FORGEROCK

Security Token Service (STS) Guide

/ ForgeRock Access Management 7.0.2

Latest update: 7.0.2

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

Abstract

Guide to working with the Security Token Service in ForgeRock® Access Management (AM). ForgeRock Access Management provides intelligent authentication, authorization, federation, and single sign-on functionality.

Table of Contents

Overview

This guide covers concepts, configuration, and usage procedures for working with the Security Token Service provided by ForgeRock Access Management.

This guide is written for anyone using the Security Token Service in Access Management to manage token exchange.

Quick Start

About the Secure Token Service Learn about the Secure Token Service (STS) and how it can transform, validate, issue, and cancel tokens.	Configure the Secure Token Service Add one or more STS instances to your AM deployment.	Query, Validate, and Cancel Tokens Manage tokens stored in the CTS token store

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 the Security Token Service

The Security Token Service (STS) lets AM bridge identities across existing federated environments to establish cross-domain trust relationships using token transformations. For example, you can federate two different environments by transforming OpenID Connect ID tokens into a SAML v2.0 assertions.

The WS-Trust specification introduces the concept of a centralized runtime component called the Security Token Service (STS), which issues, cancels, and validates security tokens in SOAP-based networks. A WS-Trust model involves communication between the components: a requestor, web service, and STS:

- The *requestor* is a web client or programmatic agent that wants to use a service offered by the web service.
- The *web service* allows authenticated and authorized clients to access resources or applications.
- The *identity provider* stores claims about subjects and works with the STS to issue security tokens.
- The *STS* acts as a trusted third-party web service that asserts the identity of a requestor across different security domains through the exchange of security tokens and brokers a trust relationship between the requestor and the web service provider. The STS issues tokens based on its configurations, which model the identity of a given identity provider, and issues tokens to a specific relying party.
- A *security token* is a SOAP STS data structure representing a set of claims that assert the identity of a subject. A single claim is identity information, such as a subject's name, age, gender, and email address.
- A *security policy*, defined in WS-SecurityPolicy, specifies the required elements, tokens, security bindings, supporting tokens, and protocol assertions, which are requirements for a web service to grant a subject access to its service. The security policy is defined in a *WSDL* document, which is an XML file that states what needs to be protected, what tokens are allowed for access, and transmission requirements for SOAP STS.

Web services and requestors (that is, consumers or clients) are typically deployed across different security domains and topologies. Each domain may require a specific security token type to assert authenticated identities. STS provides a means to exchange tokens across these different domains without re-authenticating or re-establishing trust relationships while allowing the requestor access to a web service's protected resources.

Based on this standard, AM provides two Security Token Services:

- SOAP STS, which is a fully WS-Trust 1.4-compliant Security Token Service.

  The AM SOAP STS is built upon the Apache CXF STS, an open-source implementation of JAX-WS and JAX-RS, as well as Apache WSS4j, an open-source Java implementation of the WS-Security specification.

  Caution

  The SOAP STS service is deprecated, and will be removed in a future release. Installing instances of this service is not supported.

- REST STS, which is a REST-based Security Token Services that *does not* conform to the WS-Trust specification, but that provides a simpler deployment alternative than SOAP STS for token transformations.

STS REST and SOAP Differences

Since the SOAP STS implementation is based on the WS-Trust specification and the REST STS implementation is not, there are differences between the features they support. They are summarized in the table below:

Differences Between the STS Implementations

Feature	Description	REST STS	SOAP STS
REST Endpoints	REST endpoints exposed upon instance creation.	✓	✗
SOAP Endpoints	AM `.war` and the SOAP STS `.war` files must be deployed in separate web containers to expose the SOAP endpoints.	✗	✓
Token Transformations	AM STS issues OpenID Connect V1.0 (OIDC) and SAML V2.0 tokens (bearer, holder-of-key, sender vouches). • Username token → OIDC • OIDC → OIDC • X.509 token → OIDC • AM Session token → OIDC • Username token → SAML v2.0 • X.509 token → SAML v2.0 • (REST STS only) OIDC token → SAML v2.0 • AM Session token → SAML v2.0	✓	✓
Publish Service	You can configure REST or SOAP STS instances using the AM console or programmatically. AM provides a REST STS publish service that allows you to publish these instances using a POST to the endpoints. Note	✓	✓

Feature	Description	REST STS	SOAP STS
	that a published instance can have only a single encryption key. Therefore, you need one published instance per service provider that the web service invoking the STS intends to call.		
Custom SAML Assertion Plugins	AM supports customizable SAML assertion statements. You can create custom plug-ins for `Conditions`, `Subject`, `AuthenticationStatements`, `AttributeStatements`, and `AuthorizationDecisionStatements` statements.	✓	✓
Custom Token Validators and Providers	The AM REST STS provides the ability to customize tokens that are not supported by default by the STS. For example, you can configure STS to transform a token of type CUSTOM to a SAML V2.0 token.	✓	✗
Client SDK	AM provides a SOAP STS client SDK module to allow developers to use Apache CXF-STS classes.	✗	✓
`ActAs` and `OnBehalfOf` Elements	AM SOAP STS supports delegated and proxied token relationships, as defined by the `ActAs` and `OnBehalfOf` elements in WS-Trust, which is available for Username and AM session tokens.	✗	✓
Security Binding Assertions	AM SOAP STS supports the WS-SecurityPolicy binding assertions that protect communication to and from the STS: transport, asymmetric, symmetric.	✗	✓
Custom WSDL	The AM SOAP STS comes with a pre-configured WSDL file. You can customize the policy bindings governing the input or output messages to or from the STS.	✗	✓
Logging Service	The AM STS allows SOAP-STS log entries to be configured via `java.util.logging`, which allows logging to be configured via the `logging.properties` file in the Tomcat `conf` directory.	✗	✓

For more information about both implementations, see:

- "About the SOAP STS"
- "About the REST STS"

About the SOAP STS

AM lets AM administrators publish WS-Trust 1.4-compliant STS instances, each with a distinct security policy configuration, and each issuing OpenID Connect (OIDC) v1.0 Tokens or SAML v2.0 (bearer, holder of key, and sender vouches) assertions.

The SOAP STS is deployed remotely from AM in a Tomcat or Jetty container. Deploying both the AM `.war` and the SOAP STS `.war` in the same container is not supported. The remotely-deployed SOAP STS

`.war` file authenticates to AM with SOAP STS agent credentials, and pulls the configuration state for all SOAP instances published in its realm, exposing WS-Trust-compliant SOAP web services based on this configuration state.

AM is the authentication authority for the STS instances and its configured data stores, which store the attributes that are included in OIDC tokens and generated SAML v2.0 assertions.

You can publish any number of SOAP STS instances programmatically, or by using the AM console. Each instance is published with a specific WS-SecurityPolicy binding, which specifies:

- Type of supporting token that asserts the caller's identity.
- Manner in which the supporting token is protected (symmetric, asymmetric, or transport binding).

Each published SOAP STS instance is protected by a security policy binding, which specifies what type of token must be presented to assert the caller's identity (also known as the *supporting token*), and how this supporting token is protected. There are three protection schemes: transport, symmetric, and asymmetric:

- **Transport Binding Assertion**. Transport binding is used when the message is protected at the transport level, such as HTTPS, and thus requires no explicit enforcement at the security policy binding enforcement level. The SOAP keystore configuration allows a SOAP STS instance to be published referencing the keystore state necessary to enforce the symmetric and asymmetric bindings.
- **Symmetric Binding Assertion**. Symmetric binding is used when only one party needs to generate security tokens. In a symmetric binding, the client generates symmetric key state used to sign and encrypt messages, and encrypts this symmetric key state with the STS's public key, and includes the encrypted symmetric key in the request. Thus, the SOAP keystore configuration of a published STS instance, which is protected by the symmetric binding, must reference a keystore with the STS's `PrivateKeyEntry`, so that it may decrypt the symmetric key generated by the client.
- **Asymmetric Binding Assertion**. Asymmetric binding is used when both the client and the service both have security tokens. In an asymmetric binding, client requests are signed with the client's secret key, and encrypted with the STS's public key. STS responses are signed with the STS's private key and encrypted with the client's public key. The client's X.509 certificate is included in the request, so that the STS can validate the client's signature and encrypt responses to the client without requiring the presence of the client's X.509 certificate in the STS's keystore. However, the SOAP keystore configuration of a published STS instance protected by an asymmetric binding must reference a keystore with the STS's *PrivateKeyEntry*, which allows the STS to both: 1) sign messages from STS to client, and 2) decrypt messages from the client.

  Note

  The Decryption Key Alias in a SOAP STS instance's configuration corresponds to the `PrivateKeyEntry`.

The following bindings are available:

- `UsernameToken` over the Transport, Symmetric, and Asymmetric binding
- AM Session Token over the Transport and Unprotected binding
- X.509 certificates examples seen in WS-SecurityPolicy Examples Version 1.0

A SAML v2.0 assertion, defined in SAML V2.0, contains a `Subject` element that identifies the principal, which is the subject of the statements in the assertion. The `Subject` element contains an identifier and zero or more `SubjectConfirmation` elements, which allows a relying party to verify the subject of the assertion with the entity with whom the relying party is communicating.

The `SubjectConfirmation` element contains a required `Method` attribute that specifies the URI identifying the protocol used to confirm the subject. The AM STS supports the following subject confirmation methods:

- **Holder of Key**. The holder of key subject confirmation method involves proving a relationship between the subject and claims. This is achieved by signing part of the SOAP message with a proof key sent in the SAML assertion. The additional proof key guards against any attempted man-in-the-middle attack by ensuring that the SAML assertion comes from the subject claiming to the be requestor.

  `URI: urn:oasis:names:tc:SAML:2.0:cm:holder-of-key`

- **Sender Vouches**. The sender vouches subject confirmation method is used in cases where you have a proxy gateway that propagates the client identity via the SOAP messages on behalf of the client. The proxy gateway must protect the SOAP message containing the SAML assertion, so that the web service can verify that it has not been tampered with.

  `URI: urn:oasis:names:tc:SAML:2.0:cm:sender-vouches`

- **Bearer**. The bearer subject confirmation method assumes that a trust relationship exists between the subject and the claims, and thus no keys are required when using a bearer token. No additional steps are required to prove or establish a relationship.

  Since browser-based clients use bearer tokens and no keys are required, you must protect the SOAP message using a transport-level mechanism, such as SSL, as this is the only means to protect against man-in-the-middle attacks.

  `URI: urn:oasis:names:tc:SAML:2.0:cm:bearer`

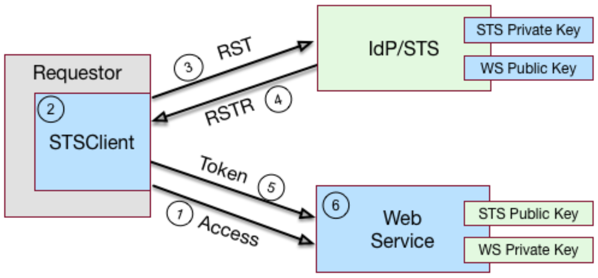
If you are interested in the SOAP STS, you should be familiar with the SOAP STS specifications:

- SAML V2.0
- SAML V2.0 Errata Composite
- Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0
- OpenID Connect Core 1.0 incorporating errata set 1
- WS-Federation

- WS-Trust
- WS-SecurityPolicy
- WS-SecurityPolicy Examples Version 1.0
- WS-Metadata Exchange
- UserNameToken
- X509Token
- SAMLTokenProfile

"Basic SOAP STS Model" illustrates a basic SOAP STS topology between a requestor, web service, and STS. The STS instance is set up with the identity provider, which has an existing trust relationship with the web service.

Basic SOAP STS Model



A basic SOAP STS process flow is as follows:

1. A requestor first accesses a protected resource for a web service. The requestor gets the web service's WSDL file, which specifies the policy requirements to access its services.
2. The requestor creates and configures an `STSClient` object whose main task is to contact the STS.
3. The `STSClient` contacts the STS to obtain its WSDL file. Each published STS instance exposes an API that is defined in its WSDL file. The WSDL file specifies the security policy bindings, which specify the type of token they must present to the API, and how this token is protected during transit.
4. The `STSClient` generates and sends a Request for Security Token (RST) to the STS. The RST specifies the what type of token is desired. The requestor's `usernameToken` is embedded in the SOAP envelope that contains the RST and is used for authentication.

   The SOAP STS client SDK provides the classes, templates, and documentation to allow developers to set the state necessary to allow the Apache CXF runtime to generate the SOAP envelopt containing the RST, which satisfies the security policy bindings of the targeted STS.

5. The STS validates the requestor's `usernameToken`, creates an interim AM session, and optionally, creates a CTS token for the session. Upon successful authentication, the STS constructs a Request for Security Token Response (RSTR), signs the SAML v2.0 token, and embeds the token within the RSTR. If STS is configured to invalidate the interim token, it does so. The STS sends a Request for Security Token Response (RSTR) to the `STSClient`.

6. The `STSClient` extracts the security token and sends it in the request's message header. The `STSClient` sends the message to the web service.

7. The web service extracts the SAML token and validates the signature to ensure that it came from the STS. The web service allows the user whose ID is specified in the SAML token to access its protected resource.

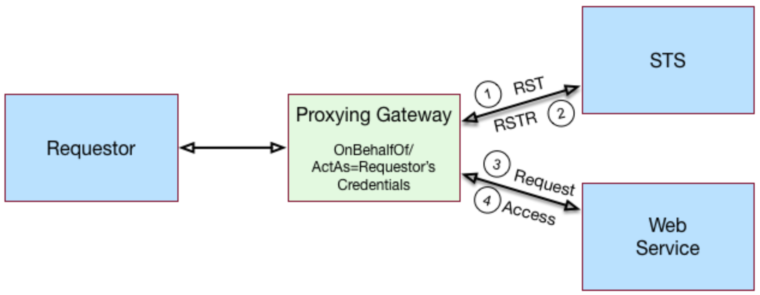
8. If a CTS token was created for the session, the web service can call the SOAP STS to invalidate the token and the corresponding AM session upon request.

Supporting Delegated Relationships in SOAP STS

SOAP STS supports the ability to issue SAML assertions with the sender vouches subject confirmation method. Sender vouches are used in proxy deployments, such as a proxying gateway, where the gateway requests a SAML assertion with a sender vouches confirmation from the STS.

In this case, the requestor's credentials are set in the `OnBehalfOf` and `ActAs` elements in the request security token (RST) request included in the `Issue` invocation. The gateway calls the STS, and the gateway’s credentials satisfy the security policy bindings protecting the STS. The presence of either the `OnBehalfOf` and `ActAs` elements together with a token type of SAML v2.0 and a key type of `PublicKey` triggers the issuance of a sender vouches SAML v2.0 assertion.

STS Sender Vouches



The STS runs token validators that validate the authenticity of the `ActAs` or `OnBehalfOf` token.

The SOAP STS configuration indicates whether token delegation relationships are supported in the STS in the `ActAs` and `OnBehalfOf` elements. If token delegation is supported, the configuration also indicates the token types that token validators use to validate the `ActAs` and `OnBehalfOf` token elements.

In the Request for Security Token (RST) invocation, `Username` and AM tokens are supported for the `OnBehalfOf` element. In addition, you can specify that the SOAP STS instance be deployed with a user-specified implementation of the token delegation handler interface, `org.apache.cxf.sts.token.delegation.TokenDelegationHandler`.

A default token delegation handler is used if no custom token delegation handler is configured. The default token delegation handler rejects the delegation relationship if the token principal set to null in the token delegation parameters (that is, TokenDelegationParameters), as this is the case when no token validators have validated the `ActAs` and `OnBehalfOf` token. Thus, if you want the STS instance to support the `ActAs` and `OnBehalfOf` elements, you must specify one of the two following configuration properties:

- The Delegation Relationships Supported property.
- One or more Delegated Token types. For example, AM or `Username` for which token validators are deployed to validate the `ActAs` or `OnBehalfOf` tokens and/or a custom token delegation handler.

Note

If you configure the `Username` token type as a delegated token type, AM uses the configuration in the Authentication Target Mappings property to authenticate `Username` tokens. AM SSO tokens require no special configuration in the Authentication Target Mappings property.

Example Proxy Gateway STS Deployment

Suppose you want to deploy the SOAP STS to receive requests from a proxy gateway and issue SAML v2.0 assertions with sender vouches subject confirmation method. The gateway sends the SAML v2.0 assertion that asserts the identity of the gateway client and vouches for its identity.

Suppose the SOAP STS deployment has a security policy binding requiring the presentation of an X.509 certificate. This security policy binding can be satisfied by presenting the gateway's X.509 certificate. However, the SOAP STS-issued SAML v2.0 assertion should assert the identity of the gateway client that presents its identity to the gateway as either a <username, password> combination or as an AM session.

In this case, the published SOAP STS would specify an X.509-based security policy, the delegation relationships to be supported, and whether both AM and `Username` token types should be supported. No custom token delegation handler need be specified.

Furthermore, the SOAP STS instance must be published with Authentication Target Mappings that specify how the `Username` token should be presented to AM's RESTful authentication context. The gateway code would then create a request for security token (RST) invocation using the classes in the `openam-sts/openam-soap-sts/openam-soap-sts-client` module, and include the gateway client's <username, password> or AM session state as the `OnBehalfOf` element. This setting allows the gateway to consume the SOAP STS to issue SAML v2.0 assertions with the sender vouches subject confirmation method, which asserts the identity of the gateway client corresponding to the presented <username, password> or AM session state.

If, at a later date, you want to exclude or blacklist some users from attaining SAML v2.0 assertions, regardless of their possession of valid <username, password> or AM session state, you can update the SOAP STS with the class name of a token delegation handler implementation, which would implement this blacklist functionality. The SOAP STS `.war` file would have to be re-created with this file in the classpath. The token delegation handler could reject the invocation for users or principals on the blacklist.

About the REST STS

AM's REST STS service provides an easier deployment alternative to SOAP STS to issue OpenID Connect 1.0 or SAML v2.0 tokens for a single service provider. Each REST STS instance is configured with the following elements:

- **Issuer**. The issuer corresponds to the IDP `EntityID`.
- **SP EntityID**. The SP `EntityID` is used in the `AudienceRestriction` element of the `Conditions` statement of the issued assertion.
- **SP Assertion Consumer Service URL**. The SP assertion consumer service URL is used as the `Recipient` attribute of the `subjectConfirmation` element in the `Subject` statement, which is required for bearer assertions according to the Web SSO profile.

For signing and encryption support, each REST STS instance has a configuration state, which specifies the keystore location containing the signing and encryption keys:

- If *assertion signature* is configured, the keystore path and password must be specified, as well as the alias and password corresponding to the `PrivateKey` used to sign the assertion.
- If *assertion encryption* is configured, the keystore path and password must be specified, as well as the alias corresponding the SP’s X509Certificate encapsulating the `PublicKey` used to encrypt the symmetric key used to encrypt the generated assertion.

  Note that the keystore location can be specified via an absolute path on the local filesystem, or a path relative to the AM classpath. Either the entire assertion can be encrypted, or the `NameID` and/or `AttributeStatement` Attributes.

All statements constituting a SAML v2.0 assertion can be customized. For each REST STS instance, you can provide custom plug-ins for `Conditions`, `Subject`, `AuthenticationStatements`, `AttributeStatements`, and `AuthorizationDecisionStatements`. If you specify the custom plug-ins in the configuration state of the published REST STS instance, the custom classes are consulted to provide the specific statements. See the interfaces in the `org.forgerock.openam.sts.tokengeneration.saml2.statements` package for details.

Each REST STS instance must specify the authentication context (that is, `AuthnContext`) to be included in the `AuthenticationStatements` of the generated assertion. This `AuthnContext` allows the generated SAML v2.0 assertion to specify the manner in which the assertion’s subject was authenticated. For a token transformation, this `AuthnContext` is a function of the input token type. By default, the following `AuthnContext` strings will be included in the SAML v2.0 assertion generated as part of the transformation of the following input token types:

- AM: `urn:oasis:names:tc:SAML:2.0:ac:classes:PreviousSession`
- Username Token and OpenID Connect Token: `urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport`
- X.509 Token: `urn:oasis:names:tc:SAML:2.0:ac:classes:X509`

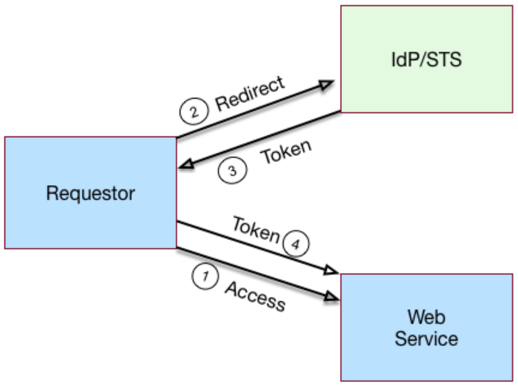
Note that you can override these default mappings by implementing the `org.forgerock.openam.sts.token.provider.AuthnContextMapper` interface, and specifying the name of this implementation in the configuration of the published REST STS instance.

If you are interested in the REST STS, you should be familiar with the following specifications before setting up your deployment:

- SAML V2.0
- SAML V2.0 Errata Composite
- Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0
- OpenID Connect Core 1.0 incorporating errata set 1

"Basic REST STS Model" illustrates a simple REST STS topology between a requestor, web service, and STS. The STS instance is set up with the identity provider, which has an existing trust relationship with the web service. The difference between the REST STS versus the SOAP STS is that REST STS does not strictly follow the WS-Trust specification for input token and output token formats. However, the REST STS provides a simpler means to deploy an STS instance, compared to that of the SOAP STS.

Basic REST STS Model



A simple REST STS process flow is as follows:

1. A requestor makes an access request to a web resource.
2. The web service redirects the requestor to the STS.
3. The requestor sends an HTTP(S) POST to the STS endpoint. The request includes credentials, token input type, and desired token output type. An example curl request is shown below:

   ```
   $ curl \
   --request POST \
   --header "Content-Type: application/json" \
   --data '{
       "input_token_state": {
           "token_type": "USERNAME",
           "username": "demo",
           "password": "Ch4ng31t"
       },
       "output_token_state": {
           "token_type": "SAML2",
           "subject_confirmation": "BEARER"
       }
   }' \
   https://openam.example.com:8443/openam/rest-sts/username-transformer?_action=translate
   ```

   Or, you can run a command for an OIDC token:

   ```
   $ curl \
   --request POST \
   --header "Content-Type: application/json" \
   --data '{
       "input_token_state": {
           "token_type": "USERNAME",
           "username": "demo",
           "password": "Ch4ng31t"
       },
       "output_token_state": {
           "token_type": "OPENIDCONNECT",
           "nonce":"12345678",
           "allow_access":true
       }
   }' \
   https://openam.example.com:8443/openam/rest-sts/username-transformer?_action=translate
   ```

4. The STS validates the signature, decodes the payload, and verifies that the requestor issued the transaction. The STS validates the requestor's credentials, creates an interim AM session, and optionally, creates a CTS token for the session. The STS then issues a token to the requestor. If STS is configured to invalidate the interim AM session, it does so. The requestor gets redirected to the web service.
5. The requestor presents the token to the web service. The web service validates the signature, decodes the payload, and verifies that the requestor issued the request. It then extracts and validates the token and processes the request.
6. If a CTS token was created for the session, the web service can call the REST STS to invalidate the token and the corresponding AM session upon request.

Validating Input Tokens

STS token transformations validate input tokens before generating output tokens. STS uses AM authentication modules and chains to perform token validation. When deploying STS, you must configure AM authentication so that it can validate input tokens.

This section describes authentication configuration requirements for username, X.509, and OpenID Connect tokens. No special authentication configuration is required when using AM session tokens as the input tokens in token transformations.

Because REST STS instances are not part of a secure framework like WS-Trust 1.4, this section also mentions security issues you should consider when sending tokens across a network to a REST STS instance.

In addition to configuring AM authentication to support input token validation, you must identify the authentication module or chain to be used to validate each input token type. To do so, configure the Authentication Target Mappings property in the STS instance configuration. For more information about this property, see "*Reference*".

Validating Username Tokens

Username tokens passed to a REST STS instance contain the username/password combination in cleartext. Tokens can be validated using any module type that supports username/password authentication, including Data Store, LDAP, and so forth.

With usernames and passwords in cleartext, be sure to configure your deployment with an appropriate level of security. Deploy REST STS instances that support input username token transformations on TLS.

Validating X.509 Certificate Tokens

REST STS instances can obtain X.509 certificates used as input tokens in two ways:

- From the header key defined in the REST STS instance's Client Certificate Header Key property. In this case, STS also confirms that the request came from a host specified in the Trusted Remote Hosts property.
- From the `javax.servlet.request.X509Certificate` attribute in the ServletRequest. The REST STS instance obtains the X.509 certificate from the ServletRequest if no header key is configured in the Client Certificate Header Key property.

The AM Certificate module authenticates the X.509 certificate input token. The module optionally performs certificate revocation list (CRL) or Online Certificate Status Protocol (OCSP) checking, and can optionally check to see that the specified certificate is in a LDAP datastore.

If certificates are passed to REST STS using HTTP headers, you must configure the Trusted Remote Hosts and Http Header Name for Client Certificate properties in the Certificate module to match your REST STS instance's configuration.

Validating OpenID Connect Tokens

To validate OpenID Connect input tokens, a REST STS instance must reference an OpenID Connect id_token bearer authentication module in the Authentication Target Mappings property.

Configure the authentication module as follows:

- Specify a header in the Name of header referencing the ID Token property. The REST STS instance's Target Authentication Mapping property must reference the same header.
- Specify the issuer name in the Name of OpenID Connect ID Token Issuer property, and configure the token issuer's discovery URL, JWK URL or, client secret in the OpenID Connect validation configuration value property.
- If incoming OpenID Connect tokens contain `azp` claims, specify valid claims in the List of accepted authorized parties property.
- If incoming OpenID Connect tokens contains `aud` claims, specify the valid claim in the Audience property.
- Configure attribute mappings so that JWK claims map to attributes in the AM user store.

For more information about OpenID Connect id_token bearer authentication module properties, see "OpenID Connect id_token bearer Module" in the *Authentication and Single Sign-On Guide*.

Note

SOAP STS instances do not accept OpenID Connect tokens as input tokens in token transformations.

Chapter 2
Configuring STS Instances

You configure STS *instances* to perform one or more token transformations. Each instance provides configuration details about how SAML v2.0 and/or OpenID Connect output tokens are encrypted or signed. Deployments that support multiple SAML v2.0 and/or OpenID Connect service providers require multiple STS instances.

When you publish an STS instance, you create an STS instance with a given configuration. You can publish instances either using the AM console or the REST API.

When you publish a REST STS instance, AM exposes a REST endpoint for accessing the instance, and the instance is immediately available for use to callers.

Caution

The SOAP STS service is deprecated, and will be removed in a future release. Installing instances of this service is not supported.

Configuring the REST STS

To implement the REST STS using the AM console, add one or more REST STS instances to your AM deployment.

To configure a REST STS instance using the AM console, navigate to Realms > *Realm Name* > STS > REST STS Instances, and then click Add.

See "REST STS Configuration Properties" for detailed information about STS configuration properties.

Tip

You can also publish REST STS instances programmatically. AM provides a Publish Service, which is a collection of endpoints you can use to publish instances instead of accessing the AM console.

For more information, see "Publishing REST STS Instances".

Chapter 3
Consuming SOAP STS Instances

You consume a SOAP STS instance by sending it SOAP messages to the instance's endpoint, or by calling it using the AM SOAP STS client SDK.

SOAP STS Instance URL

SOAP STS instances' URLs are comprised of the following parts:

- The SOAP STS deployment context
- The string `sts`
- The realm in which the REST STS instance is configured
- The deployment URL element, which is one of the configuration properties of an STS instance

The SOAP STS deployment context comprises the base URL of the web container to which the SOAP STS `.war` file is deployed, and the deployment web application name.

For example, a SOAP STS instance configured in the realm `myRealm` with the deployment URL element `soap-username-transformer` and the a deployment web application name `openam-soap-sts` would expose a URL similar to `https://soap-sts-host.com:8443/openam-soap-sts/sts/myRealm/soap-username-transformer`.

The WSDL for the service would be available at `https://soap-sts-host.com:8443/openam-soap-sts/sts/myRealm/soap-username-transformer?wsdl`.

Consuming SOAP STS Instances Using SOAP Messages

Because an AM SOAP STS instance is a WS-Trust 1.4-compliant security token service, users can consume the instance by sending it standard WS-Trust 1.4 SOAP STS framework messages, such as `RequestSecurityToken` messages, passed as the payload to WSDL ports that are implemented by the security token services.

For more information about WS-Trust 1.4 security token services, see the WS-Trust 1.4 specification.

Consuming SOAP STS Instances Using the SOAP STS Client SDK

You can consume an AM SOAP STS instance by calling it using the AM SOAP STS client SDK.

The SOAP STS client SDK is based on classes in Apache CXF, an open source service framework. Apache CXF provides the `org.apache.cxf.ws.security.trust.STSClient` class, which encapsulates consumption of a SOAP STS service. However, using this class requires considerable expertise.

The SOAP STS client SDK makes it easier to consume AM SOAP STS instances than using Apache CXF for the following reasons:

- The `org.forgerock.openam.sts.soap.SoapSTSConsumer` class in the AM SOAP STS client SDK wraps the Apache CXF class `org.apache.cxf.ws.security.trust.STSClient`, providing a higher level of abstraction that makes consumption of SOAP STS instances easier to achieve.
- The `SoapSTSConsumer` class' `issueToken`, `validateToken`, and `cancelToken` methods provide the three fundamental operations exposed by SOAP STS instances. Supporting classes facilitate the creation of state necessary to invoke these methods.
- Classes in the SDK provide logic to allow AM session tokens to be presented in order to satisfy the security policy bindings that mandate AM sessions as supporting tokens. The STS client obtains secret password state—keystore entry passwords and aliases, username token credential information, and so forth—from a callback handler. The `SoapSTSConsumerCallbackHandler` class provides the means to create a callback handler initialized with state that will be encountered when consuming SOAP STS instances. The `SoapSTSConsumerCallbackHandler` instance can be passed to an STS client. The `TokenSpecification` class provides a way to create the varying token state necessary to obtain specific tokens and create any necessary supporting state.

You can use the classes in the SOAP STS client SDK as is, or you can tailor them to your needs. For more information about the SOAP STS client SDK classes, see the source code and the Javadoc.

The SOAP STS client SDK is not part of the AM client SDK. [1] To use the SOAP STS client SDK, you must compile the source code for the SOAP STS client SDK and create a `.jar` file.

To Build the SOAP STS Client SDK

1. Download the AM source code.
2. Change to the `openam-sts/openam-soap-sts` directory.
3. Run the `mvn install` command.
4. Locate the `openam-soap-sts-client-7.0.2.jar` file in the `openam-sts/openam-soap-sts/openam-soap-sts-client/target` directory.

[1] The SOAP STS client SDK has a dependency on Apache CXF classes, which are not present in the AM API.

Chapter 4
Consuming REST STS Instances

You consume a REST STS instance by sending REST API calls to the instance's endpoint.

REST STS Instance Endpoint

REST STS instances' endpoints are comprised of the following parts:

- The AM context
- The string `rest-sts`
- The realm in which the REST STS instance is configured
- The deployment URL element, which is one of the configuration properties of an STS instance

For example, a REST STS instance configured in the realm `myRealm` with the deployment URL element `username-transformer` exposes the endpoint `/rest-sts/myRealm/username-transformer`.

JSON Representation of Token Transformations

Token transformations are represented in JSON as follows:

```
{
  "input_token_state": {
    "token_type": "INPUT_TOKEN_TYPE"
    ... INPUT_TOKEN_TYPE_PROPERTIES ...
  },
  "output_token_state": {
    "token_type": "OUTPUT_TOKEN_TYPE"
    ... OUTPUT_TOKEN_TYPE_PROPERTIES ...
  }
}
```

REST STS supports the following token types and properties:

Input token types

- `USERNAME`

  Requires the `username` and `password` properties.

- `OPENAM`

  Requires the `session_id` property, with an SSO token as its value.

- `X509`

  No properties are required, because input X.509 tokens are presented either in HTTP headers or by using TLS. For more information about X.509 tokens, see the configuration details for the Authentication Target Mappings and Client Certificate Header Key properties in "REST STS Configuration Properties".

- `OPENIDCONNECT`

  Requires the `oidc_id_token` property, with the OpenID Connect token as its value.

Output token types

- `SAML2`

  Requires the `subject_confirmation` property, the value of which determines the `<saml:ConfirmationMethod>` element for the generated SAML v2.0 assertion. Valid values are `BEARER`, `SENDER_VOUCHES`, and `HOLDER_OF_KEY`.

  When generating an assertion with a holder-of-key subject confirmation method, the `proof_token_state` property is required. The value for this property is an object that contains the `base64EncodedCertificate` property.

- `OPENIDCONNECT`

  Requires the `nonce` and `allow_access` properties.

The following are examples of JSON payloads that define REST STS token transformations:

1. Transform a username token to a SAML v2.0 token with the bearer subject confirmation method:

```
{
  "input_token_state": {
    "token_type": "USERNAME",
    "username": "demo",
    "password": "Ch4ng31t"
  },
  "output_token_state": {
    "token_type": "SAML2",
    "subject_confirmation": "BEARER"
  }
}
```

2. Transform an X.509 token to a SAML v2.0 token with the sender vouches subject confirmation method:

```
{
  "input_token_state": {
    "token_type": "X509"
  },
  "output_token_state": {
    "token_type": "SAML2",
    "subject_confirmation": "SENDER_VOUCHES"
  }
}
```

3. Transform an OpenID Connect token to a SAML v2.0 token with the holder-of-key subject confirmation method:

```
{
  "input_token_state": {
    "token_type": "OPENIDCONNECT",
    "oidc_id_token": "eyAiYWxQ.euTNnNDExNTkyMjEyIH0.kuNlKwyvZJqaC8EYpDyPJMiEcII"
  },
  "output_token_state": {
    "token_type": "SAML2",
    "subject_confirmation": "HOLDER_OF_KEY",
    "proof_token_state": {
      "base64EncodedCertificate": "MIMbFAAOBjQAwgYkCgYEArSQ...c/U75GB2AtKhbGS5pimrW0Y0Q=="
     }
  }
}
```

4. Transform an AM SSO token to an OpenID Connect token:

```
{
  "input_token_state": {
    "token_type": "OPENAM",
    "session_id": "AQIC5wM2...TMQAA*"
  },
  "output_token_state": {
    "token_type": "OPENIDCONNECT",
    "nonce": "471564333",
    "allow_access": true
  }
}
```

For more examples of JSON payloads that you can send to REST STS instances, see the comments in the sample code in "Java Example".

Command-Line Example

You can use the **curl** command to quickly verify that a published REST STS instance operates as expected.

For example, if you publish a REST instance with a deployment URL element `username-transformer` that supports username to SAML v2.0 bearer assertion token transformation, you can perform an HTTP POST to the `/rest-sts/username-transformer` endpoint, setting the `_action` parameter to `translate` as follows:

```
$ curl \
--request POST \
--header "Content-Type: application/json" \
--data '{
    "input_token_state": {
        "token_type": "USERNAME",
        "username": "demo",
        "password": "Ch4ng31t"
    },
    "output_token_state": {
        "token_type": "SAML2",
        "subject_confirmation": "BEARER"
    }
}' \
https://openam.example.com:8443/openam/rest-sts/username-transformer?_action=translate
{
  "issued_token":
     "<saml:Assertion
       xmlns:saml=\"urn:oasis:names:tc:SAML:2.0:assertion\"
       Version=\"2.0\"
       ID=\"s2c51ebd0ad10aae44fb76e4b400164497c63b4ce6\"
       IssueInstant=\"2016-03-02T00:14:47Z\">\n
       <saml:Issuer>saml2-issuer</saml:Issuer>
       <saml:Subject>\n
        <saml:NameID
         Format=\"urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress\">demo
        </saml:NameID>
        <saml:SubjectConfirmation
         Method=\"urn:oasis:names:tc:SAML:2.0:cm:bearer\">\n
         <saml:SubjectConfirmationData
          NotOnOrAfter=\"2016-03-02T00:24:47Z\" >
         </saml:SubjectConfirmationData>
        </saml:SubjectConfirmation>\n
       </saml:Subject>
       <saml:Conditions
        NotBefore=\"2016-03-02T00:14:47Z\"
        NotOnOrAfter=\"2016-03-02T00:24:47Z\">\n
        <saml:AudienceRestriction>\n
         <saml:Audience>saml2-issuer-entity</saml:Audience>\n
        </saml:AudienceRestriction>\n</saml:Conditions>\n
        <saml:AuthnStatement
         AuthnInstant=\"2016-03-02T00:14:47Z\">
         <saml:AuthnContext>
          <saml:AuthnContextClassRef>
           urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport
          </saml:AuthnContextClassRef>
         </saml:AuthnContext>
        </saml:AuthnStatement>
       </saml:Assertion>\n"
}
```

The `iPlanetDirectoryPro` header is required and should contain the SSO token of an administrative user, such as `amAdmin`, who has access to perform the operation.

Java Example

The `RestSTSConsumer.java` sample code provides an example of how to consume a published REST STS instance programmatically. Tailor this example as required to provide programmatic consumption of your own REST STS instances.

Tip

For information on downloading and building AM sample source code, see How do I access and build the sample code provided for AM (All versions)? in the *Knowledge Base*.

You can find the STS code examples under `/path/to/openam-samples-external/sts-example-code`.

Important

The sample code referenced in this section is *not* compilable, because it uses classes that are not available publicly. The code provides patterns to developers familiar with the problem domain and is intended only to assist developers who want to programmatically consume REST STS instances.

Chapter 5 Querying, Validating, and Canceling Tokens

Both REST and SOAP STS instances support *token persistence*, which is the ability to store tokens issued for the STS instance in the Core Token Service (CTS). You enable token persistence for both REST and SOAP STS instances' configuration under Realms > *Realm Name* > STS > *STS Instance Name* > General Configuration > Persist Issued Tokens in Core Token Store. Tokens are saved in the CTS for the duration of the token lifetime, which is a configuration property for STS-issued SAML v2.0 and OpenID Connect tokens. Tokens with expired durations are periodically removed from the CTS.

With token persistence enabled for an STS instance, AM provides the ability to query, validate, and cancel tokens issued for the instance:

- *Querying tokens* means listing tokens issued for an STS instance or for a user.
- *Validating a token* means verifying that the token is still present in the CTS.
- *Cancelling a token* means removing the token from the CTS.

Invoking the sts-tokengen Endpoint

The `sts-tokengen` endpoint provides administrators with the ability to query and cancel tokens issued *for both REST and SOAP STS instances* using REST API calls.

When using the `sts-tokengen` endpoint, be sure to provide the token ID for an AM administrator, such as `amAdmin`, as the value of a header whose name is the name of the SSO token cookie, by default `iPlanetDirectoryPro`.

Querying Tokens

List tokens issued for an STS instance by using the `queryFilter` action in an HTTP GET call to the `sts-tokengen` endpoint with the `/sts-id` argument.

The following example lists all the tokens issued for the `username-transformer` STS instance. The results show that AM has issued two OpenID Connect tokens for the `demo` user for the `username-transformer` STS instance:

```
$ curl \
--request GET \
--header "iPlanetDirectoryPro: AQIC5..." \
https://openam.example.com:8443/openam/sts-tokengen?_queryFilter=\/sts_id+eq+\'username-transformer\'
{
    "result":[
        {
            "_id":"B663D248CE4C3B63A7422000B03B8F5E0F8E443B",
            "_rev":"",
            "token_id":"B663D248CE4C3B63A7422000B03B8F5E0F8E443B",
            "sts_id":"username-transformer",
            "principal_name":"demo",
            "token_type":"OPENIDCONNECT",
            "expiration_time":1459376096
        },
        {
            "_id":"7CB70009970D1AAFF177AC2A08D58405EDC35DF5",
            "_rev":"",
            "token_id":"7CB70009970D1AAFF177AC2A08D58405EDC35DF5",
            "sts_id":"username-transformer",
            "principal_name":"demo",
            "token_type":"OPENIDCONNECT",
            "expiration_time":1459376098
        }
    ],
    "resultCount":2,
    "pagedResultsCookie":null,
    "totalPagedResultsPolicy":"NONE",
    "totalPagedResults":-1,
    "remainingPagedResults":-1
}
```

List tokens issued for a particular user with the `queryFilter` action in an HTTP GET call to the `sts-tokengen` endpoint with the `/token-principal` argument.

The following example lists all the tokens issued for the `demo` user. The results show that AM has issued two OpenID Connect tokens:

```
$ curl \
--request GET \
--header "iPlanetDirectoryPro: AQIC5..." \
https://openam.example.com:8443/openam/sts-tokengen?_queryFilter=\/token_principal+eq+\'demo\'
{
    "result":[
        {
            "_id":"B663D248CE4C3B63A7422000B03B8F5E0F8E443B",
            "_rev":"",
            "token_id":"B663D248CE4C3B63A7422000B03B8F5E0F8E443B",
            "sts_id":"username-transformer",
            "principal_name":"demo",
            "token_type":"OPENIDCONNECT",
            "expiration_time":1459376096
        },
        {
            "_id":"7CB70009970D1AAFF177AC2A08D58405EDC35DF5",
            "_rev":"",
            "token_id":"7CB70009970D1AAFF177AC2A08D58405EDC35DF5",
            "sts_id":"username-transformer",
            "principal_name":"demo",
            "token_type":"OPENIDCONNECT",
            "expiration_time":1459376098
        }
    ],
    "resultCount":2,
    "pagedResultsCookie":null,
    "totalPagedResultsPolicy":"NONE",
    "totalPagedResults":-1,
    "remainingPagedResults":-1
}
```

Cancelling Tokens

Cancel tokens by making an HTTP DELETE call to the `sts-tokengen`/*token_id* endpoint:

```
$ curl \
--request DELETE \
--header "iPlanetDirectoryPro: AQIC5..." \
https://openam.example.com:8443/openam/sts-tokengen/B663D248CE4C3B63A7422000B03B8F5E0F8E443B
{
    "_id":"B663D248CE4C3B63A7422000B03B8F5E0F8E443B",
    "_rev":"B663D248CE4C3B63A7422000B03B8F5E0F8E443B",
    "result":"token with id B663D248CE4C3B63A7422000B03B8F5E0F8E443B successfully removed."
}
```

Validating and Cancelling Tokens by Invoking a REST STS Instance

REST STS users can validate and cancel tokens by making an HTTP POST call to a REST STS instance's endpoint.

To validate a token, use the `validate` action. The following example validates an OpenID Connect token previously issued by the `username-transformer` REST STS instance:

```
$ curl \
--request POST \
--header "iPlanetDirectoryPro: AQIC5..." \
--header "Content-Type: application/json" \
--data '{
    "validated_token_state": {
        "token_type": "OPENIDCONNECT",
        "oidc_id_token": "eyAidHlwIjogIkpXVCIsIC..."
    }
}' \
https://openam.example.com:8443/openam/rest-sts/username-transformer?_action=validate
{
    "token_valid":true
}
```

To cancel a token, use the `cancel` action. The following example cancels an OpenID Connect token previously issued by the `username-transformer` REST STS instance:

```
$ curl \
--request POST \
--header "iPlanetDirectoryPro: AQIC5..." \
--header "Content-Type: application/json" \
--data '{
    "cancelled_token_state": {
        "token_type": "OPENIDCONNECT",
        "oidc_id_token": "eyAidHlwIjogIkpXVCIsIC..."
    }
}' \
 https://openam.example.com:8443/openam/rest-sts/username-transformer?_action=cancel
{
    "result":"OPENIDCONNECT token cancelled successfully."
}
```

Validating and Cancelling Tokens by Invoking a SOAP STS Instance

The source code for the `validateToken` and `cancelToken` methods in the `org.forgerock.openam.sts.soap.SoapSTSConsumer` class provides information needed to construct WS-Trust 1.4-compliant calls for validating and cancelling tokens.

Locate the `org.forgerock.openam.sts.soap.SoapSTSConsumer` class under `openam-sts/openam-soap-sts/openam-soap-sts-client` in the AM source code.

Chapter 6
Supporting Custom Token Types

AM supports token transformations to and from a variety of token types, including username, SAML v2.0, OpenID Connect, and X.509. In addition to these supported token types, REST STS instances can use custom token types as the input or output token, or both, in a token transformation. When you configure a REST STS instance to support a token transformation that takes a custom token type, you can also configure a custom validator and provider class for the custom token type. AM uses custom validator classes to validate custom tokens and custom provider classes to produce custom tokens.

Tip

For information on downloading and building AM sample source code, see How do I access and build the sample code provided for AM (All versions)? in the *Knowledge Base*.

You can find the STS code examples under `/path/to/openam-samples-external/sts-example-code`.

Specify custom token validator and provider classes in the AM console by configuring the Custom Token Validators and Custom Token Providers properties under Realms > *Realm Name* > STS > *REST STS Instance Name*.

A custom validator class can be used in transformations that produce standard STS output tokens, such as SAML v2.0 tokens or OpenID Connect tokens, and in transformations that produce custom output token types.

A custom provider class can be used in token transformations that take standard STS input tokens, such as username tokens or AM SSO tokens, and in transformations that take custom input token types.

Before a REST STS instance can use a custom token type validator or provider class, you must bundle the class into the AM `.war` file and restart AM.

AM invokes a single instance of a validator or provider class to run all concurrently dispatched token transformations that use the custom token type. Because there is only a single instance of the class, you must code custom validator and provider classes to be thread-safe.

Developing Custom Token Type Validator Classes

To create a custom token type validator class, implement the `org.forgerock.openam.sts.rest.token.validator.RestTokenTransformValidator` class.

Custom token type validator classes implement the `validateToken` method. This method takes a `RestTokenValidatorParameters` object as input. Note that the generic type of `RestTokenValidatorParameters` is `org.forgerock.json.fluent.JsonValue`. As a result of using this type, custom validator classes can access the JSON representation of the input token passed to the REST STS instance in the `input_token_state` JSON key.

The `validateToken` method returns an `org.forgerock.openam.sts.rest.token.validator.RestTokenTransformValidatorResult` object. At a minimum, this object contains the AM SSO token corresponding to the validated principal. It can also contain additional information specified as a JSON value, allowing a custom validator to pass extra state to a custom provider in a token transformation.

Developing Custom Token Type Provider Classes

To create a custom token type provider class, implement the `org.forgerock.openam.sts.rest.token.provider.RestTokenProvider` class.

Custom token type provider classes implement the `createToken` method. This method takes an `org.forgerock.openam.sts.rest.token.provider.CustomRestTokenProviderParameters` object as input. This object gives the custom provider access to the following information:

- The principal returned by the `RestTokenTransformValidator`
- The AM SSO token corresponding to the validated principal
- Any additional state returned in the `RestTokenValidatorResult` object
- The type of input token validated by the `RestTokenTransformValidator` in the token transformation
- The `JsonValue` corresponding to this validated token, as specified by the `input_token_state` object in the transformation request
- The `JsonValue` corresponding to the `token_output_state` object specified in the token transformation request (which can provide additional information pertinent to the creation of the output token)

The `createToken` method returns a string representation of the custom token in a format that can be transmitted across HTTP in JSON. It should be base64-encoded if binary.

Using Custom Token Type Validators and Providers

This section provides an example of how to use custom token type validators and providers.

The example assumes that you already configured a token transformation by completing the following tasks:

- Implementing the `RestTokenTransformValidator` interface to create a custom token type validator

- Implementing the `RestTokenProvider` interface to create a custom token type provider
- Bundling the two classes into the AM `.war` file
- Restarting AM
- Publishing a REST STS instance with a custom token type named `CUSTOM`, specifying the custom validator and provider classes in the instance's configuration

To transform a `CUSTOM` token to an OpenID Connect token, you might specify a JSON payload similar to the following:

```
{
    "input_token_state":
        {
            "token_type": "CUSTOM",
            "extra_stuff": "very_useful_state"
        },
    "output_token_state":
        {
            "token_type": "OPENIDCONNECT",
            "nonce": "1234",
            "allow_access": true
        }
}
```

With the preceding JSON payload, AM passes a `JsonValue` instance to the `validateToken` method of the custom token type validator class as follows:

```
{
    "token_type": "CUSTOM",
    "extra_stuff": "very_useful_state"
}
```

To transform a username token to a `CUSTOM` token, you might specify a JSON payload similar to the following:

```
{
    "input_token_state":
        {
            "token_type": "USERNAME",
            "username": "unt_user17458687",
            "password": "password"
        },
    "output_token_state":
        {
            "token_type": "CUSTOM",
            "extra_stuff_for_custom": "some_useful_information"
        }
}
```

With the preceding JSON payload, AM passes the following information to the `createToken` method of the custom token type provider:

- The principal returned by the `USERNAME` token validator: `unt_user17458687`.
- The AM SSO token corresponding to this authenticated principal.
- Additional state returned by the token validator, if any. Because the `USERNAME` token validator does not return any additional state, the additional state for this example would be null.
- The input token type: `CUSTOM`
- A `JsonValue` representation of the following:

```
{
    "token_type": "USERNAME",
    "username": "unt_user17458687",
    "password": "password"
}
```

- A `JsonValue` representation of the following:

```
{
    "token_type": "CUSTOM",
    "extra_stuff_for_custom": "some_useful_information"
}
```

To transform a `CUSTOM` token to a `CUSTOM` token, you might specify a JSON payload similar to the following:

```
{
    "input_token_state":
        {
            "token_type": "CUSTOM",
            "extra_stuff": "very_useful_state"
        },
    "output_token_state":
        {
            "token_type": "CUSTOM",
            "extra_stuff_for_custom": "some_useful_information"
        }
}
```

The input to the custom validator and provider would be similar to the preceding examples, with the possible addition of any additional state that the custom validator returned from the `validateToken` method.

Chapter 7 Reference

This reference section covers configuration settings for AM's Security Token Service.

This section covers the following settings:

- "The Publish Service"
- "REST STS Configuration Properties"
- "SOAP STS Configuration Properties"
- "Shared STS Configuration Properties"

The Publish Service

To publish an STS instance, perform an HTTP POST on the `/sts-publish/rest` endpoint, specifying the `_action=create` parameter in the URL.

For example, you could publish a REST STS instance named `username-transformer` in the Top Level Realm as follows:

```
$ curl \
--request POST \
--header "iPlanetDirectoryPro: AQIC5..." \
--header "Content-Type: application/json" \
--data '{
    "invocation_context": "invocation_context_client_sdk",
    "instance_state":
    {
        "saml2-config":
        {
            "issuer-name":"saml2-issuer",
            ...
        },
        "deployment-config":
        {
            "deployment-url-element":"username-transformer",
            "deployment-realm":"/",
            ...
        },
        "persist-issued-tokens-in-cts":"false",
        "supported-token-transforms":[{
            "inputTokenType":"USERNAME",
            "outputTokenType":"OPENIDCONNECT",
            "invalidateInterimOpenAMSession":false
        }],
        "oidc-id-token-config":{
            "oidc-issuer":"test",
            ...
        }
    }
}' \
https://openam.example.com:8443/openam/sts-publish/rest?_action=create
{
  "_id":"username-transformer",
  "_rev":"21939129",
  "result":"success",
  "url_element":"username-transformer"}
}
```

The `instance_state` object in the JSON payload represents the STS instance's configuration. For a complete example of an `instance_state` object, see the sample code for the `RestSTSInstancePublisher` class in "Publishing REST STS Instances".

Accessing the `sts-publish` endpoint requires administrative privileges. Authenticate as an AM administrative user, such as `amAdmin`, before attempting to publish an STS instance.

In addition to publishing instances, the `sts-publish` endpoint can also return the configuration of an STS instance when you perform an HTTP GET on endpoint for the instance, such as `/sts-publish/rest/realm/deployment-URL-element`.

In the endpoint, *deployment-URL-element* is the value of the STS instance's deployment URL element —one of the instance's configuration properties. *realm* is the realm in which the instance has been configured.

For example, you could obtain the configuration of a REST STS instance configured in the Top Level Realm with the deployment URL element `username-transformer` as follows:

```
$ curl \
--request GET \
--header "iPlanetDirectoryPro: AQIC5..." \
https://openam.example.com:8443/openam/sts-publish/rest/username-transformer
{
   "_id":"username-transformer",
   "_rev":"-659999943",
   "username-transformer":{
      "saml2-config":{
         "issuer-name":"saml2-issuer",
         ...
      },
      "deployment-config":{
         "deployment-url-element":"username-transformer",
         ...
      },
      "persist-issued-tokens-in-cts":"false",
      "supported-token-transforms":[
         {
            "inputTokenType":"USERNAME",
            "outputTokenType":"OPENIDCONNECT",
            "invalidateInterimOpenAMSession":false
         }
      ],
      "oidc-id-token-config":{
         "oidc-issuer":"test",
         ...
      }
   }
}
```

You can delete STS instances by performing an HTTP DELETE on the `sts-publish` endpoint:

- For REST STS instances, perform an HTTP DELETE on `/sts-publish/rest/realm/deployment-URL-element`.
- For SOAP STS instances, perform an HTTP DELETE on `/sts-publish/soap/realm/deployment-URL-element`.

Caution

SOAP STS instances are deprecated and cannot be deployed in ths version of AM. If you delete your instances, you will not be able to redeploy them.

Publishing REST STS Instances

The sample code referenced in this section provides an example of how to programmatically publish REST STS instance. The code is not intended to be a working example. Rather, it is a starting point—code that you can modify to satisfy your organization's specific requirements.

For information on downloading and building AM sample source code, see How do I access and build the sample code provided for AM (All versions)? in the *Knowledge Base*.

You can find the STS code examples under `/path/to/openam-samples-external/sts-example-code`.

After publishing a REST STS instance programmatically, you can view the instance's configuration in the AM console. The instance is ready for consumption.

Sample code is available for the following classes:

RestSTSInstancePublisher

The `RestSTSInstancePublisher` class exposes an API to publish, delete, and update REST STS instances by calling methods that perform an HTTP POST operation on the `soap-sts/publish` endpoint.

RestSTSInstanceConfigFactory

The `RestSTSInstancePublisher`class calls the `RestSTSInstanceConfigFactory` class to create a `RestSTSInstanceConfig` instance. `RestSTSInstanceConfig` objects encapsulate all the configuration information of a REST STS instance, and emit JSON values that you can post to the `sts-publish/rest` endpoint to publish a REST STS instance.

STSPublishContext

The sample `STSPublishContext` class specifies the configuration necessary to publish REST and SOAP STS instances. The class provides a programmatic method for setting configuration properties—the same configuration properties available through the AM console under Realms > *Realm Name* > STS.

CustomTokenOperationContext

The sample `CustomTokenOperationContext` class specifies custom validators, token types, and transformations that a REST STS instance can support.

Important

The sample code referenced in this section is *not* compilable, because it uses classes that are not available publicly. The code provides patterns to developers familiar with the problem domain and is intended only to assist developers who want to programmatically publish REST STS instances.

The sample code imports a number of classes, introducing dependencies. Classes imported from the AM API can remain in your code, but other imported classes must be removed and replaced with code that provides similar functionality in your environment. For example, the `RestSTSInstanceConfigFactory` class uses a constant named `CommonConstants.DEFAULT_CERT_MODULE_NAME` from the imported `com.forgerock.openam.functionaltest.sts.frmwk.common.CommonConstants` utility class. This utility class is not publicly available. Therefore, you need to replace this constant with another construct.

The critical part of the sample code is the idioms that programmatically set all the state necessary to publish a REST STS instance.

REST STS Configuration Properties

Deployment Url Element

Specifies a string that identifies this REST STS instance.

The Deployment Url Element is a component of the REST STS instance's endpoint. For example, if you specified `myRESTSTSInstance` as the Deployment Url Element, the REST STS endpoint would be `rest-sts/myRealm/myRESTSTSInstance`.

General Configuration Properties

The following are general configuration properties for REST STS instances:

Persist Issued Tokens in Core Token Store

Specifies whether to enable token persistence in the Core Token Service (CTS).

AM saves all STS-issued tokens to CTS when token persistence is enabled. A token's lifetime in CTS has the same length as the Token Lifetime property specified for issued tokens.

STS token validation and cancellation capabilities require tokens to be present in CTS. Therefore, if your deployment requires token validation and cancellation, you must enable token persistence.

Supported Token Transforms

Specifies one or more token transformations supported by this REST STS instance. Token transformations are listed in the AM console using the notation *`input_token_type -> output_token_type`*.

For each supported token transformation, AM provides an option to invalidate the interim AM session. When transforming a token, the STS creates an AM session. If desired, you can invalidate the AM session after token transformation is complete.

Custom Token Validators

Specifies a validator class for a custom token type.

Use the format *`CUSTOM_TOKEN_TYPE|custom_validator_class`* to specify each validator class. For example, `CUSTOM|org.mycompany.tokens.myCustomTokenValidator`.

For more information about custom token validators, see "*Supporting Custom Token Types*".

Custom Token Providers

Specifies a provider class for a custom token type.

Use the format `CUSTOM_TOKEN_TYPE|custom_provider_class`. To specify each provider class. For example, `CUSTOM|org.mycompany.tokens.myCustomTokenProvider`.

For more information about custom token providers, see "*Supporting Custom Token Types*".

Custom Token Transforms

Specifies one or more token transformations that take a custom token type as the input or output token. If you specify a custom token validator or provider, you must also specify a custom token transform.

Specify the custom transform using three values separated by the vertical bar character | as follows:

1. The input token type
2. The output token type
3. Whether to invalidate the AM session created during token transformation. Specify `TRUE` to invalidate the session or `FALSE` to let the session remain valid.

For example, a value of `CUSTOM|SAML2|TRUE` configures a token transformation that transforms a `CUSTOM` token to a SAML v2.0 assertion and then invalidates the created AM session.

Deployment Configuration Properties

The following are deployment configuration properties for REST STS instances:

Authentication Target Mappings

Specifies one or more mappings that define how the REST STS instance authenticates input tokens.

Each mapping is a set of arguments separated by the vertical bar character | as follows:

1. (Required) The input token type: `USERNAME`, `OPENAM`, `X509`, `OPENIDCONNECT`, or a custom token type.
2. (Required) The value `service` or `module`. If the third argument is an authentication chain, specify `service`. If the third argument is an authentication module, specify `module`.
3. (Required) The name of an AM authentication chain or module to which the input token is authenticated.
4. (Optional) The name of the header to place the token in when authenticating to AM. Specify this parameter for input `X509` and `OPENIDCONNECT` tokens as follows:
    - For `X509` input tokens, the format is `x509_token_auth_target_header_key=`*`Header Name`*.
    - For `OPENIDCONNECT` input tokens, the format is `oidc_id_token_auth_target_header_key=`*`Header Name`*.

Be sure to specify the header names configured in the Certificate or OpenID Connect id_token bearer authentication module properties as the *Header Name* argument.

This argument can also be used with custom token types to specify the name of a header or cookie from which to obtain a token. When using this argument with a custom token type, its format is determined by the custom validator class that validates the custom token type.

The following are example mappings:

- `USERNAME|service|myLDAPChain` configures STS to authenticate input `USERNAME` tokens to the `myLDAPChain` authentication chain.
- `X509|module|CertModule|x509_token_auth_target_header_key=ClientCert` configures STS to obtain an X.509 certificate from the `ClientCert` header, use it as the input token, and authenticate it using the `CertModule` authentication module.

Client Certificate Header Key

Specifies the name of a header that a TLS offloader should use to use to transmit client certificates.

Token transformations that take an X.509 certificate as the input token require the certificate to be presented using two-way TLS, so that the TLS handshake can validate client certificate ownership. A common way of obtaining the client certificate with two-way TLS is to use the `javax.servlet.request.X509Certificate` attribute in the servlet request.

However, in deployments with TLS offloading, the offloader must use an HTTP header to transmit the certificate to its destination. This configuration property is the name of the HTTP header whose value contains the certificate.

Trusted Remote Hosts

Specifies one or more IP addresses of hosts trusted to transmit client X.509 certificates in deployments with TLS offloading.

To allow any host to transmit a certificate, specify `any` as the value of this property.

As with the Client Certificate Header Key property, configure this property for deployments with TLS offloading.

SOAP STS Configuration Properties

Deployment Url Element

Specifies a string that identifies this SOAP STS instance.

The Deployment Url Element is a component of the SOAP STS instance's endpoint. For example, if you specified `mySOAPSTSInstance` as the Deployment Url Element, the SOAP STS endpoint would be `/`*`SOAP STS .war File Name`*`myRealm/mySOAPSTSInstance`.

General Configuration Properties

The following are general configuration properties for SOAP STS instances:

Persist Issued Tokens in Core Token Store

Specifies whether to enable token persistence in the Core Token Service (CTS).

AM saves all STS-issued tokens to CTS when token persistence is enabled. A token's lifetime in CTS has the same length as the Token Lifetime property specified for issued tokens.

STS token validation and cancellation capabilities require tokens to be present in CTS. Therefore, if your deployment requires token validation and cancellation, you must enable token persistence.

Issued Tokens

Specifies the types of tokens that this SOAP STS instance issues as output tokens for token transformations.

Security Policy Validated Token

Specifies the `SupportingToken` type in the WS-SecurityPolicy bindings in the SOAP STS deployment's WSDL, and whether the AM session created during token transformation should be invalidated after the token is issued.

Deployment Configuration Properties

The following are deployment configuration properties for SOAP STS instances:

Authentication Target Mappings

Specifies one or more mappings that define how the SOAP STS instance should authenticate input tokens.

Each mapping is a set of arguments separated by the | character as follows:

1. (Required) The input token type: `USERNAME`, `OPENAM`, or `X509`.
2. (Required) The value `service` or `module`. If the third argument is an authentication chain, specify `service`. If the third argument is an authentication module, specify `module`.
3. (Required) The name of an AM authentication chain or module to which the input token is authenticated.
4. (Optional) The name of the header in which to place the token when authenticating to AM. For `X509` input tokens, the format is `x509_token_auth_target_header_key=`*`Header Name`*.

   Be sure to specify the header name configured in the Certificate authentication module properties as the *Header Name* argument.

The following are example mappings:

- `USERNAME|service|myLDAPChain` configures STS to authenticate input `USERNAME` tokens to the `myLDAPChain` authentication chain.
- `X509|module|CertModule|x509_token_auth_target_header_key=ClientCert` configures STS to obtain an X.509 certificate from the `ClientCert` header, use it as the input token, and authenticate it using the `CertModule` authentication module.

OpenAM URL

Specifies the AM URL. For example, `https://openam.example.com:8443/openam`.

Wsdl File Referencing Security Policy Binding Selection

Specifies a supporting token type and security policy binding to protect the SOAP STS instance. This choice will determine the SecurityPolicy bindings in the wsdl file defining the WS-Trust API.

If you select the `Custom wsdl file` option, you must provide the path to a custom WSDL file in the Custom wsdl File property.

Custom wsdl File

Specifies the path to a custom WSDL file that defines the WS-Trust API.

Custom Service QName

Specifies the `name` attribute of the `wsdl:service` element. Configure this property when using a custom WSDL file.

Custom Port QName

Specifies the `name` attribute of the `wsdl:port` element. Configure this property when using a custom WSDL file.

Delegation Relationships Supported

Enable this option if the request security token messages can include `wst14:ActAs` or `wst:OnBehalfOf` parameters. Note that you must enable this option if the SOAP STS instance issues SAML v2.0 assertions with `SenderVouches` subject confirmations.

Delegated Token Types

Specifies the types of validation support to enable in the SOAP STS instance for `USERNAME` and `OPENAM` tokens in `wst14:ActAs` or `wst:OnBehalfOf` parameters specified in request security token messages.

If the SOAP STS instance supports delegated relationships, configure either the Delegated Token Types property or the Custom Delegation Handlers property, but not both properties.

Custom Delegation Handlers

Specifies custom handlers that implement the `org.apache.cxf.sts.token.delegation.TokenDelegationHandler` interface. The handlers provide validation support for the tokens in `wst14:ActAs` or `wst:OnBehalfOf` parameters specified in request security token messages. Custom delegation handlers are typically used when the tokens are custom tokens.

If the SOAP STS instance supports delegated relationships, configure either the Delegated Token Types property or the Custom Delegation Handlers property, but not both properties.

SOAP Keystore Configuration Properties

The following are SOAP keystore configuration properties for SOAP STS instances:

Soap Keystore Location

Specifies the path to a JKS keystore containing keys for signing and encryption when using the symmetric and asymmetric bindings with SOAP messaging. Specify an absolute path or a location in the AM classpath.

Note that the Wsdl File Referencing Security Policy Binding Selection property determines the binding for a SOAP STS instance.

AM provides a JKS keystore with demo keys, `/path/to/openam/security/keystores/keystore.jks`. For more information about keystores in AM, see "*Configuring Secrets, Certificates, and Keys*" in the *Security Guide*.

Keystore Password

Specifies the password used to decrypt the keystore.

Signature Key Alias

Specifies the key alias in the keystore used to sign messages from this SOAP STS instance. You must configure this property when using asymmetric binding.

Signature Key Password

Specifies the password for the signature key.

Decryption Key Alias

Specifies the key alias in the keystore used by this SOAP STS instance to decrypt client messages for the asymmetric binding, and to decrypt the client-generated symmetric key for the symmetric binding.

Decryption Key Password

Specifies the password for the decryption key.

Shared STS Configuration Properties

These properties are available in both the REST and STS configuration pages.

Issued SAML v2.0 Token Configuration Properties

This section lists configuration properties associated with STS-issued SAML v2.0 assertions for both REST and SOAP STS instances. The properties fall into two categories:

1. Properties that determine content in STS-issued SAML v2.0 assertion. For information about SAML v2.0 assertions, see Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0.
2. Properties that determine how the issued SAML v2.0 assertion is signed or encrypted.

The SAML2 issuer Id

Specifies the IDP entity ID. Populates the `Issuer` element of the SAML v2.0 assertion.

Service Provider Entity Id

Specifies an audience attribute value. Populates the `AudienceRestriction` sub-element of the `Conditions` element of the SAML v2.0 assertion.

This value is required when issuing Bearer assertions.

Service Provider Assertion Consumer Service Url

Specifies a recipient attribute value. Populates the `Recipient` sub-element of the `SubjectConfirmation` element of the SAML v2.0 assertion.

The scheme, FQDN, and port configured must exactly match those of the service provider as they appear in its metadata.

This value is required when issuing Bearer assertions.

NameIdFormat

Specifies the name identifier format for the SAML v2.0 assertion.

Token Lifetime

Specifies the lifetime, in seconds, for the assertion. The default is 600 seconds.

Custom Conditions Provider Class Name

Specifies the name of a custom class that generates a `Conditions` element in the SAML v2.0 assertion. This property is optional: use a custom class when the `Conditions` element created by the default provider does not meet your needs.

The class must implement the `org.forgerock.openam.sts.tokengeneration.saml2.statements.ConditionsProvider` interface, and must be bundled in the AM `.war` file.

Customs Subject Provider Class Name

Specifies the name of a custom class that generates a `Subject` element in the SAML v2.0 assertion. This property is optional: use a custom class when the `Subject` element created by the default provider does not meet your needs.

The class must implement the `org.forgerock.openam.sts.tokengeneration.saml2.statements.SubjectProvider` interface and must be bundled in the AM `.war` file.

Custom AuthenticationStatements Class Name

Specifies the name of a custom class that generates an `AuthnStatement` element in the SAML v2.0 assertion. This property is optional: use a custom class when the `AuthnStatement` element created by the default provider does not meet your needs.

The class must implement the `org.forgerock.openam.sts.tokengeneration.saml2.statements.AuthenticationStatementsProvider` interface and must be bundled in the AM `.war` file.

Custom AttributeStatements Class Name

Specifies the name of a custom class that generates an `AttributeStatement` element in the SAML v2.0 assertion. This property is optional: use a custom class when the `AttributeStatement` element created by the default provider does not meet your needs.

The class must implement the `org.forgerock.openam.sts.tokengeneration.saml2.statements.AttributeStatementsProvider` interface and must be bundled in the AM `.war` file.

Custom Authorization Decision Statements Class Name

Specifies the name of a custom class that generates an `AuthzDecisionStatement` element in the SAML v2.0 assertion. This property is optional: use a custom class when the `AuthzDecisionStatement` element created by the default provider does not meet your needs.

The class must implement the `org.forgerock.openam.sts.tokengeneration.saml2.statements.AuthzDecisionStatementsProvider` interface and must be bundled in the AM `.war` file.

Custom Attribute Mapper Class Name

Specifies the name of a custom attribute mapper class. An attribute mapper generates `attribute` elements to be included in the SAML v2.0 assertion.

The class must implement the `org.forgerock.openam.sts.tokengeneration.saml2.statements.AttributeMapper` interface and must be bundled in the AM `.war` file.

Custom Authentication Context Class Name

Specifies the name of a custom class that generates an `AuthnContext` element in the SAML v2.0 assertion. This property is optional: use a custom class when the `AuthnContext` element created by the default provider does not meet your needs.

The class must implement the `org.forgerock.openam.sts.tokengeneration.saml2.statements.AuthnContextMapper` interface and must be bundled in the AM `.war` file.

By default, AM generates the `AuthnContext` element based on the input token type as follows:

- For input AM tokens: `urn:oasis:names:tc:SAML:2.0:ac:classes:PreviousSession`
- For input username tokens and OpenID Connect ID tokens: `urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport`
- For input X.509 tokens: `urn:oasis:names:tc:SAML:2.0:ac:classes:X509`

Attribute Mappings

Configures mappings between SAML v2.0 attribute names—*map keys*—and AM user profile attributes or session properties in order to generate `Attribute` elements in the SAML v2.0 assertion.

AM's default attribute mapper generates `Attribute` elements as follows:

- The map key populates the `Attribute` element's `Name` property.
- The user profile or session property value populates the `Attribute` element's `AttributeValue` property.

When specifying map keys in the Attribute Mappings property, use the following format: `[NameFormatURI]|SAML_ATTRIBUTE_NAME`.

Map values enclosed in quotes are included in the attribute without mapping. Specify `';binary'` at the end of a map value for attributes that have binary values.

The following are examples of attribute mappings:

- `EmailAddress=mail`
- `Address=postaladdress`
- `urn:oasis:names:tc:SAML:2.0:attrname-format:uri|urn:mace:dir:attribute-def:cn=cn`
- `partnerID="staticPartnerIDValue"`
- `urn:oasis:names:tc:SAML:2.0:attrname-format:uri|nameID="staticNameIDValue"`
- `photo=photo;binary`
- `urn:oasis:names:tc:SAML:2.0:attrname-format:uri|photo=photo;binary`

Sign Assertion

Specifies whether or not to sign the SAML v2.0 assertion.

When enabling assertion signing, you must also specify the KeystorePath, Keystore Password, Signature Key Alias, and Signature Key Password properties.

Encrypt Assertion

Specifies whether to encrypt the entire SAML v2.0 assertion. When enabling assertion encryption:

- You must also specify the KeystorePath, Keystore Password, and Encryption Key Alias properties.
- You must not specify the Encrypt Attributes or Encrypt NameID options.

The Encryption Key Alias corresponds to the public key of the service provider that is the intended audience of the assertion. SAML v2.0 assertion encryption works as follows:

1. AM generates a symmetric key.
2. AM encrypts the symmetric key with the recipient's public key.
3. AM includes the encrypted key in the part of the assertion that is not symmetric key-encrypted.
4. The service provider—owner of the corresponding private key—uses the private key to decrypt the symmetric key included in the assertion.
5. The service provider can then use the decrypted symmetric key to decrypt the assertion.

Encrypt Attributes

Specifies whether to encrypt the assertion's attributes only. When specifying this option, do not specify the Encrypt Assertion option.

When encrypting attributes, you must also specify the KeystorePath, Keystore Password, and Encryption Key Alias properties.

Encrypt NameID

Specifies whether to encrypt the assertion's NameID only. When specifying this option, do not specify the Encrypt Assertion option.

When encrypting the NameID, you must also specify the KeystorePath, Keystore Password, and Encryption Key Alias properties.

Encryption Algorithm

Specifies the encryption algorithm to use when encrypting the entire assertion, the assertion's attributes, or the NameID.

Key Transport Algorithm

Specifies the algorithm used to encrypt the symmetric encryption key when SAML v2.0 token encryption is enabled. Possible values are:

- http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p.
- http://www.w3.org/2009/xmlenc11#rsa-oaep.

  When this algorithm is configured, AM will use the Mask Generation Function Algorithm property (Configure > Global Services > Common Federation Configuration) to encrypt the transport key.

  For a list of supported mask generation function algorithms, see "Algorithms" in the *Reference*.
- http://www.w3.org/2001/04/xmlenc#rsa-1_5

KeystorePath

Specifies the path to the JKS keystore containing the key aliases for encrypting and signing SAML assertions. Specify an absolute path or a location in the AM classpath.

AM provides a JKS keystore with demo keys, `/path/to/openam/security/keystores/keystore.jks`. For more information about keystores in AM, see "*Configuring Secrets, Certificates, and Keys*" in the *Security Guide*.

Keystore Password

Specifies the password used to decrypt the keystore.

Encryption Key Alias

Specifies the key alias in the keystore that holds the service provider's X.509 certificate for this STS instance. This key alias is used to encrypt assertions.

Token Lifetime

Specifies the lifetime, in seconds, for the assertion. The default is 600 seconds.

Custom Conditions Provider Class Name

Specifies the name of a custom class that generates a `Conditions` element in the SAML v2.0 assertion. This property is optional: use a custom class when the `Conditions` element created by the default provider does not meet your needs.

The class must implement the `org.forgerock.openam.sts.tokengeneration.saml2.statements.ConditionsProvider` interface, and must be bundled in the AM `.war` file.

Customs Subject Provider Class Name

Specifies the name of a custom class that generates a `Subject` element in the SAML v2.0 assertion. This property is optional: use a custom class when the `Subject` element created by the default provider does not meet your needs.

The class must implement the `org.forgerock.openam.sts.tokengeneration.saml2.statements.SubjectProvider` interface and must be bundled in the AM `.war` file.

Custom AuthenticationStatements Class Name

Specifies the name of a custom class that generates an `AuthnStatement` element in the SAML v2.0 assertion. This property is optional: use a custom class when the `AuthnStatement` element created by the default provider does not meet your needs.

The class must implement the `org.forgerock.openam.sts.tokengeneration.saml2.statements.AuthenticationStatementsProvider` interface and must be bundled in the AM `.war` file.

Custom AttributeStatements Class Name

Specifies the name of a custom class that generates an `AttributeStatement` element in the SAML v2.0 assertion. This property is optional: use a custom class when the `AttributeStatement` element created by the default provider does not meet your needs.

The class must implement the `org.forgerock.openam.sts.tokengeneration.saml2.statements.AttributeStatementsProvider` interface and must be bundled in the AM `.war` file.

Custom Authorization Decision Statements Class Name

Specifies the name of a custom class that generates an `AuthzDecisionStatement` element in the SAML v2.0 assertion. This property is optional: use a custom class when the `AuthzDecisionStatement` element created by the default provider does not meet your needs.

The class must implement the `org.forgerock.openam.sts.tokengeneration.saml2.statements.AuthzDecisionStatementsProvider` interface and must be bundled in the AM `.war` file.

Custom Attribute Mapper Class Name

Specifies the name of a custom attribute mapper class. An attribute mapper generates `attribute` elements to be included in the SAML v2.0 assertion.

The class must implement the `org.forgerock.openam.sts.tokengeneration.saml2.statements.AttributeMapper` interface and must be bundled in the AM `.war` file.

Custom Authentication Context Class Name

Specifies the name of a custom class that generates an `AuthnContext` element in the SAML v2.0 assertion. This property is optional: use a custom class when the `AuthnContext` element created by the default provider does not meet your needs.

The class must implement the `org.forgerock.openam.sts.tokengeneration.saml2.statements.AuthnContextMapper` interface and must be bundled in the AM `.war` file.

By default, AM generates the `AuthnContext` element based on the input token type as follows:

- For input AM tokens: `urn:oasis:names:tc:SAML:2.0:ac:classes:PreviousSession`
- For input username tokens and OpenID Connect ID tokens: `urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport`

- For input X.509 tokens: `urn:oasis:names:tc:SAML:2.0:ac:classes:X509`

Attribute Mappings

Configures mappings between SAML v2.0 attribute names—*map keys*—and AM user profile attributes or session properties in order to generate `Attribute` elements in the SAML v2.0 assertion.

AM's default attribute mapper generates `Attribute` elements as follows:

- The map key populates the `Attribute` element's `Name` property.
- The user profile or session property value populates the `Attribute` element's `AttributeValue` property.

When specifying map keys in the Attribute Mappings property, use the following format: `[NameFormatURI]|SAML_ATTRIBUTE_NAME`.

Map values enclosed in quotes are included in the attribute without mapping. Specify `';binary'` at the end of a map value for attributes that have binary values.

The following are examples of attribute mappings:

- `EmailAddress=mail`
- `Address=postaladdress`
- `urn:oasis:names:tc:SAML:2.0:attrname-format:uri|urn:mace:dir:attribute-def:cn=cn`
- `partnerID="staticPartnerIDValue"`
- `urn:oasis:names:tc:SAML:2.0:attrname-format:uri|nameID="staticNameIDValue"`
- `photo=photo;binary`
- `urn:oasis:names:tc:SAML:2.0:attrname-format:uri|photo=photo;binary`

Sign Assertion

Specifies whether or not to sign the SAML v2.0 assertion.

When enabling assertion signing, you must also specify the KeystorePath, Keystore Password, Signature Key Alias, and Signature Key Password properties.

Encrypt Assertion

Specifies whether to encrypt the entire SAML v2.0 assertion. When enabling assertion encryption:

- You must also specify the KeystorePath, Keystore Password, and Encryption Key Alias properties.

- You must not specify the Encrypt Attributes or Encrypt NameID options.

The Encryption Key Alias corresponds to the public key of the service provider that is the intended audience of the assertion. SAML v2.0 assertion encryption works as follows:

1. AM generates a symmetric key.
2. AM encrypts the symmetric key with the recipient's public key.
3. AM includes the encrypted key in the part of the assertion that is not symmetric key-encrypted.
4. The service provider—owner of the corresponding private key—uses the private key to decrypt the symmetric key included in the assertion.
5. The service provider can then use the decrypted symmetric key to decrypt the assertion.

Encrypt Attributes

Specifies whether to encrypt the assertion's attributes only. When specifying this option, do not specify the Encrypt Assertion option.

When encrypting attributes, you must also specify the KeystorePath, Keystore Password, and Encryption Key Alias properties.

Encrypt NameID

Specifies whether to encrypt the assertion's NameID only. When specifying this option, do not specify the Encrypt Assertion option.

When encrypting the NameID, you must also specify the KeystorePath, Keystore Password, and Encryption Key Alias properties.

Encryption Algorithm

Specifies the encryption algorithm to use when encrypting the entire assertion, the assertion's attributes, or the NameID.

KeystorePath

Specifies the path to the JKS keystore containing the key aliases for encrypting and signing SAML assertions. Specify an absolute path or a location in the AM classpath.

AM provides a JKS keystore with demo keys, `/path/to/openam/security/keystores/keystore.jks`. For more information about keystores in AM, see "*Configuring Secrets, Certificates, and Keys*" in the *Security Guide*.

Keystore Password

Specifies the password used to decrypt the keystore.

Encryption Key Alias

Specifies the key alias in the keystore that holds the service provider's X.509 certificate for this STS instance. This key alias is used to encrypt assertions.

Signature Key Alias

Specifies the private key alias in the keystore used to sign assertions.

Signature Key Password

Specifies the password of the private key used to sign the assertion.

Issued OpenID Connect Token Configuration Properties

This section lists configuration properties associated with STS-issued OpenID Connect tokens for both REST and SOAP STS instances. The properties fall into two categories:

1. Properties that determine content in the issued OpenID Connect ID token. For information about OpenID Connect ID tokens, see the OpenID Connect Core 1.0 specification.
2. Properties that determine how the issued token is signed.

An STS instance configured to issue OpenID Connect tokens models the relationship between an OpenID Connect token provider and relying party. In other words, an STS instance issues tokens for a particular OAuth 2.0 client. The tokens contain `aud` and `azp` claims for the OAuth 2.0 client, and signing key state corresponding to a token provider.

In this model, when users call an STS instance to generate an OpenID Connect ID token, the process is analogous to the exchange between an OAuth 2.0 authorization server and resource owner following the initial redirection from an OAuth 2.0 client initiating the implicit flow. The STS instance returns the OpenID Connect ID token that corresponds to the authorization server's authentication of the resource owner. AM authenticates one of the following:

- For REST STS, the token specified as the `input_token_state` for the token transformation
- For SOAP STS, the supporting token necessary to traverse the SecurityPolicy bindings protecting the WS-Trust operation

Implicit in this model is the notion that an OpenID Connect ID token has value outside of an OAuth 2.0 flow, and that an OAuth 2.0 client, as a relying party, could be generalized as a SAML v2.0 service provider. The ID token is not simply an an entity-provided verifiable authorized access to a specific resource, but rather a generic service provider that consumes an OpenID Connect ID token to authenticate and authorize the subject asserted by the token.

Therefore, the configuration of an STS instance that issues OpenID Connect ID tokens contains information that defines the token provider and relying party.

Note that the `nonce` claim in the ID token is not a configuration property of an STS instance. STS consumers requesting an output OpenID Connect token provide a `nonce` value when making token transformation requests.

OpenID Connect Token Provider ID

Specifies the OpenID Connect token provider issuer ID. Populates the `iss` claim of the ID token.

Token Lifetime

Specifies, in seconds, the ID token's expiration. Populates the `exp` claim of the ID token.

Token signature algorithm

Specifies an HMAC or RSA algorithm used to sign ID tokens.

Public key reference type

Specifies how public keys should be referenced in issued ID tokens signed with RSA. OpenID Connect ID tokens are issued as JSON web tokens (JWTs). Tokens can reference RSA public keys as JSON web keys (JWKs), or not at all.

Used with RSA signing.

KeyStore Location

Specifies the path to the JKS keystore containing the key alias for signing the ID token. Specify an absolute path or a location in the AM classpath.

Used with RSA signing.

AM provides a JKS keystore with demo keys, `/path/to/openam/security/keystores/keystore.jks`. For more information about keystores in AM, see "*Configuring Secrets, Certificates, and Keys*" in the *Security Guide*.

KeyStore Password

Specifies the password used to decrypt the keystore.

Used with RSA signing.

KeyStore Signing Key Alias

Specifies the private key alias in the keystore used to sign the ID token.

Used with RSA signing.

Signature Key Password

Specifies the password of the private key alias used to sign the ID token.

Used with RSA signing.

Client secret

Specifies the secret shared between the client and the ID token generator used to sign the ID token.

Used with HMAC signing.

Issued Tokens Audience

Specifies the intended audience for the ID token. Populates the `aud` claim of the ID token.

The authorized party

Specifies the party to which the ID token is being issued. Populates the `azp` claim of the ID token.

Claim map

Specifies additional claim entries to be inserted into the ID token.

Specifies entries using the format *`claim_name=user_profile_attribute`*. When issuing the ID token, AM populates the claim value with the value of the attribute in the authenticated user's profile.

For example, suppose the Claim map property had an entry with the value `email=mail`. A generated OpenID Connect ID token for user Sam Carter would contain the claim `"email":"scarter@example.com"` if the `mail` attribute in Sam Carter's user profile had the value `scarter@example.com`.

Custom claim mapper class

Specifies the name of a custom claim mapper class. A claim mapper generates additional claims to be included in the OpenID Connect ID token.

The class must implement the `org.forgerock.openam.sts.tokengeneration.oidc.OpenIdConntectTokenClaimMapper` interface and must be bundled in the AM `.war` file.

Custom authn context mapper class

Specifies the name of a custom class that generates an `acr` claim in the OpenID Connect ID token. An `acr` claim indicates which authentication context class was satisfied by the authentication of the principal asserted in the OpenID Connect ID token. The `acr` claim is optional and is not included in the generated ID token by default.

For REST STS instances, the class must implement the `org.forgerock.openam.sts.rest.token.provider.oidc.OpenIdConnectTokenAuthnContextMapper` interface and must be bundled in the AM `.war` file.

For SOAP STS instances, the class must implement the `org.forgerock.openam.sts.soap.token.provider.oidc.SoapOpenIdConnectTokenAuthnContextMapper` interface and must be bundled into the SOAP STS deployment `.war` file.

Custom authn methods references mapper class

Specifies the name of a custom class that generates an `amr` claim in the OpenID Connect ID token. An `amr` claim indicates which authentication methods were used to authenticate the principal asserted in the OpenID Connect ID token. The `amr` claim is optional and is not included in the generated ID token by default.

For REST STS instances, the class must implement the `org.forgerock.openam.sts.rest.token.provider.oidc.OpenIdConnectTokenAuthMethodReferencesMapper` interface and must be bundled in the AM `.war` file.

For SOAP STS instances, the class must implement the `org.forgerock.openam.sts.soap.token.provider.oidc.SoapOpenIdConnectTokenAuthnMethodReferencesMapper` interface and must be bundled into the SOAP STS deployment `.war` file.

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.