Sessions

These topics cover concepts and implementation procedures to manage sessions in your AM environment.

This information is written for administrators configuring AM's authentication and authorization components.

Introduction to sessions

Learn about the different types of sessions in AM.

Session upgrade

Discover how AM performs step-up authentication.

Compare sessions

Decide where sessions should be stored in each realm.

The session cookie

Learn about the session cookie, and why you must secure it.

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.

Introduction to 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	✔[1]	✔[1] (Default in new installations)	✔[2] (Default after upgrade)
Sessions	✔ (Default)	✔	✖

[1] Authentication trees only.

[2] 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.

Server-side sessions

Server-side sessions reside in the CTS token store and can be cached in memory on one or more AM servers to improve system performance.

TIP

For information about configuring AM with sticky load balancing, see Load balancing.

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.

- **Server-side authentication sessions**

  Server-side 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-side sessions, AM returns session state to the client and deletes the server-side session.

  Authentication session allowlisting 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 Authentication session allowlisting.
- **Server-side 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: Choose where to store sessions

Client-side sessions

For *client-side sessions*, AM returns the session state to the client after each request and requires the session state to be passed in with the subsequent request.

IMPORTANT

Some features are not supported in realms configured for client-side sessions. For more information, refer to Limitations of using client-side sessions.

For security reasons, configure AM to sign and/or encrypt client-side sessions and client-side authentication sessions. Decrypting and verifying the session can be an expensive operation to perform on each request. AM therefore caches the decrypt sequence in memory to improve performance.

TIP

Find information about configuring AM with sticky load balancing in Load balancing.

Find information about configuring client-side security in Client-side session security.

- **Client-side authentication sessions**

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

  During authentication, the 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.

  If the realm the user authenticated to is configured for server-side sessions, AM creates the user's session in the CTS token store when the authentication flow completes.

  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 allowlisting 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. Learn more in Authentication session allowlisting.
- **Client-side sessions**

  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. Find more information about session cookies in Session cookies and session security.

  Session denylisting is an optional feature that maintains a list of logged out client-side sessions in the CTS token store. Find more information about session termination and session denylisting in Session termination.

IMPORTANT

IMPORTANT

A user is granted a client-side authentication session *while they are completing the authentication tree*. If session denylisting is enabled, this authentication session is "logged out" when the tree is completed, to prevent replay attacks. This "logging out" adds the authentication session to the session *denylist* for client-side sessions. In the CTS store, this takes the form of a `SESSION_BLACKLIST` token that exists for the life of the authentication session.

Learn more in Choose 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**

  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 server-side 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-side sessions, AM stores the user’s session in a cookie on the user’s browser and deletes the authentication session from memory.

  Authentication session allowlisting 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 Authentication session allowlisting.

  IMPORTANT

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 server-side or client-side authentication sessions.

For information about configuring AM with sticky load balancing, see Load balancing.

- **In-memory sessions**

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

Related information: Choose where to store sessions

Session termination

AM manages active sessions, allowing single sign-on when authenticated users attempt to access system resources in AM's control.

AM ensures that user sessions are terminated when a configured timeout is reached, or when AM users perform actions that cause session termination. Session termination effectively logs the user out of all systems protected by AM.

With server-side sessions, AM terminates sessions in four situations:

- When a user explicitly logs out.
- When an administrator monitoring sessions explicitly terminates a session.
- When a session exceeds the maximum time-to-live.
- When a user is idle for longer than the maximum session idle time.

Under these circumstances, AM responds by removing server-side sessions from the CTS token store and from AM server memory caches. With the user's session no longer present in CTS, AM forces the user to reauthenticate during subsequent attempts to access resources protected by AM.

When a user explicitly logs out of AM, AM also attempts to invalidate the `iPlanetDirectoryPro` cookie in users' browsers by sending a `Set-Cookie` header with an invalid session ID and a cookie expiration time that is in the past. In the case of administrator session termination and session timeout, AM cannot invalidate the `iPlanetDirectoryPro` cookie until the next time the user accesses AM.

Session termination differs for client-side sessions. Since client-side sessions are not maintained in the CTS token store, administrators cannot monitor or terminate them. Because AM does not modify the `iPlanetDirectoryPro` cookie for client-side sessions after authentication, the session idle time is not maintained in the cookie. Therefore, AM does not automatically terminate client-side sessions that have exceeded the idle timeout.

As with server-side sessions, AM attempts to invalidate the `iPlanetDirectoryPro` cookie from a user's browser when the user logs out. When the maximum session time is exceeded, AM also attempts to invalidate the `iPlanetDirectoryPro` cookie in the user's browser the next time the user accesses AM.

It is important to understand that AM cannot guarantee cookie invalidation. For example, the HTTP response containing the `Set-Cookie` header might be lost. This is not an issue for server-side sessions, because a logged out session no longer exists in the CTS token store, and a user who attempts to access AM after previously logging out will be forced to reauthenticate.

However, the lack of a guarantee of cookie invalidation is an issue for deployments with client-side sessions. It could be possible for a logged out user to have an `iPlanetDirectoryPro` cookie. AM could not determine that the user previously logged out. Therefore, AM supports a feature that takes additional action when users log out of client-side sessions. AM can maintain a list of logged out client-side sessions in a session denylist in the CTS token store. Whenever users attempt to access AM with client-side sessions, AM checks the session denylist to validate that the user has not, in fact, logged out.

Since AM does not modify client-side session cookies once they are stored in the end user's browser, and client-side sessions contain, among others, the session maximum time-to-live, it is imperative to protected them against tampering. See Client-side session security for more information.

Configure maximum session time-to-live

When configuring the maximum session time-to-live, you must balance security and user experience. Depending on your application, it may be acceptable for your users to log in once a month. Financial applications, for example, tend to expire their sessions in less than an hour.

The longer a session is valid, the larger the window during which a malicious user could impersonate a user if they were able to hijack a session cookie.

1. In the AM admin UI, go to **Realms** > ***Realm Name*** > **Services** > **Session** > **Dynamic Attributes**.

   Note that you can also change maximum session time settings globally for the AM site at **Configure** > **Sessions** > **Dynamic Attributes**.

2. In the **Maximum Session Time** field, set a value suitable for your environment.
3. Save your changes.

Configure server-side session idle timeout

Consider a user with a valid session navigating through pages or making changes to the configuration. If for any reason they leave their desk and their computer remains open, a malicious user could take the opportunity to impersonate them.

Session idle timeout can help mitigate those situations, by logging out users after a specified duration of inactivity. Session idle timeout can only be used in realms configured for server-side sessions.

1. In the AM admin UI, go to **Realms > *Realm Name* > Services > Session > Dynamic Attributes**.

   Note that you can also change idle timeout settings globally for the AM site by navigating to **Configure > Sessions > Dynamic Attributes**.
2. On the **Maximum Time Idle** property, configure a value suitable for your environment.
3. Save your changes.

Configure client-side session denylisting

Session denylisting ensures that users who have logged out of client-side sessions cannot achieve single sign-on without reauthenticating to AM. Session denylisting does not apply to authentication sessions.

1. Make sure that you deployed the Core Token Service during AM installation.

   The session denylist is stored in the Core Token Service’s token store.
2. Go to **Configure > Global Services**, click Session, and locate the **Client-Side Sessions** tab.
3. Select the **Enable Session Denylisting** option to enable session denylisting for client-side sessions.

   When you configure one or more AM realms for client-side sessions, you should enable session denylisting in order to track session logouts across multiple AM servers.

   Changing the value of this property takes effect immediately.
4. Configure the **Session Denylist Cache Size** property.

   AM maintains a cache of logged out client-side sessions. The cache size should be around the number of logouts expected in the maximum session time. Change the default value of 10,000 when the expected number of logouts during the maximum session time is an order of magnitude greater than 10,000. An underconfigured session denylist cache causes AM to read denylist entries from the Core Token Service store instead of obtaining them from cache, which results in a small performance degradation.

   Changing the value of this property takes effect immediately.

5. Configure the **Denylist Poll Interval** property.

   AM polls the Core Token service for changes to logged out sessions if session denylisting is enabled. By default, the polling interval is 60 seconds. The longer the polling interval, the more time a malicious user has to connect to other AM servers in a cluster and make use of a stolen session cookie. Shortening the polling interval improves the security for logged out sessions, but might incur a minimal decrease in overall AM performance due to increased network activity.

   Changing the value of this property does not take effect until you restart AM.

6. Configure the **Denylist Purge Delay** property.

   When session denylisting is enabled, AM tracks each logged out session for the maximum session time plus the denylist purge delay. For example, if a session has a maximum time of 120 minutes and the denylist purge delay is one minute, then AM tracks the session for 121 minutes. Increase the denylist purge delay if you expect system clock skews in a cluster of AM servers to be greater than one minute. There is no need to increase the denylist purge delay for servers running a clock synchronization protocol, such as Network Time Protocol.

   Changing the value of this property does not take effect until you restart AM.

7. Click **Save Changes**.

   IMPORTANT

   Enabling or disabling the session denyist, or altering the cache size, takes effect immediately.

   Changes to any other session denylist properties **do not** take effect until you restart AM.

For detailed information about session service configuration attributes, see the entries for Session.

Choose 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-side authentication sessions and server-side sessions if it suits your environment.

AM stores server-side 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 server-side sessions

Advantage	Applies to authentication sessions?	Applies to sessions?
Full feature support Server-side sessions support all AM features, such as CDSSO and quotas. Client-side sessions do not. For information about restrictions on AM usage with client-side sessions, see Limitations of client-side sessions. This advantage does not apply to authentication sessions, since they do not provide features.	—	✔

Advantage	Applies to authentication sessions?	Applies to sessions?
Session information is not resident in browser cookies With server-side sessions, all the information about the session resides in CTS and might be cached on one or more AM servers. With client-side sessions, session information is held in browser cookies. This information could be very long-lived.	✔	✔

▼ Advantages of client-side sessions

Advantage	Applies to authentication sessions?	Applies to sessions?
Unlimited horizontal scalability for session infrastructure Client-side 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-side sessions can be easily improved by adding more hosts to the AM deployment.	✔	✔

Advantage	Applies to authentication sessions?	Applies to sessions?
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-side sessions are not constrained by the replication speed of the CTS token store. Therefore, client-side 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-side or server-side authentication sessions.	✔	✖

Advantage	Applies to Authentication Sessions?	Applies to 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-side sessions, authentication session information is held in browser cookies.	✔	✖

▼ Impact of storage location for authentication sessions

	Server-side authentication sessions	**Client-side 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.
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 allowlisting, when enabled.	Session allowlisting, when enabled.	None.

	Server-side authentication sessions	Client-side authentication sessions	In-memory authentication sessions
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

	Server-side Sessions	Client-side 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 denylisting for logged out sessions.
Core token service demand	Heavier.	Lighter.

	Server-side Sessions	Client-side Sessions
Session security	Sessions reside in the CTS token store, and are not accessible to users.	Sessions should be signed and encrypted.(1)
Cross-domain single sign-on support	All AM capabilities supported.	Web agents and Java agents: Supported without restricted tokens.

(1) Web agents and Java agents support either signing or encrypting client-side sessions, but not both. For more information, see Client-side session security and agents.

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-side and server-side 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-side session cookies are comprised of two parts. The first part of the cookie is identical to the cookie used by server-side 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 server-side sessions:

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

- `iPlanetDirectoryPro` cookie for client-side sessions:

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

Note that the examples are not to scale. The size of the client-side 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:

- Secure sessions
- Secure session cookies
- What information is contained in the AM session cookie?

Configure server-side sessions

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

Server-side sessions and authentication sessions benefit from configuring sticky load balancing. For more information, see Load balancing.

Configure server-side 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 Introduction to sessions.

1. In the AM admin UI, go to **Realms** > ***Realm Name*** > **Authentication** > **Settings** > **Trees**.
2. From the **Authentication session state management scheme** drop-down list, select `CTS` .
3. In the **Max duration (minutes)** field, enter the maximum life of the authentication session in minutes.
4. Save your changes.

5. Go to **Configure > Authentication > Core > Security**.
6. 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.
7. Save your changes.

Configure server-side sessions

1. In the AM admin UI, go to **Realms > *Realm Name* > Authentication > Settings > General**.
2. Ensure the **Use Client-Side Sessions** check box is not selected.
3. Save your changes.
4. Verify that AM creates a server-side session when non-administrative users authenticate to the realm. Perform the following steps:
    - Authenticate to AM as a non-administrative user in the realm you enabled for server-side sessions.
    - In a different browser, authenticate to AM as an administrative user. For example, `amAdmin`.
    - Go to **Realms > *Realm Name* > Sessions**.
    - Verify that a session is present for the non-administrative user.

Configure client-side sessions

Client-side 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-side sessions. Review the following list before configuring client-side sessions:

Planning for client-side sessions

- Ensure the trust store used by AM has the necessary certificates installed:
    - A certificate is required for encrypting JWTs containing client-side 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 Secrets, certificates, and keys.
- Ensure that your users' browsers can accommodate larger session cookie sizes required by client-side 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-side sessions.

▼ *Limitations of client-side sessions*

Client-side sessions cannot use the following functionality:

- Session quotas
- Session idle timeout
- Cross-domain single sign-on with restricted tokens (web agents and Java agents)
- Session signing and encryption (web agents and Java agents)
- Uncompressed sessions (web agents and Java agents)
- SAML v2.0 single logout using the SOAP binding
- SNMP session monitoring
- Session management using the AM admin UI
- Session notification
- Refresh token grace period

Configure client-side 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 Introduction to sessions.

1. In the AM admin UI, go to **Realms** > ***Realm Name*** > **Authentication** > **Settings** > **Trees**.
2. From the **Authentication session state management scheme** drop-down list, select `JWT`.
3. In the **Max duration (minutes)** field, enter the maximum life of the authentication session in minutes.
4. Save your changes.
5. Go to **Configure** > **Authentication** > **Core** > **Security**.
6. 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.

7. Save your changes.
8. Protect your client-side authentication sessions.

   See Client-side session security.

Configure client-side sessions

1. In the AM admin UI, go to **Realms** > ***Realm Name*** > **Authentication** > **Settings** > **General**.
2. Select the **Use Client-Side Sessions** check box.
3. Save your changes.
4. Protect your client-side sessions. See Client-side session security.
5. Verify that AM creates a client-side session when non-administrative users authenticate to the realm.

   Perform the following steps:

   - Authenticate to the AM admin UI 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.
   - Go to **Realms** > ***Realm Name*** > **Sessions**.
   - Verify that a session is present for the `amAdmin` user.
   - 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.
   - 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 Microsoft Edge, start InPrivate browsing.
     - In Firefox, open a new private window.
     - In Safari, open a new private window.
   - Authenticate to AM as a non-administrative user in the realm for which you enabled client-side sessions. Be sure *not* to authenticate as the `amAdmin` user this time.
   - 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-side session cookie's value should be considerably larger than the size of the cookie used by the server-side session for the `amAdmin` user. If the cookie is not larger, you have not enabled client-side sessions correctly.

- Return to the original browser window in which the AM admin UI appears.
- Refresh the window containing the Sessions page.
- 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-side sessions enabled.

Configure in-memory authentication sessions

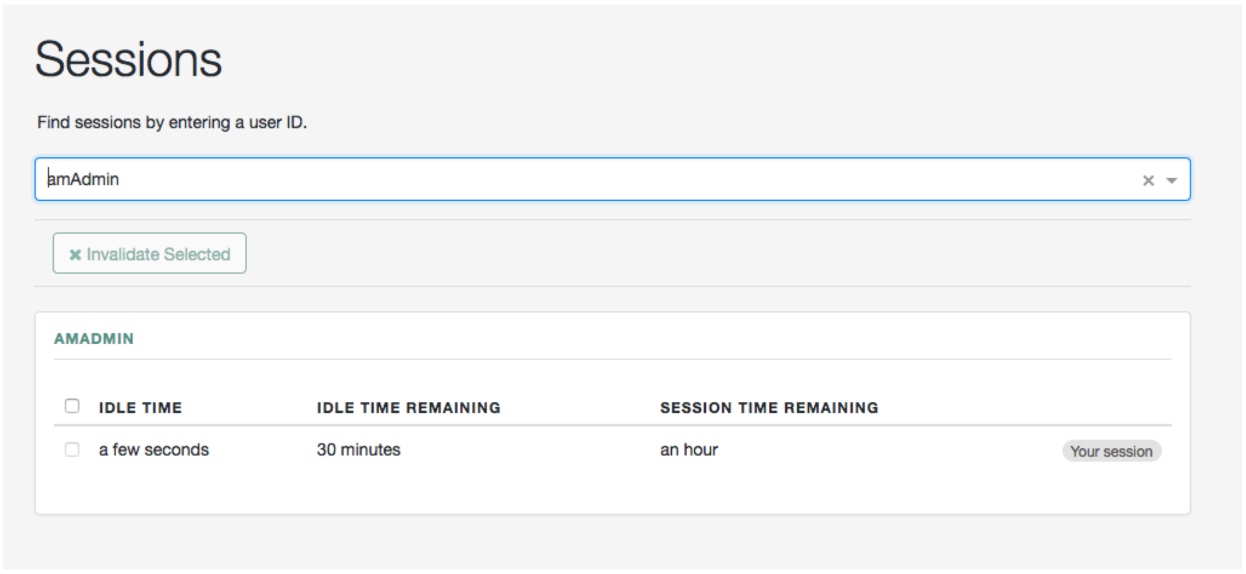
Authentication chains always store authentication sessions in AM’s memory. Perform the steps in the following procedure only for realms that configure authentication trees:

1. Ensure you have configured AM for sticky load balancing.

   For more information, see Load balancing.
2. In the AM admin UI, go to **Realms** > ***Realm Name*** > **Authentication** > **Settings** > **Trees**.
3. From the **Authentication session state management scheme** drop-down list, select `In-Memory`.
4. In the **Max duration (minutes)** field, enter the maximum life of the authentication session in minutes.
5. Save your changes.
6. Go 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.

Manage sessions in the UI

The AM admin UI lets the administrator view and manage active server-side user sessions by realm by going to **Realms** > ***Realm Name*** > **Sessions**.



Sessions

Find sessions by entering a user ID.

amAdmin

AMADMIN

	IDLE TIME	IDLE TIME REMAINING	SESSION TIME REMAINING	
☐	a few seconds	30 minutes	an hour	Your session

Figure 1. 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 server-side 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 server-side 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.

Manage sessions over REST

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

Get information about sessions over REST

To get information about a session, send an HTTP POST request 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 server-side session and also reset the idle timeout, use the `getSessionInfoAndResetIdleTime` endpoint, as follows. You cannot reset the idle timeout of client-side sessions.

```
$ 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 for session queries` setting, under **Realms** > ***Realm Name*** > **Services** > **Session Property Whitelist Service**.

Validate sessions over 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:

```
{
  "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.

Refresh server-side sessions over REST

To reset the idle time of a server-side 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 "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 server-side session, and returns timeout details of the session.

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

Invalidate sessions over 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 "Content-type: application/json" \
--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:

```
{
  "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 "Content-type: 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=logout"
{
    "result": "Successfully logged out"
}
```

Get and set 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 allowlist service configuration. For information on allowlisting session properties, see Session Property Whitelist service.

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 server-side session at any time during the session’s lifetime.
- You can only set and update properties on a client-side 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-side 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:

```
$ curl \
--request POST \
--header "Content-type: 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=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:

```
$ 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 `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"
}
```

For a list of the default session properties, see Session properties.

Session upgrade

Sessions can be upgraded to provide access to sensitive resources.

Consider a website for a University. Some resources, 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. This portal is protected with a policy that requires users to authenticate. 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*. Session upgrade 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=/alpha&ForceAuth=true#login`

  In this case, the user is asked to reauthenticate to the default authentication service in the `alpha` realm.

  When a new session is created, the old session should no longer be valid. For *client-side sessions*, invalidating the old session depends on the value of the Enable Session Denylisting configuration option. If this option is `false` (default), then both the old and new sessions are considered valid after the session upgrade. If this option is `true`, the old session is no longer valid

- 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 an *advice* that the user must perform one of the following actions:

  - Authenticate at an authentication level greater than the current level
  - Authenticate to a specific service
  - Authenticate to a specific module

  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 an authorization decision.
  3. AM returns an authorization decision that denies access to the resource, and returns an *advice* indicating that the user must 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**. AM performs one of the following actions depending on the type of session configured for the realm:
    - If the realm is configured for server-side sessions, the resulting action depends on the mechanism used to perform session upgrade:
        - When using the `ForceAuth` parameter:
            - *(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-side 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 fails 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.

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.

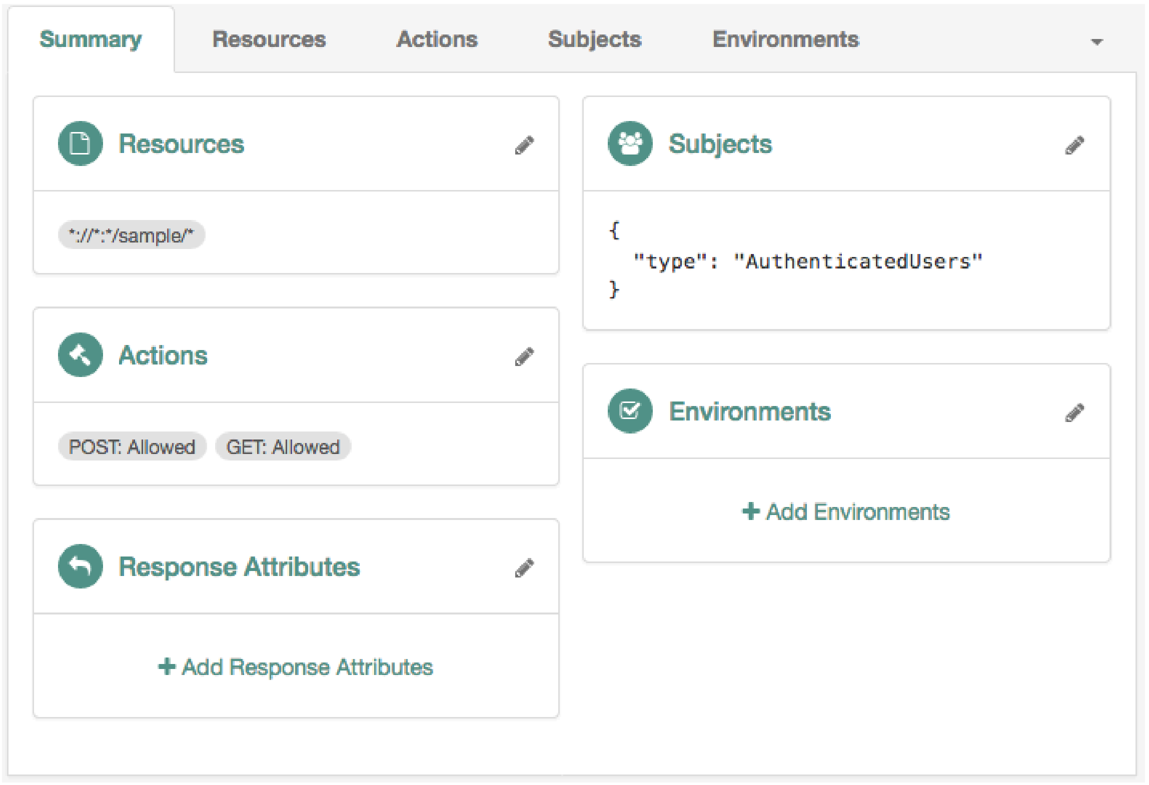
  AM web and Java agents handle session upgrade without additional configuration because the agents are built to handle AM's advices. If you build your own PEPs, however, you must take advices and session upgrade into consideration.

  ▼ *Resources*
    - Web Agents documentation.
    - Java Agents documentation.
    - Request policy decisions over REST (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:



://:*/sample/*

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

POST: Allowed GET: Allowed

Figure 2. Authorization Policy Example

Configure the environment for session upgrade

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

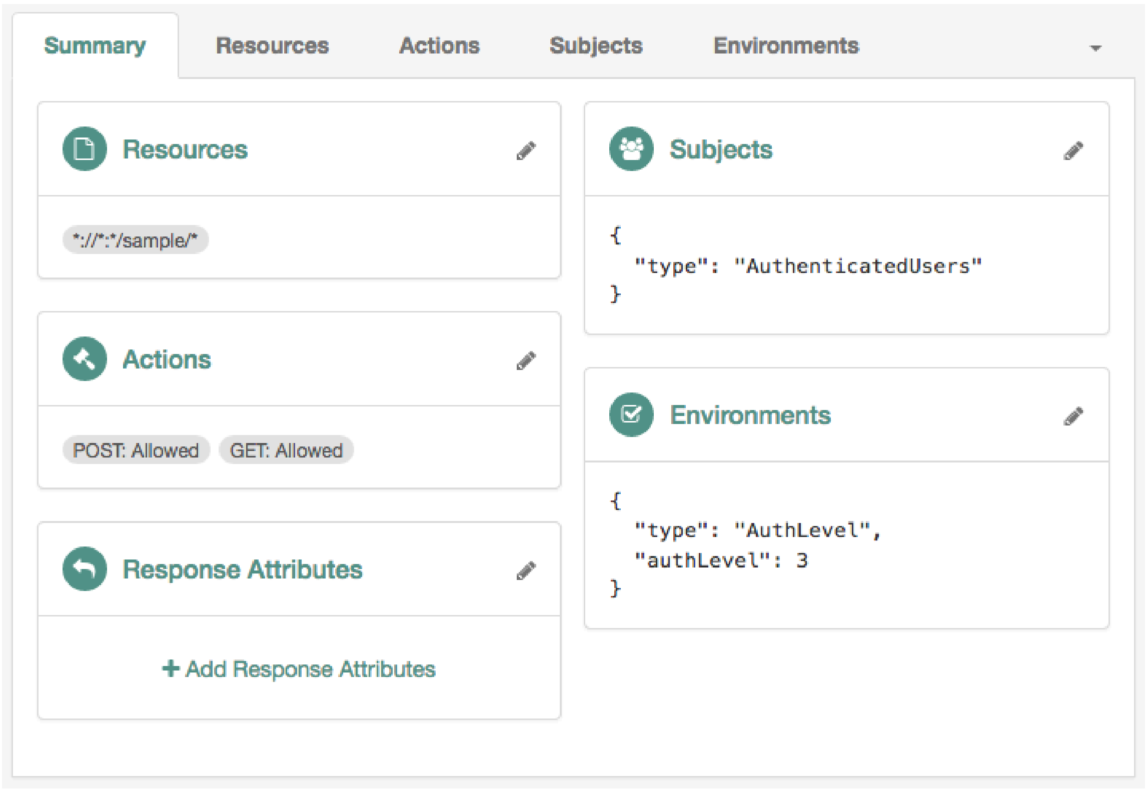
   Authentication trees and chains do not require additional configuration to perform session upgrade. However, because session upgrade is a mechanism that can be used to grant users access to sensitive information, you should consider configuring a strong authentication method such as multi-factor authentication. Also, 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, consider implementing transactional authorization instead.

   For information about configuring authentication trees and chains, refer to Authentication and SSO.

2. Configure at least one of the following environment conditions in the authentication policy that 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 modules that provide a greater or equal authentication level to the one specified in the condition. The user selects their service of choice if multiple services are able to meet the criteria of the condition. For example, the following policy requires a module that provides authentication level `3` or greater:



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

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

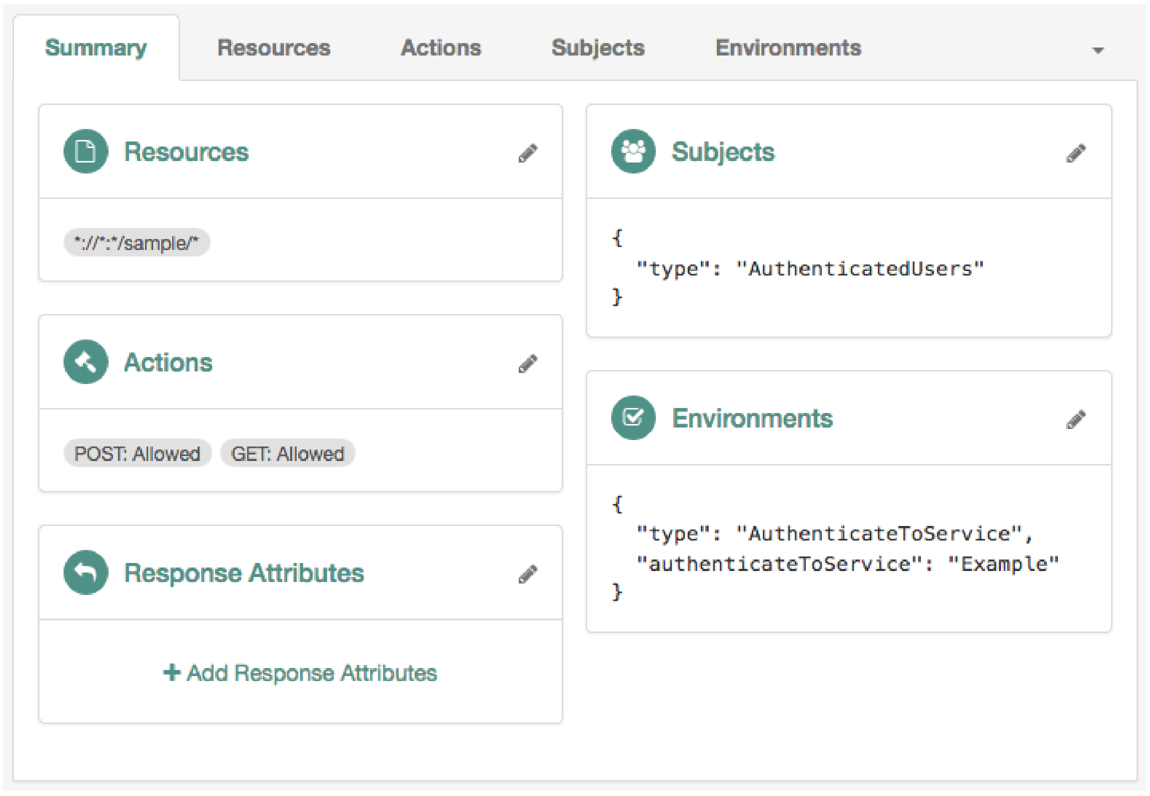
Figure 3. Session upgrade by authentication level

TIP

For more information about configuring the authentication level by authentication module, refer to Authentication levels for chains.

▼ *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:



://:*/sample/*

POST: Allowed GET: Allowed

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

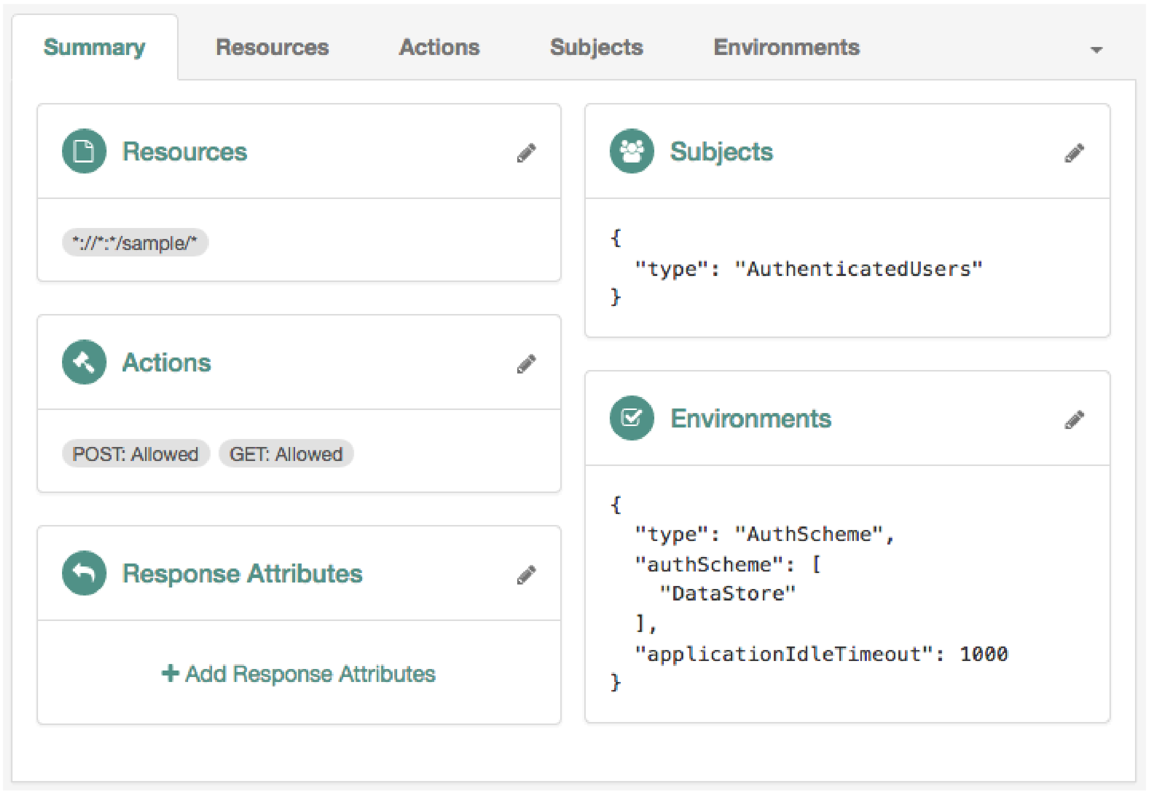
```
{
  "type": "AuthenticateToService",
  "authenticateToService": "Example"
}
```

Figure 4. 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:



://:*/sample/*

POST: Allowed GET: Allowed

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

```
{
  "type": "AuthScheme",
  "authScheme": [
    "DataStore"
  ],
  "applicationIdleTimeout": 1000
}
```

Figure 5. Session upgrade by module instance

NOTE

The examples show simple policy conditions. For more information about configuring policies and environment conditions, refer to Policies.

3. Test session upgrade:
   - To test session upgrade with a browser, refer to Perform session upgrade with a browser.
   - To test session upgrade using REST, refer to Perform session upgrade over REST.

Perform session upgrade with 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 Configure the environment for session upgrade.
2. In a browser, go 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.

Perform session upgrade over REST

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

1. Ensure you have performed the tasks in Session upgrade prerequisites and 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/realm
s/alpha/authenticate'
{
    "tokenId":"AQIC5wM2…",
    "successUrl":"/openam/console",
    "realm":"/alpha"
}
```

TIP

You can also assign privileges to a user to evaluate policies.

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/realm
s/alpha/authenticate'
{
    "tokenId":"AQIC5wM…TU3OQ*",
    "successUrl":"/openam/console",
    "realm":"/alpha"
}
```

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/realms/root/realm
s/alpha/policies?_action=evaluate"
[
   {
      "resource":"http://www.example.com:9090/sample",
      "actions":{

      },
      "attributes":{

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

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

For more information about requesting policy decisions, refer to Request policy decisions over REST.

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

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, refer to Policy decision advice.

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/realms/alpha/authenticate?
authIndexType=composite_advice&authIndexValue=%3CAdvices%3E%3CAttributeValuePair%3E…'
{

"authId":"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhdXRoSW5kZ…",
   "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, refer to Authenticate over REST.

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.eyJhdXRoSW5
kZ…",
                "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/r
ealms/alpha/authenticate?
authIndexType=composite_advice&authIndexValue=%3CAdvices%3
E%3CAttributeValuePair%3E…'
{
   "tokenId":"wpU01SaTq4X2x…NDVFMAAlMxAAA.*",
   "successUrl":"/openam/console",
   "realm":"/alpha"
}
```

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/realms/root/realm
s/alpha/policies/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.

Was this helpful?