmin`, validating a session token for the demo user:

```
$ curl \
--request POST \
--header "Content-type: application/json" \
--header "iPlanetDirectoryPro: AQICS...NzEz*" \
--header "Accept-API-Version: resource=2.1, protocol=1.0" \
--data '{ "tokenId": "BXCCq...NX*1*" }' \
https://openam.example.com:8443/openam/json/realms/root/sessions?_action=validate
```

If the session token is valid, the user ID and its realm is returned, as shown below:

```
{
  "valid":true,
  "sessionUid":"209331b0-6d31-4740-8d5f-740286f6e69f-326295",
  "uid":"demo",
  "realm":"/"
}
```

By default, validating a session resets the session's idle time, which triggers a write operation to the Core Token Service token store. To avoid this, perform a call using the `validate&refresh=false` action.

Refreshing CTS-Based Sessions Using REST

To reset the idle time of a CTS-based session using REST, perform an HTTP POST to the `/json/sessions/` endpoint, using the `refresh` action. The endpoint will refresh the session token provided in the `iPlanetDirectoryPro` header by default. To refresh a different session token, include it in the POST body as the value of the `tokenId` query parameter.

The following example shows an administrative user passing their session token in the `iPlanetDirectoryPro` header, and the session token of the demo user as the `tokenId` parameter:

```
$ curl \
--request POST \
--header 'Content-Type: application/json' \
--header 'Accept: application/json' \
--header "iPlanetDirectoryPro: AQICS...NzEz*" \
--header "Accept-API-Version: resource=3.1, protocol=1.0" \
--data '{ "tokenId": "BXCCq...NX*1*" }' \
https://openam.example.com:8443/openam/json/realms/root/sessions/?_action=refresh
{
  "uid": "demo",
  "realm": "/",
  "idletime": 17,
  "maxidletime": 30,
  "maxsessiontime": 120,
  "maxtime": 7106
}
```

On success, AM resets the idle time for the CTS-based session, and returns timeout details of the session.

Resetting a CTS-based session's idle time triggers a write operation to the Core Token Service token store. Therefore, to avoid the overhead of write operations to the token store, be careful to use the refresh action only if you want to reset a CTS-based session's idle time.

Because AM does not monitor idle time for client-based sessions, do not use the tokenId of a client-based session when refreshing a session's idle time.

Invalidating Sessions Using REST

To invalidate a session, perform an HTTP POST to the /json/sessions/ endpoint using the logout action. The endpoint will invalidate the session token provided in the iPlanetDirectoryPro header:

```
$ curl \
--request POST \
--header "iPlanetDirectoryPro: AQICS...NzEz*" \
--header "Accept-API-Version: resource=3.1, protocol=1.0" \
https://openam.example.com:8443/openam/json/realms/root/sessions/?_action=logout
{
    "result": "Successfully logged out"
}
```

On success, AM invalidates the session and returns a success message.

If the token is not valid and cannot be invalidated an error message is returned, as follows:

```
{
  "result": "Token has expired"
}
```

To invalidate a different session token, include it in the POST body as the value of the tokenId parameter.

For example, the following command shows an administrative user passing their session token in the `iPlanetDirectoryPro` header, and the session token of the `demo` user as the `tokenId` parameter:

```
$ curl \
--request POST \
--header "iPlanetDirectoryPro: AQICS...NzEz*" \
--header "Accept-API-Version: resource=3.1, protocol=1.0" \
--header "Content-Type: application/json" \
--data '{ "tokenId": "BXCCq...NX*1*" }' \
"https://openam.example.com:8443/openam/json/realms/root/sessions/?_action=logout"
{
    "result": "Successfully logged out"
}
```

Getting and Setting Session Properties Using REST

AM lets you read and update properties on users' sessions using REST API calls.

Before you can perform operations on session properties using the REST API, you must first define the properties you want to set in the Session Property Whitelist Service configuration. For information on whitelisting session properties, see "Session Property Whitelist Service" in the *Reference*.

You can use REST API calls for the following purposes:

- To retrieve the names of the properties that you can read or update. This is the same set of properties configured in the Session Property Whitelist Service.

- To read property values.

- To update property values.

Session state affects the ability to set and delete properties as follows:

- You can set and delete properties on a CTS-based session at any time during the session's lifetime.

- You can only set and update properties on a client-based session during the authentication process, before the user receives the session token from AM. For example, you could set or delete properties on a client-based session from within a post-authentication plugin.

Differentiate the user who performs the operation on session properties from the session affected by the operation as follows:

- Specify the session token of the user performing the operation on session properties in the `iPlanetDirectoryPro` header.

- Specify the session token of the user whose session is to be read or modified as the `tokenId` parameter in the body of the REST API call.

- Omit the `tokenId` parameter from the body of the REST API call if the session of the user performing the operation is the same session that you want to read or modify.

The following examples assume that you configured a property named `LoginLocation` in the Session Property Whitelist Service configuration.

To retrieve the names of the properties you can get or set, and their values, perform an an HTTP POST to the sessions endpoint, `/json/sessions/`, using the `getSessionProperties` action, as shown in the following example:

```
$ curl \
--request POST \
--header "iPlanetDirectoryPro: AQICS...NzEz*" \
--header "Accept-API-Version: resource=3.1, protocol=1.0" \
--header "Content-Type: application/json" \
--data '{ "tokenId": "BXCCq...NX*1*" }' \
https://openam.example.com:8443/openam/json/realms/root/sessions/?_action=getSessionProperties
{
    "LoginLocation": ""
}
```

To set the value of a session property, perform an HTTP POST to the sessions endpoint, `/json/sessions/`, using the `updateSessionProperties` action. If no `tokenId` parameter is present in the body of the REST API call, the session affected by the operation is the session specified in the `iPlanetDirectoryPro` header, as follows:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "iPlanetDirectoryPro: AQICS...NzEz*" \
--header "Accept-API-Version: resource=3.1, protocol=1.0" \
--data '{"LoginLocation":"40.748440, -73.984559"}' \
https://openam.example.com:8443/openam/json/realms/root/sessions/?_action=updateSessionProperties
{
    "LoginLocation": "40.748440, -73.984559"
}
```

You can set multiple properties in a single REST API call by specifying a set of fields and their values in the JSON data. For example:

```
--data '{"property1":"value1", "property2":"value2"}'
```

To set the value of a session property on another user's session, specify the session token of the user performing the `updateSessionProperties` action in the `iPlanetDirectoryPro`, and specify the session token to be modified in the POST body as the value of the `tokenId` parameter:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "iPlanetDirectoryPro: AQICS...NzEz*" \
--header "Accept-API-Version: resource=3.1, protocol=1.0" \
--data '{"LoginLocation": "40.748440, -73.984559", "tokenId": "BXCCq...NX*1*"}' \
https://openam.example.com:8443/openam/json/realms/root/sessions/?_action=updateSessionProperties
{
    "LoginLocation": "40.748440, -73.984559"
}
```

If the user attempting to modify the session does not have sufficient access privileges, the preceding examples result in a 403 Forbidden error.

You cannot set properties internal to AM sessions. If you try to modify an internal property in a REST API call, a 403 Forbidden error is returned. For example:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--header "iPlanetDirectoryPro: AQICS...NzEz*" \
--header "Accept-API-Version: resource=3.1, protocol=1.0" \
--data '{"AuthLevel":"5", "tokenId": "BXCCq...NX*1*"}' \
https://openam.example.com:8443/openam/json/realms/root/sessions/?_action=updateSessionProperties
{
    "code": 403,
    "reason": "Forbidden",
    "message": "Forbidden"
}
```

Chapter 9
Session Upgrade

Sessions can be upgraded to provide access to sensitive resources.

Consider a website for a University. Some information, such as courses and degree catalogs, are free for anyone to see and therefore, do not need to be protected. The University also provides the students with a portal they can use to see their grades, which is protected with a policy that requires users to authenticate. However, to pay tuition, students are required to present additional credentials to increase their authentication level and gain access to these functions.

Allowing authenticated users to provide additional credentials to access sensitive resources is called *session upgrade*, which is AM's mechanism to perform step-up authentication.

+ *What Triggers a Session Upgrade?*

- An authenticated user being redirected to a URL that has the `ForceAuth` parameter set to `true`. For example, `https://openam.example.com:8443/openam/XUI/?realm=/myRealm&ForceAuth=true#login`

  In this case, the user is asked to reauthenticate to the default chain in the realm `myRealm`.

  > **Important**
  >
  > Session upgrade using the `ForceAuth` parameter is only supported for CTS-based sessions.

- An authenticated user trying to access a resource protected by a web or Java agent (or a custom policy enforcement point (PEP)). In this case, AM sends the agent or PEP *advice* that the user should perform one of the following actions:

  - Authenticate at an authentication level greater than the current level

  - Authenticate to a module

  - Authenticate to a service

  The flow of the session upgrade during policy evaluation is as follows:

  1. An authenticated user tries to access a resource.

  2. The PEP, for example a web or Java agent, sends the request to AM for policy decision.

3. AM returns an authorization decision that denies access to the resource, and returns an *advice* indicating that the user needs to present additional credentials to access the resource.

4. The policy enforcement point sends the user back to AM for session upgrade.

5. The user provides additional credentials. For example, they may provide a one-time password, swipe their phone screen, or use face recognition.

6. AM authenticates the user.

7. The user can now access the sensitive resource.

+ *Session Upgrade Outcomes*

- **Successful**. One of the following will happen depending on the type of sessions configured for the realm:

  - If the realm is configured for CTS-based sessions, one of the following will happen depending on the mechanism used to perform session upgrade:

    - When using the `ForceAuth` parameter, AM does one of the following:

      - (Authentication trees only) AM issues new session tokens to users on reauthentication, even if the current session already meets the security requirements.

      - (Authentication chains only) AM does not issue new session tokens on reauthentication, regardless of the security level they are authenticating to. Instead, it updates the session token with the new authentication information, if required.

    - When using *advices*, AM copies the session properties to a new session and hands the client a new session token to replace the original one. The new session reflects the successful authentication to a higher level.

  - If the realm is configured for client-based sessions, AM hands the client a new session token to replace the original one. The new session reflects the successful authentication to a higher level.

- **Unsuccessful**. AM leaves the user session as it was before the attempt at stronger authentication. If session upgrade failed because the login page times out, AM redirects the user's browser to the success URL from the last successful authentication.

Tip

Anonymous sessions can also be upgraded to non-anonymous sessions by using the "Anonymous Session Upgrade Node" in the *Authentication and Single Sign-On Guide*.

Session Upgrade Prerequisites

- Configure a policy enforcement point (PEP), for example, a web or Java agent, that enforces AM policies on a website or application.

  AM web and Java agents handle session upgrade without additional configuration because the agents are built to handle AM's advices. If you build your own PEPs, however, you must take advices and session upgrade into consideration.

  + *Resources*

    - *ForgeRock Web Agent User Guide*.

    - *ForgeRock Java Agent User Guide*.

    - "Requesting Policy Decisions Using REST" in the *Authorization Guide* (For RESTful PEPs).

- Configure an authorization policy to protect a resource protected by the Java or web agent, or a RESTful PEP.

  + *Example*

    The following policy allows GET and POST access to the `*://*:*/sample/*` resource to any authenticated user:

Authorization Policy Example

Tasks:

- "To Configure the Environment for Session Upgrade"

- "To Perform Session Upgrade Using a Browser"

- "To Perform Session Upgrade Using REST"

To Configure the Environment for Session Upgrade

1. Configure an authentication tree or chain to validate users' credentials during session upgrade.

   Authentication trees and chains do not require additional configuration to perform session upgrade. However, because session upgrade is a mechanism which may be used to grant users access to sensitive information, you should consider configuring a strong authentication method such as multi-factor authentication. Also, you may want to consider how long-lived sessions in your environment are. For example, if users should only have access to the protected resource to perform an operation, such as check the balance of an account, you may want to consider implementing transactional authorization instead.

   - For more information, see "*Transactional Authorization*" in the *Authorization Guide*.

- For more information about configuring authentication trees and chains, see the Authentication and Single Sign-On Guide.

2. Configure at least one of the following environment conditions in the authentication policy you created as part of the prerequisites:

+ *Authentication Level (greater than or equal to) (Authentication modules only)*

Use this condition to present a list of authentication chains that provide a greater or equal authentication level to the one specified in the condition. The user selects their service of choice if multiple chains are able to meet the criteria of the condition. For example, the following policy requires a chain that provides authentication level 3 or greater:

Session Upgrade by Authentication Level (greater than or equal to)

| Summary | Resources | Actions | Subjects | Environments | ▾ |

Resources ✏

`*://*:*/sample/*`

Actions ✏

POST: Allowed GET: Allowed

Response Attributes ✏

➕ Add Response Attributes

Subjects ✏

```
{
    "type": "AuthenticatedUsers"
}
```

Environments ✏

```
{
    "type": "AuthLevel",
    "authLevel": 3
}
```

> **Tip**
>
> For more information about configuring the authentication level by authentication module, see "About Authentication Levels for Chains" in the *Authentication and Single Sign-On Guide*.

+ *Authentication by Service*

Use this condition to specify the chains or authentication trees to which the user needs to use to authenticate. For example, the following policy requires the user to log in with the `Example` tree:

Session Upgrade by Service

Note that the names of the authentication trees and chains are case-sensitive.

+ *Authentication by Module Instance (Authentication modules only)*

Use this condition to enforce that a user has gone through a specific authentication module. For example, the following policy requires the user to log in with the `DataStore` module:

FORGEROCK

Session Upgrade by Module Instance

3. Test session upgrade:

   • To test session upgrade with a browser, see "To Perform Session Upgrade Using a Browser".

   • To test session upgrade with REST, see "To Perform Session Upgrade Using REST".

To Perform Session Upgrade Using a Browser

To upgrade a session using a browser, perform the following steps:

1. Ensure you have performed the tasks in Session Upgrade Prerequisites and "To Configure the Environment for Session Upgrade".

2. In a browser, navigate to your protected resource. For example, `http://www.example.com:9090/sample`.

   The agent redirects the browser to the AM login screen.

3. Authenticate to AM as the `demo` user.

   AM requires additional credentials to grant access to the resource. For example, if you set the policy environment condition to `Authentication by Service` and `Example`, you will be required to log in again as the `demo` user.

4. Authenticate as the `demo` user. Note that providing credentials for a different user will fail.

   You can now access the protected resource.

To Perform Session Upgrade Using REST

To upgrade a session using REST, perform the following steps:

1. Ensure you have performed the tasks in Session Upgrade Prerequisites and "To Configure the Environment for Session Upgrade".

   > **Note**
   >
   > This example uses composite advice with an authentication level condition, which only applies to authentication chains.

2. Log in with an administrative user that has permission to evaluate policies, such as `amAdmin`. For example:

   ```
   $ curl \
   --request POST \
   --header "Content-Type: application/json" \
   --header "X-OpenAM-Username: amadmin" \
   --header "X-OpenAM-Password: password" \
   --header "Accept-API-Version: resource=2.0, protocol=1.0" \
   'https://openam.example.com:8443/openam/json/realms/root/authenticate'
        {
       "tokenId":"AQIC5wM2...",
       "successUrl":"/openam/console",
       "realm":"/"
   }
   ```

   > **Tip**
   >
   > You can also assign privileges to a user to evaluate policies. For more information, see "To Allow a User to Evaluate Policies" in the *Authorization Guide*.

3. Log in with the user that should access the resources. For example, log in as the `demo` user:

```
$ 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/authenticate'
    {
    "tokenId":"AQIC5wM...TU3OQ*",
    "successUrl":"/openam/console",
    "realm":"/"
}
```

4. Request a policy decision from AM for a protected resource, in this case, `http://openam.example.com:9090/sample`. The `iPlanetDirectoryPro` header sets the SSO token for the administrative user, and the `subject` element of the payload sets the SSO token for the `demo` user:

```
$ curl --request POST \
 --header "Content-Type: application/json" \
 --header "iPlanetDirectoryPro: AQIC5wM2..." \
 --header "Accept-API-Version:protocol=1.0,resource=2.1" \
 --data '{
 "resources": [
     "http://www.example.com:9090/sample"
 ],
 "application": "iPlanetAMWebAgentService",
 "subject": { "ssoToken": "AQIC5wM...TU3OQ*"}
}' \
"https://openam.example.com:8443/openam/json/policies?_action=evaluate"
     [
   {
       "resource":"http://www.example.com:9090/sample",
       "actions":{

       },
       "attributes":{

       },
       "advices":{
          "AuthLevelConditionAdvice":[
             "3"
          ]
       },
       "ttl":9223372036854775807
   }
]
```

AM returns with advice, which means the user must present additional credentials to access that resource.

For more information about requesting policy decision, see "Requesting Policy Decisions Using REST" in the *Authorization Guide*.

5. Format the advice as XML, without spaces or line breaks. The following example is spaced and tabulated for readability purposes only:

```
<Advices>
    <AttributeValuePair>
        <Attribute name="AuthLevelConditionAdvice"/>
        <Value>3</Value>
    </AttributeValuePair>
</Advices>
```

> **Note**
>
> The example shows the XML render of a single advice. Depending on the conditions configured in the policy, the advice may contain several lines. For more information about advices, see "Policy Decision Advice" in the *Authorization Guide*.

6. URL-encode the XML advice. For example: `%3CAdvices%3E%3CAttributeValuePair%3E%3CAttribute%20name`
   `%3D%22AuthLevelConditionAdvice%22%2F%3E%3CValue%3E3%3C%2FValue%3E%3C%2FAttributeValuePair%3E%3C`
   `%2FAdvices%3E`.

   Ensure there are no spaces between tags when URL-encoding the advice.

7. Call AM's `authenticate` endpoint to request information about the advice. Use the following details:

   • Add the following URL parameters:

     • `authIndexType=composite_advice`

     • `authIndexValue=URL-encoded_Advice`

   • Set the `iPlanetDirectoryPro` cookie as the SSO token for the `demo` user.

   For example:

```
$ curl --request POST \
--header "Content-Type: application/json" \
--cookie "iPlanetDirectoryPro=AQIC5wM...TU3OQ*" \
--header "Accept-API-Version: protocol=1.0,resource=2.1" \
'https://openam.example.com:8443/openam/json/realms/root/authenticate?
authIndexType=composite_advice&amp;authIndexValue=%3CAdvices%3E%3CAttributeValuePair%3E...'
    {
    "authId":"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhdXRoSW5kZXZ...",
    "template":"",
    "stage":"DataStore1",
    "header":"Sign in",
    "callbacks":[
        {
            "type":"NameCallback",
            "output":[
                {
                    "name":"prompt",
                    "value":"User Name:"
                }
            ],
            "input":[
```

```
            {
                "name":"IDToken1",
                "value":""
            }
        ]
    },
    {
        "type":"PasswordCallback",
        "output":[
            {
                "name":"prompt",
                "value":"Password:"
            }
        ],
        "input":[
            {
                "name":"IDToken2",
                "value":""
            }
        ]
    }
  ]
}
```

AM returns information about how the user can authenticate in a callback; in this case, providing a username and password. For a list of possible callbacks, and more information about the /json/authenticate endpoint, see "*Authenticating (REST)*" in the *Authentication and Single Sign-On Guide*.

8. Call AM's authenticate endpoint to provide the required callback information. Use the following details:

   • Add the following URL query parameters:

      • authIndexType=composite_advice

      • authIndexValue=*URL-encoded_Advice*

   • Set the iPlanetDirectoryPro cookie as the SSO token for the demo user.

   • Send as data the complete payload AM returned in the previous step, ensuring you provide the requested callback information.

   In this example, provide the username and password for the demo user in the input objects, as follows:

```
$ curl --request POST \
     --header 'Content-Type: application/json' \
     --header "Accept-API-Version: protocol=1.0,resource=2.1" \
     --cookie "iPlanetDirectoryPro=AQIC5wM...TU3OQ*" \
     --data '{
                "authId":"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhdXRoSW5kZ...",
                "template":"",
                "stage":"DataStore1",
                "header":"Sign in",
```

```
                "callbacks":[
                    {
                        "type":"NameCallback",
                        "output":[
                            {
                                "name":"prompt",
                                "value":"User Name:"
                            }
                        ],
                        "input":[
                            {
                                "name":"IDToken1",
                                "value":"demo"
                            }
                        ]
                    },
                    {
                        "type":"PasswordCallback",
                        "output":[
                            {
                                "name":"prompt",
                                "value":"Password:"
                            }
                        ],
                        "input":[
                            {
                                "name":"IDToken2",
                                "value":"Ch4ng31t"
                            }
                        ]
                    }
                ]
            }
        }' \
    'https://openam.example.com:8443/openam/json/realms/root/authenticate?
authIndexType=composite_advice&amp;authIndexValue=%3CAdvices%3E%3CAttributeValuePair%3E...'
    {
    "tokenId":"wpU01SaTq4X2x...NDVFMAAlMxAAA.*",
    "successUrl":"/openam/console",
    "realm":"/"
}
```

Note that AM returns a new SSO token for the demo user.

9. Request a new policy decision from AM for the protected resource. The `iPlanetDirectoryPro` header sets the SSO token for the administrative user, and the subject element of the payload sets the new SSO token for the demo user:

```
$ curl --request POST \
--header "Content-Type: application/json" \
--header "iPlanetDirectoryPro: AQIC5wM2..." \
--header "Accept-API-Version:protocol=1.0,resource=2.1" \
--data '{
    "resources":[
        "http://www.example.com:9090/sample"
    ],
    "application":"iPlanetAMWebAgentService",
    "subject":{
        "ssoToken":"wpU01SaTq4X2x...NDVFMAAlMxAAA.*"
    }
}' \
"https://openam.example.com:8443/openam/json/policies?_action=evaluate"
        [
    {
        "resource":"http://www.example.com:9090/sample",
        "actions":{
            "POST":true,
            "GET":true
        },
        "attributes":{

        },
        "advices":{

        },
        "ttl":9223372036854775807
    }
]
```

AM returns that `demo` can perform `POST` and `GET` operations on the resource.

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.