

## Node development

---

These topics provide guidance and best practices for developing and maintaining authentication nodes in AM.



### **About authentication nodes**

Learn how authentication nodes define actions taken during authentication.



### **Prepare your environment**

Discover the prerequisites for building and customizing authentication nodes.



### **Translate nodes**

Internationalize the text in your nodes.



### **Build and install nodes**

Find out how to build and install authentication nodes for use in authentication trees.

For information on configuring and using authentication trees, see [Authentication nodes and trees](#).

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><sup>↗</sup>.

# Authentication nodes

Authentication trees (also referred to as *intelligent authentication*) provide fine-grained authentication by allowing multiple paths and decision points throughout the authentication flow.

Authentication trees are made up of *authentication nodes*, which define actions taken during authentication, similar to authentication modules in chains.

You can create complex yet customer-friendly authentication experiences by linking nodes together, creating loops, and nesting nodes within a tree, as follows:

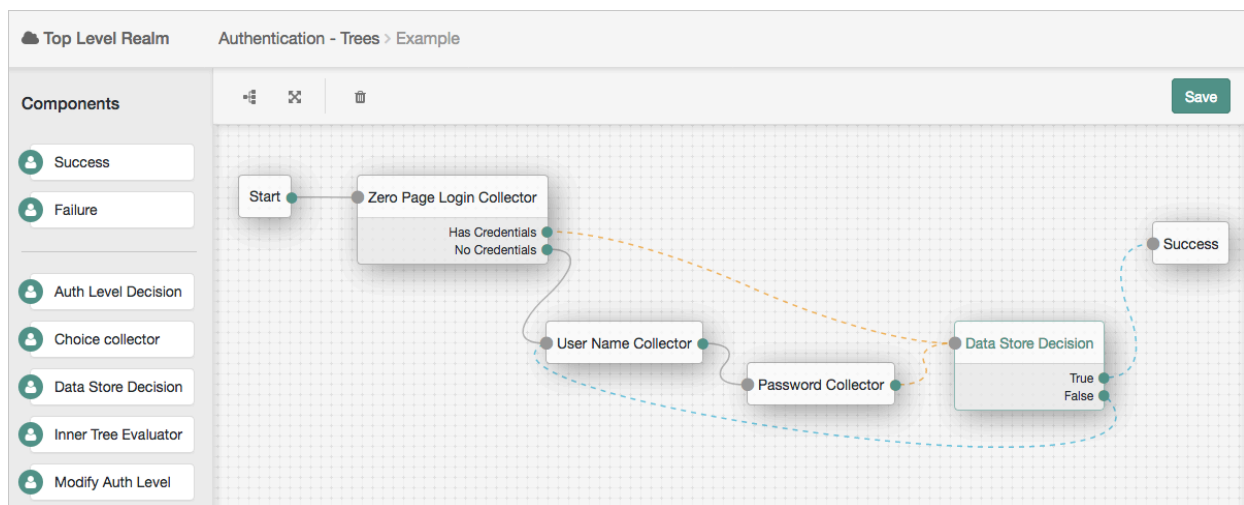


Figure 1. Example authentication tree

## Node types

Nodes are designed to have a single responsibility. Where appropriate, they should be loosely coupled with other nodes, enabling reuse in multiple situations.

For example, if a newly written node requires a username value, it should not collect it itself, but rely on another node, namely the Username Collector node.

There are two broad node types: collector nodes and decision nodes.

### Collector nodes

Collector nodes capture data from a user during the authentication process. This data is often captured by a *callback* that is rendered in the UI as a text field, drop-down list, or other form component.

Examples of collector nodes include the Username Collector node and Password Collector node.



LOG IN

Collector nodes can perform basic processing of the collected data, before making it available to subsequent nodes in the authentication tree.

The Choice Collector node provides a drop-down list, populated with options defined when the tree is created, or edited.



Please choose an option

✓ Yes  
No  
Maybe

LOG IN

Not all collector nodes use callbacks. For example, the Zero Page Login Collector node retrieves a username and password value from the request headers, if present.

### *Decision nodes*

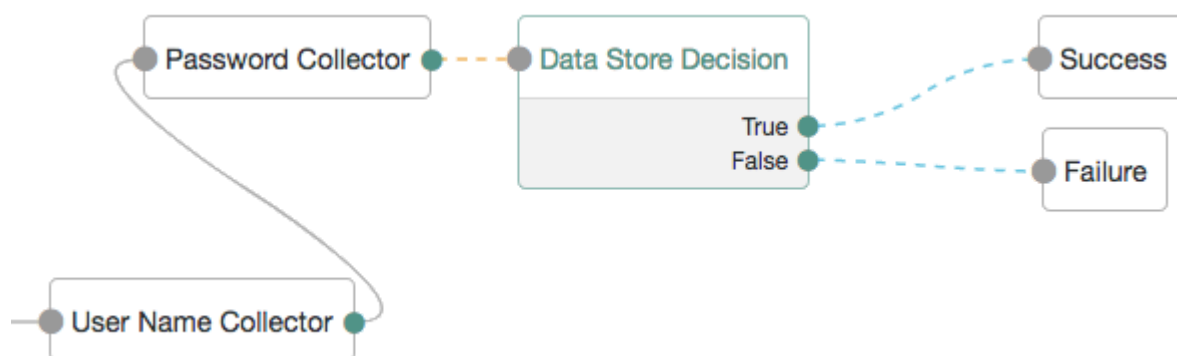
Decision nodes do the following:

- Retrieve the state produced by one or more nodes.
- Perform some processing on that state.
- Optionally, store some derived information in the shared state.
- Provide one or more outcomes, depending on the result.

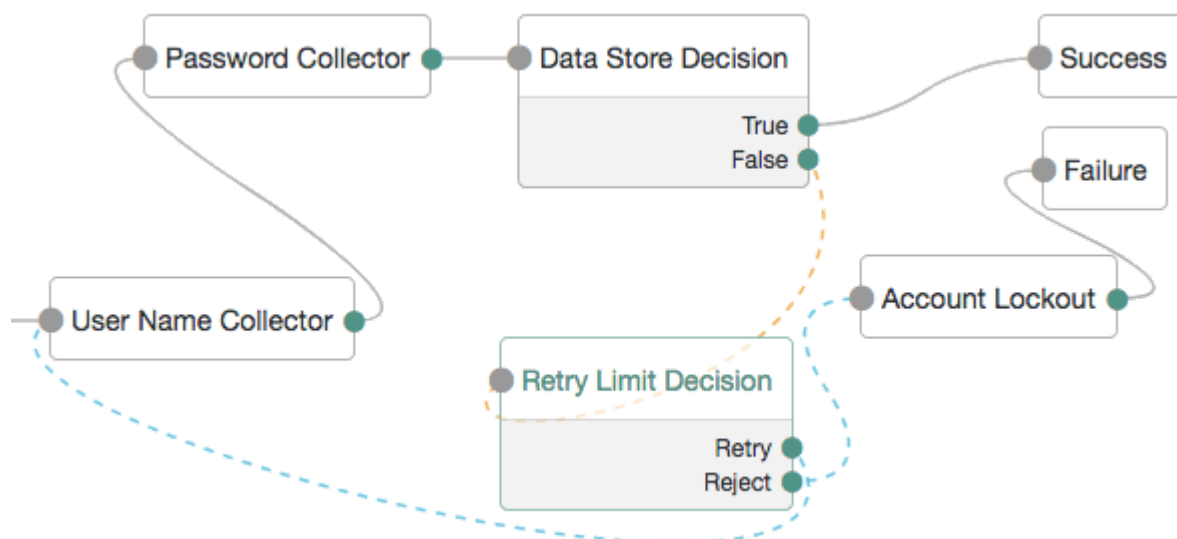
The simplest decision node returns a boolean outcome - `true` , or `false` .

Complex nodes may have additional outcomes. For example, the LDAP Decision node provides the additional outcomes *Locked*, *Expired*, and *Cancelled*. The tree administrator decides what to do with each outcome; for example, the `True` outcome is often routed to a *Success* node, or to additional nodes for further authentication.

In the following example tree, two collector nodes are connected before a Data Store Decision node. The node then uses the credentials to authenticate the user against the identity stores configured for the realm. In this instance, an unsuccessful login attempt leads directly to failure; the user must restart the process from scratch.



A more user-friendly approach might route unsuccessful attempts to a Retry Limit Decision node. In the following example, unsuccessful authentication attempts at the Data Store Decision node stage are routed into a Retry Limit Decision node. Depending on how many retries have been configured, the node either retries or rejects the new login attempt. Rejected attempts lead to a locked account.



***Nodes can have prerequisite stages***

Some Decision nodes are only applicable when used in conjunction with other nodes. For example, the Persistent Cookie Decision node looks for a persistent cookie that has been set in the request, typically by the Set Persistent Cookie node. The OTP Collector Decision node, which is both a collector and a decision node, only works when used in conjunction with a one-time password generated by a HOTP Generator node.

## Compare nodes and modules

Authentication modules contain multiple states, with the associated inputs and outputs of each state resulting in either callbacks returned to the user, or state changes inside either AM or at a third party service.

States within a module either collect input from the user or process it. For example, a module can collect the username and the password from the user, then authenticate the user against the data store. When finished, the module decides whether to return a boolean success or failure flag.

The outcome of an authentication module can only be success or failure. Any branching or looping must be handled within the module. An authentication mechanism is implemented in full within a single module, rather than across multiple modules. Therefore, authentication modules can become large - handling multiple steps within the flow of an authentication journey.

Authentication nodes, however, can have an arbitrary number of outcomes that do not need to represent success or failure. Branching and looping are handled by connecting nodes within the tree and are fully controlled by the tree administrator, rather than the node developer. Nodes are often considerably smaller in terms of code size, and are responsible for handling a single step within the authentication flow. For example, an individual node could capture user input, another could make a decision based on available state, and another could invoke an external API.

Nodes expose this approach to the tree administrator. Unlike modules, where the journey through the module's states is defined by the module's developer, a journey through a collection of nodes may be different for each user.

Node developers should be aware of the expectations for a node to deliver a limited amount of specific functionality, which tree administrators can connect together in a variety of ways.

***Key differentiators:***

- Nodes are responsible for a single step of the authentication flow. Modules are responsible for an entire authentication mechanism.

- Tree administrators control the branching, looping, and sequencing of steps by linking nodes together. Module developers set these state transitions within the module itself.
- Nodes are stateless; instances of node objects are not retained between HTTP requests, and all state captured must be saved to the authentication session's shared state. Modules store state within the module object.
- Node configuration is handled with annotations. Modules use XML.
- Nodes are easier to test due to their smaller code size and their specific functionality.

## Convert modules to nodes

The ease of transitioning from a module to a node depends on the amount of functionality provided in the module. As nodes are more fine-grained than modules, split the functionality of the module into individual nodes which can then work together to provide functionality similar to the module, but in a more flexible manner.

An example of this approach was applied to the one-time password nodes, which were developed from the HOTP authentication module. The module performs the following duties:

1. Generates one-time passwords.
2. Sends one-time passwords by using SMS messaging.
3. Sends one-time passwords by using SMTP.
4. Collects and verifies the one-time password.

The four distinct functions are encapsulated into separate nodes, allowing greater control over the one-time password process.

For example, a tree administrator who is only interested in sending one-time passwords by using SMS messages can omit the SMTP node. Separating out the decision functionality means that it can be combined with another decision node, or simply routed to an alternative authentication process.

Some authentication modules delegate their functionality to utility classes in AM, which simplifies the process of creating a similarly functioning node.

For example, the LDAP authentication module and [LDAP Decision node](#) share the `LDAPAuthUtils` class for LDAP authentication. In cases where such utility classes do not exist, consider extracting the common functionality used by the module into such a class, so that it can be more easily used by nodes. For information on sharing configuration, see [Share configuration between nodes](#).

## Restrict a node's functionality

To determine the functionality of a node, reduce the node's responsibility to its core purpose, while ensuring it performs sufficient tasks to be useful as a step in an authentication journey.

Before you create a set of nodes, assess the level of granularity the nodes should produce. For example, a customer's environment may require a series of utility nodes which, on their own, do not perform authentication actions, but have multiple use cases in many authentication journeys. In this case, you can create nodes that take values from the shared state and save it to the user's profile.

Individual nodes can respond to a variety of inputs and outputs, and return different sets of callbacks to the user without leaving the node. (This is similar to the *state* mechanism used by modules.)

The following guidelines help a node developer determine the best point at which to split a node into multiple instances:

- If a node's process method takes input from the user, and immediately processes it.

Consider splitting the functionality over two nodes. A collector node returns callbacks to the user, and stores the response in the shared state. A decision node uses the inputs collected so far in the tree to determine the next course of action.

A node that takes input from the user and makes a decision should only be designed as a single node if there is no possible additional use for the data gathered, other than making that specific decision.

- If a processing stage in a node is duplicated in other nodes.

In this case, take the repeating stage out and place it in its own node. Connect this node appropriately to each of the other nodes.

If multiple nodes contain the same step in processing, such as returning a set of callbacks to ask the user for a set of data before processing it in different ways, the common functionality should be pulled out into its own node.

- If a single function within the node has obvious use cases in other authentication journeys.

In this case, the functionality should be written into a single, reusable node. For example, in multi-factor authentication, a mechanism for reporting a lost device is applicable to many node types, such as mobile push, OATH, and others.

## Prepare for development

---

This page explains the prerequisites for building custom authentication nodes, and shows how to use either a Maven archetype, or the samples provided with AM, to set up a project for building nodes.

For information about customizing post-authentication hooks for a tree, see [Create post-authentication hooks for trees](#).

## Prepare an environment for building custom authentication nodes

1. Make sure your Backstage account is part of a subscription:
  - In a browser, go to the [Backstage](#) website and sign on or register for an account.
  - Confirm or request your account is added to a subscription. Learn more in [Getting access to product support](#) in the *Knowledge Base*.
2. Install Apache Maven 3.2.5 or later, and Oracle JDK or OpenJDK version 11 or later.

### TIP

To verify the installed versions, run the `mvn --version` command:

```
$ mvn --version
Maven home: /usr/local/Cellar/maven/3.6.0/libexec
Java version: 11.0.4, vendor: AdoptOpenJDK, runtime:
/Library/Java/JavaVirtualMachines/adoptopenjdk-11.jdk/Contents/Home
Default locale: en_US, platform encoding: UTF-8
OS name: "mac os x", version: "10.16", arch: "x86_64", family: "mac"
```

3. Configure Maven to be able to access the proprietary repositories by adding your Backstage credentials to the Maven `settings.xml` file. Learn more in [How do I access the proprietary protected Maven repositories?](#).

If you want to use the archetype to create a project for custom authentication nodes, you also need access to the `forgerock-private-releases` repository. Ensure your `settings.xml` file contains a profile similar to the following:

```
<profiles>
  <profile>
    <id>forgerock</id>
    <repositories>
      <repository>
        <id>forgerock-private-releases</id>

        <url>https://maven.forgerock.org:443/artifactory/private-releases</url>
        <releases>
          <enabled>true</enabled>
          <checksumPolicy>fail</checksumPolicy>
        </releases>
      </repository>
    </repositories>
  </profile>
</profiles>
```



```

        <snapshots>
            <enabled>false</enabled>
            <checksumPolicy>warn</checksumPolicy>
        </snapshots>
    </repository>
</repositories>
</profile>
</profiles>
<activeProfiles>
    <activeProfile>forgerock</activeProfile>
</activeProfiles>

```

## Set up a Maven project to build custom authentication nodes

ForgeRock provides a Maven archetype that creates a starter project, suitable for building an authentication node. You can also download the projects used to build the authentication nodes included with AM and modify those to match your requirements.

### NOTE

Complete the steps in Prepare an environment for building custom authentication nodes before proceeding.

Complete either of the following steps to set up or download a Maven project to build custom authentication nodes:

1. To use the ForgeRock `auth-tree-node-archetype` archetype to generate a starter Maven project:

- In a terminal window, go to a folder where you'll create the new Maven project. For example:

```
$ cd ~/Repositories
```

- Run the `mvn archetype:generate` command, providing the following information:

#### ***groupId***

A domain name that you control, used for identifying the project.

#### ***artifactId***

The name of the JAR created by the project, without version information.  
Also the name of the folder created to store the project.

#### ***version***

The version assigned to the project.

#### ***package***

The package name in which your custom authentication node classes are generated.

### ***authNodeName***

The name of the custom authentication node, also used in the generated README.md file and for class file names.

#### IMPORTANT

AM stores installed nodes with a reference generated from the node's class name. An installed node registered through a plugin is stored with the name returned as a result of calling `Class.getSimpleName()`.

AM doesn't protect installed node names. The most recently installed node with a specific name will overwrite any previous installation of that node (including the nodes that are provided with AM by default). You must therefore choose a unique name for your custom node, and make sure the name isn't already used for an existing node.

For example:

```
$ mvn archetype:generate \
  -DgroupId=com.example \
  -DartifactId=customAuthNode \
  -Dversion=1.0.0-SNAPSHOT \
  -Dpackage=com.example.customAuthNode \
  -DauthNodeName=myCustomAuthNode \
  -DarchetypeGroupId=org.forgerock.am \
  -DarchetypeArtifactId=auth-tree-node-archetype \
  -DarchetypeVersion=7.2.0 \
  -DinteractiveMode=false
```

```
[INFO] Project created from Archetype in dir:
/Users/ForgeRock/Repositories/customAuthNode
```

```
[INFO] -----
```

```
[INFO] BUILD SUCCESS
```

```
[INFO] -----
```

```
[INFO] Total time: 1.397 s
```

```
[INFO] Finished at: 2018-01-18T15:45:06+00:00
```

```
[INFO] Final Memory: 16M/491M
```

```
[INFO] -----
```

```
-----
```

A new custom authentication node project is created; for example, in the /Users/ForgeRock/Repositories/customAuthNode folder.

### ▼ Example

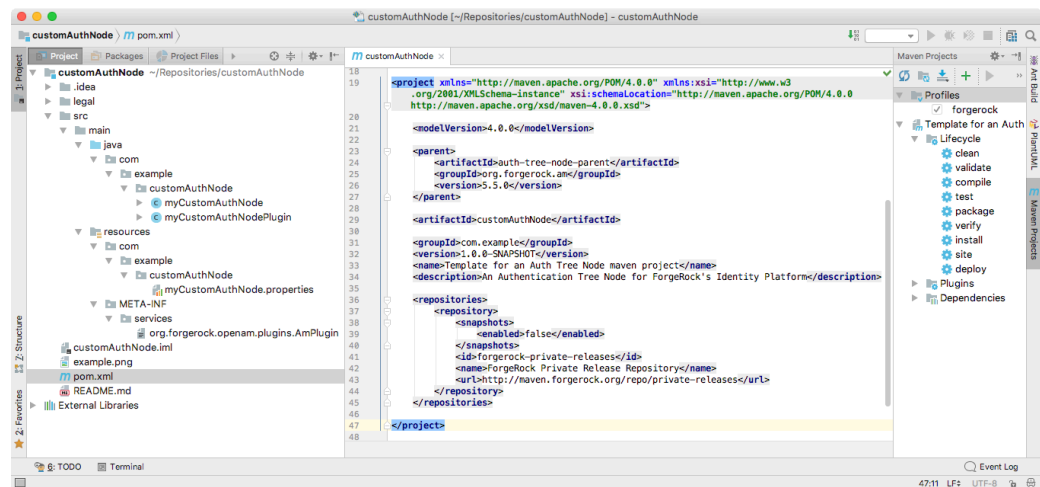


Figure 2. Node project created by using the archetype

2. To download the project containing the default AM authentication nodes from the am-external repository:

- Clone the am-external repository:

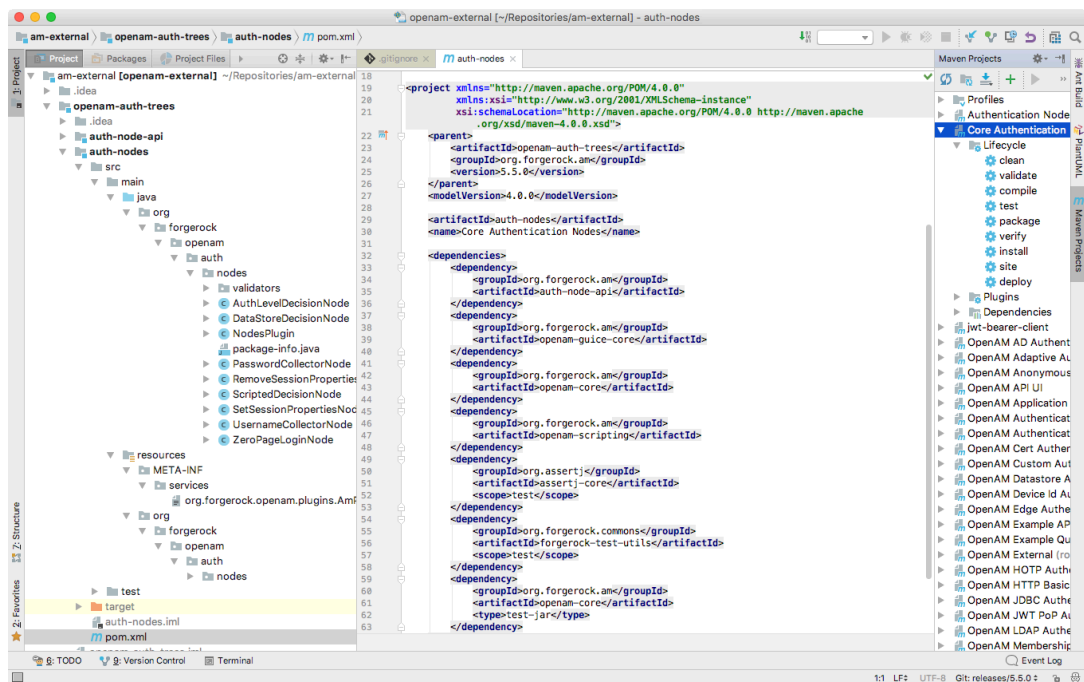
```
$ git clone https://github.com/ForgeRock/am-external.git
```

- Check out the release/7.2.0 branch:

```
$ cd am-external
$ git checkout releases/7.2.0
```

The AM authentication nodes project is located in the am-external/openam-auth-trees/auth-nodes/ folder.

### ▼ Example



*Figure 3. Node Project Cloned from ForgeRock*

## Tips for custom authentication node projects

When you configure a project for creating custom nodes, consider the following points:

- Your node may be deployed into a different AM version to that which you compiled against.


ForgeRock endeavours to make nodes from previous product versions binary compatible with subsequent product versions, so a node built against AM 6 APIs may be deployed in an AM 7.2.0 instance.

- Other custom nodes may depend on your node, which may be being built against a different version of the AM APIs.
- Other custom nodes, or AM itself, may be using the same libraries as your node; for example, Guava or Apache Commons, and so on. This may cause version conflicts.

To help protect against some of these issues, consider the following recommendations:

- Mark all ForgeRock product dependencies as `provided` in your build system configuration.
- Repackage all non-internal, non-ForgeRock dependencies inside your own `.jar` file. Repackaged dependencies will not clash with a different version of the same library from another source.

**TIP**

If you are using Maven, use the [maven-shade-plugin](#)  to repackage dependencies.

## ▼ [Files contained in the Maven project](#)

### *pom.xml*

Apache Maven project file for the custom authentication node.

This file specifies how to build the custom authentication node, and also specifies its dependencies on AM components.

The following is an example `pom.xml` file from a node project:

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>example-node-plugin</artifactId>
  <version>1.0.0</version>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.forgerock.am</groupId>
        <artifactId>openam-bom</artifactId>
        <version>7.2.0-SNAPSHOT</version>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <dependencies>
    <dependency>
      <groupId>org.forgerock.am</groupId>
      <artifactId>auth-node-api</artifactId>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.forgerock.am</groupId>
      <artifactId>openam-annotations</artifactId>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>com.google.guava</groupId>
      <artifactId>guava</artifactId>
      <version>26.0-jre</version>
    </dependency>
  </dependencies>
</project>
```

```

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <configuration>

<shadedArtifactAttached>false</shadedArtifactAttached>

<createDependencyReducedPom>true</createDependencyReducedPom>
      <relocations>
        <relocation>
          <pattern>com.google</pattern>

<shadedPattern>com.example.node.guava</shadedPattern>
        </relocation>
      </relocations>
      <filters>
        <filter>
          <artifact>com.google.guava:guava</artifact>
          <excludes>
            <exclude>META-INF/**</exclude>
          </excludes>
        </filter>
      </filters>
      <transformers>
        <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
          <manifestEntries>
            <Import-
Package>javax.annotation;resolution:=optional,sun.misc;resolu
tion:=optional</Import-Package>
          </manifestEntries>
        </transformer>
      </transformers>
    </configuration>
  </plugin>
</plugins>
</build>
</project>

```

### ***authNodeName.java***

Core class for the custom authentication node. See [Node class](#).

### ***authNodeNamePlugin.java***

Plugin class for the custom authentication node. See [Plugin class](#).

### ***authNodeName.properties***

Properties file containing the localized strings displayed by the custom authentication node. See [Internationalization](#).

You must include a `nodeDescription` property in your node to ensure that it appears in the authentication tree designer. AM uses the `nodeDescription` property value as the name of your node.

The `authNodeName` reflects the name of your authentication node. For example, the ForgeRock `auth-tree-node-archetype` for Maven uses `myCustomAuthNode` as the `authNodeName`.

## Node class

---

In Java terms, an authentication node is a class that implements the `Node` interface, `org.forgerock.openam.auth.node.api.Node`.

The `Node` class may access and modify the persisted state that is shared between the nodes within a tree, and may request input by using callbacks. The class also defines the possible exit paths from the node.

The class is *annotated* with `org.forgerock.openam.auth.node.api.Node.Metadata`. The annotation has two main attributes - `configClass` and `outcomeProvider`. Typically, the `configClass` attribute is an inner interface in the node implementation class.

For simple use cases, the abstract implementations of the node interface, `org.forgerock.openam.auth.node.api.SingleOutcomeNode` and `org.forgerock.openam.auth.node.api.AbstractDecisionNode`, have their own outcome providers that can be used. For more complex use cases you can provide your own implementation.

The sections that follow describe the `Node` class.

## Annotation

The *annotation* specifies the outcome provider and configuration class. The outcome provider can use the default `SingleOutcomeNode` or `MultipleOutcomeNode`, or a custom `OutcomeProvider` can be created and referenced from the annotation.

See the [Choice Collector node](#) for an example of a custom outcome provider.

## Private constants

These are optional.

## Config

The *config* interface defines the configuration data for a node. A node cannot have state, but it can have configuration data.

Note that you do not need to provide the implementation class for the `Config` interface you define. AM will create these automatically, as required.

An example is the [Account Lockout node](#). The node can be configured to lock or unlock the users' account, based on its configuration.

Configuration is per-node. Different nodes of the same type in the same tree have their own configuration.

The `config` interface configures values using methods. To provide no default value to the tree administrator, provide the method's signature but not the implementation. To provide a default value to the tree administrator, mark the method as `default` and provide both a method and a value. For example:

```
public interface Config {

    //This will have no default value for the UI
    @Attribute(order = 10)
    String noDefaultAttribute();

    //This will default to the value LOCK.
    @Attribute(order = 20)
    default LockStatus lockAction() {
        return LockStatus.LOCK;
    }
}
```

The `Config` above would resemble the following in the tree designer view:



**My Custom Node**

**Node name**

My Custom Node

**No Default Value** ⓘ

Please provide a value

**Lock Action** ⓘ

Lock ▼

The `@Attribute` annotation is required. It can be configured with an order value, which determines the position of the attribute in the UI, and with validators, to validate the values being set.

In the example above, a custom enum called `LockStatus` is returned. The options are displayed to the user automatically.

## Constructor

Dependencies should be injected by using Guice as this makes it easier to unit test your node. For example, you should accept the `config` as a parameter.

You may also wish to obtain AM core classes, such as `CoreWrapper`, instances of third-party dependencies, or your own types.

```
@Inject
public AccountLockoutNode(CoreWrapper coreWrapper, @Assisted
Config config)
throws NodeProcessException {
    this.coreWrapper = coreWrapper;
    this.config = config;
}
```

## process method

The *process* method takes a `TreeContext` parameter, does some processing, and returns an *Action* object.

An action encapsulates changes to state and flow control. The `TreeContext` parameter is used to access the request, callbacks, shared state and other input.

The `process` method is where state is retrieved and stored. The returning `Action` can be a response of callback to the user, an update of state, or a choice of outcome.

The choice of outcome in a simple decision node would be `true` or `false`, resulting in the authentication tree flow moving from the current node to a node at the relevant connection.

## Private methods

These are optional.

## Custom outcome provider

This is optional.

## Example implementation

The following example is the `SetSessionPropertiesNode` class, taken from the [Set Session Properties node](#):

```
package org.forgerock.openam.auth.nodes;

import java.util.Map;

import javax.inject.Inject;

import org.forgerock.openam.annotations.sm.Attribute;
import org.forgerock.openam.auth.node.api.Action;
import org.forgerock.openam.auth.node.api.Node;
import org.forgerock.openam.auth.node.api.SingleOutcomeNode;
import org.forgerock.openam.auth.node.api.TreeContext;
import
org.forgerock.openam.auth.nodes.validators.SessionPropertyValidato
r;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.google.inject.assistedinject.Assisted;

/**
 * A node which contributes a configurable set of properties to be
 * added to the user's session, if/when it is created.
```

```

*/
@Node.Metadata(outcomeProvider =
SingleOutcomeNode.OutcomeProvider.class,
    configClass = SetSessionPropertiesNode.Config.class) ①
public class SetSessionPropertiesNode extends SingleOutcomeNode {
②

    /**
     * Configuration for the node.
     */
    public interface Config { ③
        /**
         * A map of property name to value.
         * @return a map of properties.
         */
        @Attribute(order = 100, validators =
SessionPropertyValidator.class)
        Map<String, String> properties();
    }

    private final Config config;
    private final Logger logger =
LoggerFactory.getLogger("amAuth");

    /**
     * Constructs a new SetSessionPropertiesNode instance.
     * @param config Node configuration.
     */
    @Inject ④
    public SetSessionPropertiesNode(@Assisted Config config) {
        this.config = config;
    }

    @Override
    public Action process(TreeContext context) { ⑤
        logger.debug("SetSessionPropertiesNode started");
        Action.ActionBuilder actionBuilder = goNext();
        config.properties().entrySet().forEach(property -> {
            actionBuilder.putSessionProperty(property.getKey(),
property.getValue());
            logger.debug("set session property {}", property);
        });
        return actionBuilder.build();
    }
}

```

- ① The `@Node.Metadata` annotation. See [Metadata annotation](#).
- ② Implementing the `Node` interface. See [Node interface](#).
- ③ Implementing the `Config` interface. See [Config interface](#).
- ④ Injecting the `Node` instance. See [Inject objects into a node instance](#).
- ⑤ Creating an `Action` instance. See [Action class](#).

## Metadata annotation

---

The *annotation* specifies the outcome provider and config class, and optional config validator class.

### ***outcomeProvider***

The class name that the node uses to set up the possible outcomes.

The `SingleOutcomeNode` and `AbstractDecisionNode` base classes provide suitable outcome provider classes for those node types. You can create a custom outcome provider for other circumstances.

For example, the following is the custom outcome provider from the *LDAP Decision* node, which has `True`, `False`, `Locked`, `Cancelled`, and `Expired` exit paths:

```
/**
 * Defines the possible outcomes from this Ldap node.
 */
public static class LdapOutcomeProvider implements
OutcomeProvider {
    @Override
    public List<Outcome> getOutcomes(PreferredLocales locales,
JsonValue nodeAttributes) {
        ResourceBundle bundle =
locales.getBundleInPreferredLocale(LdapDecisionNode.BUNDLE,
LdapOutcomeProvider.class.getClassLoader());
        return ImmutableList.of(
            new Outcome(LdapOutcome.TRUE.name(),
bundle.getString("trueOutcome")),
            new Outcome(LdapOutcome.FALSE.name(),
bundle.getString("falseOutcome")),
            new Outcome(LdapOutcome.LOCKED.name(),
bundle.getString("lockedOutcome")),
            new Outcome(LdapOutcome.CANCELLED.name(),
bundle.getString("cancelledOutcome")),
            new Outcome(LdapOutcome.EXPIRED.name(),
bundle.getString("expiredOutcome")));
    }
}
```

```
}  
}
```

### ***configClass***

The class name that contains the configuration of any attributes requested by the node when using it as part of a tree.

For more information, See [Config interface](#).

### ***configValidator***

An optional class name used to validate the provided configuration.






### ***tags***

An optional list of tags which help to categorize the node when using the tree designer view.

Tags are made up of one or more text strings that let users find the node more easily when designing trees. For example, you could include common pseudonyms for the functionality the node provides, such as `mfa` for a node that provides multi-factor authentication functionality.


The tree designer view organizes nodes into a number of categories, based on the presence of certain tag values, as described in the table below:

#### ***Authentication Node Tag Categories***

Icon	Category	Tag	Example Nodes
	Basic Authentication	"basic authentication"	<a href="#">Data Store Decision node</a> <a href="#">Username Collector node</a>
	MFA	"mfa"	<a href="#">Push Sender node</a> <a href="#">WebAuthn Authentication node</a>
	Risk	"risk"	<a href="#">Account Lockout node</a> <a href="#">CAPTCHA node</a>
	Behavioral	"behavioral"	N/A
	Contextual	"contextual"	<a href="#">Cookie Presence Decision node</a> <a href="#">Set Persistent Cookie node</a>

Icon	Category	Tag	Example Nodes
	Federation	"federation"	<a href="#">OAuth 2.0 node</a> <a href="#">OpenID Connect node</a> <a href="#">Social Facebook node</a> <a href="#">Social Google node</a>
	Identity Management	"identity management"	<a href="#">Anonymous User Mapping node</a>
	Utilities	"utilities"	<a href="#">Choice Collector node</a> <a href="#">Polling Wait node</a> <a href="#">Scripted Decision node</a>

#### NOTE

Nodes which are not tagged with one of the previous tags appear in an  **Uncategorized** section.

For example, the following is the `@Node.Metadata` annotation from the [Set Session Properties node](#):

```
@Node.Metadata(outcomeProvider =
AbstractSocialAuthLoginNode.SocialAuthOutcomeProvider.class,
    configClass = SocialFacebookNode.FacebookOAuth2Config.class,
    tags = {"social", "federation"})
```

For more information on the `@Node.Metadata` annotation, see the [Node.Metadata](#) annotation type in the *AM Public API Javadoc*.

## Node interface

The code for an authentication node must implement the `Node` interface.

AM provides base classes you can extend to implement the `Node` interface, depending on the type of custom authentication node you are creating. The available base classes are as follows:

### ***SingleOutcomeNode***

Used in nodes that only have a single exit path.

The [Modify Auth Level node](#) is an example of a node that uses the `SingleOutcomeNode` base class.

For more information, see the [SingleOutcomeNode](#) class in the *AMPublic API Javadoc*.

### ***AbstractDecisionNode***

Used in nodes that have a boolean-type exit path. For example, true or false, yes or no, or allow or deny.

The [Data Store Decision node](#) is an example of a node that uses the `AbstractDecisionNode` base class.

For more information, see the [AbstractDecisionNode](#) class in the *AMPublic API Javadoc*.

Implement the `Node` interface yourself if your custom node exit paths do not match the scenarios outlined above.

For more information, see the [Node](#) interface in the *AMPublic API Javadoc*.

## Config interface

---

The `Config` interface of a node contains the configuration values required by a particular node instance.

Note that you do not need to write a class that implements the interface you define, AM will create it automatically, as required.

Define the properties the node will use in the `Config` interface, by using the `@Attribute` annotation. The `@Attribute` annotation specifies the order of the properties in the tree designer view as well as providing a way to specify additional validators.

Example:

```
public interface Config {
    @Attribute(order = 1)
    String domain(); ①

    @Attribute(order = 2, validators =
RequiredValueValidator.class)
    boolean isVerificationRequired(); ②

    @Attribute(order = 3, validators =
RequiredValueValidator.class)
    @Password
```

```

char[] clientSecret(); ③

@Attribute(order = 4)
default YourCustomEnum action() {
    return YourCustomEnum.LockScreen; ④
};
}

```

- ① The `domain` attribute is a string-typed value which can be provided in the tree designer view by the tree administrator. It can be read in the `process` method by using a reference to the `config` interface; for example, `config.domain()`.
- ② A boolean attribute with an additional parameter, `validators`, containing a reference to a validation class. In this case, a value is required to be provided by the tree administrator.
- ③ Use the [Password](#) annotation to mask the input characters and encrypt the value of the attribute.
- ④ A custom enum attribute. This provides type safety and negates the misuse of Strings as generic type-unsafe value holders. The UI will correctly handle the enum and only let the tree administrator chose from the defined enum values.

The defined properties appear as configurable options in the tree designer view when adding a node of the relevant type. For example, the configuration for the [Scripted Decision node](#) appears as follows:

**Scripted Decision**

**Script**

My Auth Tree Script

The script to evaluate.

**outcomes**

x true x false x

Note that attribute names are used when localizing the node's text, see [Internationalization](#).

For more information, see the [Config](#) annotation type and the [Attribute](#) annotation type in the *AM Public API Javadoc*.

## Share configuration between nodes



You can share configuration between nodes that have common properties. For example, a number of your nodes may call out to an external service that requires a username, password, IP address, and port setting.

Rather than repeat the same configuration in each of these nodes, you can create a shared, auxiliary *service* to hold the common properties in one of your nodes, and reference that service from multiple other nodes.

The following sections explain how to create this auxiliary service and reference it in your nodes. Also covered is how to run more than one instance of an auxiliary service if required, and how to obtain the configuration from services built-in to AM.

### *Create a shared auxiliary service*

You can create a shared auxiliary service in the configuration interface defined as part of a node. Annotate the service with the

`org.forgerock.openam.annotations.sm.Config` annotation to describe how the service functions.

Specify the scope of the service, either `GLOBAL` or `REALM`, as shown below:

```
@Config(scope = Config.Scope.REALM)
public interface MyAuxService {
    @Attribute(order = 1)
    String serviceUrl();
}
```

You can also specify other features of the service, such as whether the service is a singleton in its scope, or if it can have multiple instances. For information about supporting multiple instances, see [Allow multiple instances of an auxiliary service](#).

### *Reference a shared auxiliary service instance*

To access the shared auxiliary service, add

`org.forgerock.openam.sm.AnnotatedServiceRegistry` to the `@Inject`-annotated constructor of the node.

Obtain the instance using the `getInstance` methods on that class, for example:

```
serviceRegistry.getRealmSingleton(MyAuxService.class, realm)
```

### *Reinstall a shared auxiliary service instance*

When developing a custom authentication node that references a shared auxiliary service, it can be useful for the node to be able to remove and reinstall the auxiliary

service during upgrade, so that any existing configuration is cleared.

In the `upgrade` function of your plugin class, use the following example code to remove and reinstall a service:

```
public void upgrade(String fromVersion) throws PluginException {
    SSOToken adminToken =
    AccessController.doPrivileged(AdminTokenAction.getInstance());
    if (fromVersion.equals(PluginTools.DEVELOPMENT_VERSION)) {
        ServiceManager sm = new ServiceManager(adminToken);
        if (sm.getServiceNames().contains("MyAuxService")) {
            sm.removeService("MyAuxService", "1.0");
        }
        pluginTools.install(MyAuxService.class);
    }
}
```

For more information on upgrading custom authentication nodes, see [Upgrade nodes and change node configuration](#).

### *Allow multiple instances of an auxiliary service*

To enable configuration of multiple instances of the auxiliary service in either the same realm or at a global level, set the `collection` attribute to `true` in the `Config` annotation.

You can present the names of the instances of the service as a drop-down menu to the tree administrator.

To be able to present the names, make sure the service instance exposes its `id`, as follows:

```
@Config(scope = Config.Scope.REALM, collection = true)
public interface MyAuxService {
    @Id
    String id();

    @Attribute(order = 1)
    String serviceUrl();
}
```

Change the nodes that will be using a service instance to store the `id` it uses, and implement `choiceValuesClass` as shown below:

```

public class MyCustomNode implements Node {
    public interface Config {
        @Attribute(order = 1, choiceValuesClass =
ExternalServiceValues.class)
        String serviceId();
    }

    public static class ExternalServiceValues extends ChoiceValues
{

        @Override
        public Map<String, String> getChoiceValues() {
            return getChoiceValues(null);
        }

        @Override
        public Map<String, String> getChoiceValues(Map envParams) {
            String realmName = "/";
            if (envParams != null) {
                realmName = (String)
envParams.getDefault(Constants.ORGANIZATION_NAME, "/");
            }
            try {
                return
InjectorHolder.getInstance(AnnotatedServiceRegistry.class)
                    .getRealmInstances(MyAuxService.class,
Realms.of(realmName))
                    .stream()
                    .collect(Collectors.toMap(MyAuxService::id,
MyAuxService::id));
            } catch (SSOException | SMSEException |
RealmLookupException e) {
                LoggerFactory.getLogger("amAuth").error("Couldn't
load realm {}", realmName, e);
                throw new IllegalStateException("Couldn't load
realm that was passed", e);
            }
        }
    }
    // ...
}

```

*Get configuration of built-in services*

You can obtain configuration from services built-in to AM. For example, you might want to access the Email Service configuration to obtain the SMTP settings for the realm.

AM services are defined by two methods:

1. Most services are defined by using an annotated interface.
2. Legacy services that are defined by using an XML file.

See the following sections for information on obtaining the configuration from services defined with either of the two methods.

### *Get configuration from an annotated service*

To obtain the configuration from a service that uses an annotated interface, add `org.forgerock.openam.sm.AnnotatedServiceRegistry` to your Guice constructor. If the configuration is realm-based, include the realm in the constructor, as follows:

```
public class MyCustomNode extends SingleOutcomeNode {

    private final AnnotatedServiceRegistry serviceRegistry;
    private final Realm realm;

    @Inject
    public MyCustomNode(@Assisted Realm realm,
        AnnotatedServiceRegistry serviceRegistry) {
        this.realm = realm;
        this.serviceRegistry = serviceRegistry;
    }
    // ...
}
```

Obtain an instance of the service using one of the get methods of `AnnotatedServiceRegistry` in the constructor.

If the calls you make depend on input from elsewhere in the tree you can add `AnnotatedServiceRegistry` to the `process` method. Note that the following example assumes that a previous node has stored the ID of the AM service to use in shared state:

```
public Action process(TreeContext context) throws
NodeProcessException {
    String serviceId = context.getState.get("myAuxServiceId");
    MyAuxService instance =
serviceRegistry.getRealmInstance(MyAuxService.class, realm,
serviceId);
```

```
// ...  
}
```

### Get configuration from a legacy service

To obtain an instance of the configuration from a legacy service, use the APIs in the `com.sun.identity.sm` package.

For example, to obtain the configuration values from a *realm* instance of a service, use `ServiceConfigManager` as follows:

```
ServiceConfigManager scm = new  
ServiceConfigManager("legacyServiceName", token);  
ServiceConfig sc = scm.getOrganizationConfig(realm.asPath(),  
null);  
final Map<String, Set<String>> configMap = sc.getAttributes();
```

However, to obtain the configuration values from a *global* instance of a service, use `ServiceSchemaManager` as follows:

```
ServiceSchemaManager ssm = new  
ServiceSchemaManager("legacyServiceName", getAdminToken());  
Map<String, Set<String>> configMap =  
ssm.getGlobalSchema().getAttributeDefaults();
```

## Inject objects into a node instance

A node instance is constructed every time that node is reached in a tree, and is discarded as soon as it has been used to process the state once.

This model is different to authentication modules, which are instantiated once for each end-user authentication process, and then all authentication interactions for the life of the authentication process address the same instance in the same JVM.

Modules can store state in the module instance. However, state stored in a node will be lost when the node's process method is complete. To make state available for other nodes in the tree, nodes must either return the state to the user or store it in the shared state.

AM uses Google's *Guice* dependency injection framework for authentication nodes. AM uses Guice to manage most of its object life-cycles. You can use just-in-time bindings from the constructor to inject an object from Guice.

The following node-specific instances are available from Guice:

**@Assisted Realm**

The realm that the node is in.

**@Assisted UUID**

The unique ID of the node instance.

**@Assisted AuthTree**

The tree that this node is being processed as part of.

**<T> @Assisted T**

The configuration object that is an instance of the interface specified in the `configClass` metadata parameter.

**TIP**

Any other objects in AM that are managed by Guice can also be obtained from within the constructor.

The following example is the injection used by the [Account Lockout node](#):

```
@Inject
public AccountLockoutNode(CoreWrapper coreWrapper, @Assisted
    Config config)
    throws NodeProcessException {
    this.coreWrapper = coreWrapper;
    this.config = config;
}
```

For more information, see the [Inject](#) annotation type and the [Assisted](#) annotation type in the *Google Guice Javadoc*.

## Use a cache

You can use Guice injection to cache information in a node by annotating the object that contains the cache with the `@Singleton` annotation, for example:

```
@Node.Metadata(
    outcomeProvider = SingleOutcomeNode.OutcomeProvider.class,
    configClass = MyCustomNode.Config.class)
public class MyCustomNode extends SingleOutcomeNode {

    public interface Config {
        String url();
    }
}
```

```

    private final Config config;
    private final MyCustomNodeCache cache;

    @Inject
    public MyCustomNode(@Assisted Config config, MyCustomNodeCache
cache) {
        this.config = config;
        this.cache = cache;
    }

    @Override
    public Action process(TreeContext context) {
        CachedThing thing = cache.getThing(config.url());
        // implement node logic here
    }
}

@Singleton
class MyCustomNodeCache {
    private final LoadingCache<String, CachedThing> cache =
        CacheBuilder.newBuilder()
            .build(CacheLoader.from(url -> read(url)));

    public CachedThing get(String url) {
        return cache.get(url);
    }

    private CachedThing read(String url) {
        // Access resource and construct
    }
}

```

## Custom Guice bindings

If just-in-time bindings are not sufficient for your use case, you can add your own Guice module into the injector configuration by implementing your own `com.google.inject.Module` and registering it using the service loader mechanism. For example:

```

// com/example/MyCustomModule.java
public class MyCustomModule extends AbstractModule {
    @Override
    protected void configure() {

```

```

        bind(Thing.class).to(MyThing.class);
        // and so on
    }
}

```

```

// META-INF/services/com.google.inject.Module
// See https://docs.oracle.com/javase/tutorial/ext/basics/spi.html
com.example.MyCustomModule

```

The `MyCustomModule` object will then be automatically configured as part of the injector creation.

## Action class

The `Node` class returns an `Action` instance from its `process()` method.

The `Action` class encapsulates changes to authentication tree state and flow control.

For example, the following implementation demonstrates an authentication level decision:

```

@Override
public Action process(TreeContext context) throws
NodeProcessException {
    NodeState state = context.getStateFor(this);
    if (!state.isDefined(AUTH_LEVEL)) {
        throw new NodeProcessException("Auth level is required");
    }
    JsonValue authLevel = state.get(AUTH_LEVEL);
    boolean authLevelSufficient =
        !authLevel.isNull()
        && authLevel.asInteger() >= config.authLevelRequirement();
    return goTo(authLevelSufficient).build();
}

```

For more information, refer to the [Action](#) class in the *AM Public API Javadoc*.

## Action fields and methods

The `Action` class uses the following fields:



Fields	Description
callbacks	A list of the callbacks requested by the node. This list may be null.
errorMessage	<p>A custom error message string included in the response JSON if the authentication tree reaches the Failure authentication node.</p> <p>Each node in a tree can replace or update the error message string as the user traverses through the authentication tree.</p> <p>If required, your custom node or custom UI must localize the error string.</p>
lockoutMessage	<p>A custom lockout message string included in the response JSON when the user is locked out.</p> <p>If required, your custom node or custom UI must localize the error string.</p>
outcome	The result of the node.
returnProperties	<p>A map of properties returned to the client.</p> <p>Use <code>withReturnProperty(String key, Object value)</code> to add a property to the map.</p>
sessionHooks	The list of classes implementing the TreeHook interface that run after a successful login.
sessionProperties	<p>A map of properties added to the final session if the authentication tree completes successfully.</p> <p>Use <code>putSessionProperty(String key, String value)</code> and <code>removeSessionProperty(String key)</code> to add or remove entries from the map.</p>
sharedState and transientState	<p>State that AM shares between nodes through the tree context—the properties set so far by nodes in the tree.</p> <p>Refer to Store values in shared tree state.</p>
suspensionHandler	The handler class to invoke when the authentication framework suspends authentication.
webhooks	The list of webhooks that run after logout.

The `Action` class provides the following methods:

Methods	Description
goTo	<p>Specify the exit path to take, and move on to the next node in the tree.</p> <p>For example:</p> <pre>return goTo(false).build();</pre>
send	<p>Send the specified callbacks to the user for them to interact with.</p> <p>For example, the <a href="#">Username Collector node</a> uses the following code to send the <code>NameCallback</code> callback to the user to request the <code>USERNAME</code> value:</p> <pre>return send(new NameCallback(bundle.getString("callback.use rname"))).build();</pre>
sendingCallbacks	<p>Returns true if the action is a request for input from the user.</p>
suspend	<p>Suspends the authentication tree, and lets the user resume it from the point it was suspended.</p> <p>For example, the following call is taken from the <a href="#">Email Suspend node</a>:</p> <pre>return suspend(resumeURI -&gt; createSuspendOutcome(context, resumeURI, recipient, templateObject)).build();</pre> <p>Use the <a href="#">SuspensionHandler</a> interface for handling the suspension request.</p>

The inner class `ActionBuilder` provides the following methods for constructing the `Action` object and setting action-related properties:

Methods	Description
addNodeType	<p>Add a node type to the session properties and shared state. Replace any existing shared state with the specified <code>TreeContext</code>'s shared state.</p>

Methods	Description
addSessionHook addSessionHooks	Add one or more session hook classes for AM to run after a successful login.
addWebhook addWebhooks	Add one or more webhook names to the list of webhooks.
build	Creates and returns an Action instance providing the mandatory fields are set.
putSessionProperty	Add a new session property.
removeSessionProperty	Remove the specified session property.
replaceSharedState	Replace the current shared state with the specified shared state.
replaceTransientState	Replace the current transient state with the specified transient state.
withDescription	Set a description for this action.
withErrorMessage	Set a custom message for when the authentication tree reaches the failure node.
withHeader	Set a header for this action.
withIdentifiedIdentity	<p>Add an identity, authenticated or not, that is confirmed to exist in an identity store. Specify the username and identity type or an <code>AMIdentity</code> object.</p> <p>Use this method to record the type of identified user. If the advanced server property, <code>org.forgerock.am.auth.trees.authenticate.identified.identity</code> is set to true, AM uses the stored identified identities to decide which user to log in.</p> <p>This lets the authentication tree engine correctly resolve identities that have the same username.</p> <p>For more information, refer to <a href="#">advanced server properties</a>.</p>
withLockoutMessage	Set a custom message for when the user is locked out.
withReturnProperty	Add a property to the list that is returned to the client.

Methods	Description
<code>withStage</code>	Set a stage name to return to the client to aid the rendering of the UI. The property is only sent if the node also sends callbacks.
<code>withUniversalId</code>	<i>Deprecated.</i>  Use <code>withIdentifiedIdentity</code> instead.

## Store values in shared tree state

Tree state exists for the lifetime of the *authentication session*. Once tree execution is complete, the authentication session is terminated and a *user session* is created. The purpose of tree state is to hold state between the nodes.

A good example is the Username Collector node, which gets the user name from the user and stores it the shared tree state. Later, the Data Store Decision node can pull this value from shared tree state and use it to authenticate the user.

Authentication sessions when using chains and modules are *stateful* - the AM server that starts the authentication flow must not change. A load balancer cookie is set on the responses to the user to ensure the same AM server is used.

In contrast, authentication trees can be made *stateless*, so that any AM instance in a deployment can continue the authentication session.

For more information on configuring sessions, see Sessions.

## Store values in a tree's node states

Always store the authentication state in the `NodeState` object that AM lets you access from the `TreeContext` object passed to the node's `action()` method. AM ensures that the node state is made available to downstream nodes:

- Store non-sensitive information with the `NodeState.putShared()` method.
- Store sensitive information, such as passwords, with the `NodeState.putTransient()` method.

AM encrypts the transient state with the key that has the `am.authn.trees.transientstate.encryption` secret ID. Downstream consumers, such as IDM user self-service nodes, must have the same key to decrypt and read it.

To ensure that the authentication flow is not bloated with calls to encrypt/decrypt data, and to ensure the authentication session size stays small, limit what you store

with `putTransient()`. This is especially true when the realm is configured for client-side authentication sessions.

### *Get and set values stored in tree state*

Internally, AM distinguishes the following node state data:

- Shared state, where nodes store non-sensitive information that needs to be available during the authentication flow.

You store this with the `NodeState.putShared()` method.

- Transient state, where nodes store sensitive information that AM encrypts on round trips to the client.

You store this with the `NodeState.putTransient()` method.

- Secure state, where nodes store decrypted transient state.

For details, see [NodeState](#).

### *Set values in the tree state*

To set node state values, get the `NodeState` using the `TreeContext.getStateFor(Node node)` method. Then, use the `NodeState.putShared()` and `NodeState.putTransient()` methods as described above.

For example:

```
// Setting values in NodeState
public Action process(TreeContext context) {
    String username;
    String password;
    // ...
    NodeState state = context.getStateFor(this);
    state.putShared(USERNAME, username);      // Non-sensitive
information
    state.putTransient(PASSWORD, password);   // Sensitive
information
    if (!state.isDefined(OPTIONAL_NUMERIC)) { // Check before
updating
        state.putShared(OPTIONAL_NUMERIC, 42);
    }
    goToNext().build();
}
```

### Get values in the tree state

To read node state values, use the `NodeState.isDefined(String key)` and `NodeState.get(String key)` methods.

For example:

```
// Getting values from NodeState
public Action process(TreeContext context) {
    NodeState state = context.getStateFor(this);
    String username;
    if (state.isDefined(USERNAME)) {
        username = state.get(USERNAME);
    } else {
        throw new NodeProcessException("Username is required");
    }
    // ...
    goToNext().build();
}
```

The `get(String key)` method retrieves the state for the key from `NodeState` states in the following order:

1. transient
2. secure
3. shared

For example, if the same property is stored in the transient and shared states, the method returns the value of the property in the transient state first.

## Access an identity's profile

AM allows a node to read and write data to and from an identity's profile. This is useful if a node needs to store information more permanently than when using either the authentication trees' `NodeState`, or the identity's session.

### WARNING

Any node which reads or writes to an identity's profile must only occur in a tree after the identity has been verified. For example, as the final step in a tree, or directly after a Data Store Decision node.

To store a verified identity in the authentication session, call `ActionBuilder.withIdentifiedIdentity()`. This ensures identities with the same username are correctly resolved.

## *Read an identity's profile*

Use the `IdUtils` static class:

```
AMIdentity id = IdUtils.getIdentity(username, realm);
```

### **TIP**

Wrap the method call in an instantiable class to ease testing.

If AM is configured to search for the identity's profile using a different search attribute to the default, provide the attributes as a third argument to the method.

To obtain the attributes you could request them in the configuration of the node, or obtain them from the realm's authentication service configuration.

The following example demonstrates how to obtain the user alias:

```
public AMIdentity getIdentityFromSearchAlias(String username,
String realm) {
    ServiceConfig serviceConfig = coreWrapper

.getServiceConfigManager(ISAAuthConstants.AUTH_SERVICE_NAME,

AccessController.doPrivileged(AdminTokenAction.getInstance()))
    .getOrganizationConfig(realm);

    Set<String> realmAliasAttrs = serviceConfig.getAttributes()
        .get(ISAAuthConstants.AUTH_ALIAS_ATTR);

    return IdUtils.getIdentity(username, realm, realmAliasAttrs);
}
```

By combining these approaches, you can search for an identity by using the ID and whichever configured attribute field(s) as necessary.

## *Read attributes of an identity's profile*

After obtaining the profile, use the `AMIdentity#getAttribute(String name)` method.

## *Write a value into an identity's profile*

Create a `Map<String, Set<String>>` structure of the attributes you wish to write, as follows:

```
Map<String, Set<String>> attrs = new HashMap<>();
attrs.put("attribute", Collections.singleton("value"));
user.setAttributes(attrs);
user.store();
```


## Include callbacks

Nodes use *callbacks* to enable interaction with the authenticating user.

AM doesn't support creating your own custom callbacks, but there are many existing implementations available to you. Learn more in [Supported callbacks](#).

Calling the `getCallbacks(Class<t> callbackType)` method on a `TreeContext` - the sole argument to the `process()` method of a node - returns all callbacks of a particular type for the most recent request from the current node. For example, calling `context.getCallbacks(PasswordCallback.class)` returns a list of the `PasswordCallback` callbacks displayed in the UI most recently.

Below is an example of multiple callbacks created by a node and passed to the UI:



PASSWORD EXPIRES IN: 59 SEC



To process the responses to callbacks, you must know the order of the callbacks in the list. You can find the position of the callbacks created by the current node by using the constant properties for each callback position in the processing node.

If the callbacks were created in previous nodes, their positions must be stored in the shared state before subsequent nodes can use them.

The following is the code that created the UI displayed in the previous image:

```
ImmutableList.of(
    new TextOutputCallback(messageType, message.toUpperCase()),
    new PasswordCallback(bundle.getString("oldPasswordCallback"),
false),
    new PasswordCallback(bundle.getString("newPasswordCallback"),
false),
    new
PasswordCallback(bundle.getString("confirmPasswordCallback"),
false),
    confirmationCallback
);
```

Note that the order of callbacks defined in code is preserved in the UI.

### *Send and execute JavaScript in a callback*

A node can provide JavaScript for execution on the client side browser.

For example, the following is a simple JavaScript script named `hello-world.js`:

```
alert("Hello, World!");
```

Execute the script on the client by using the following code:

```
String helloScript = getScriptAsString("hello-world.js");
ScriptTextOutputCallback scriptCallback = new
ScriptTextOutputCallback(helloScript);
ImmutableList<Callback> callbacks =
ImmutableList.of(scriptCallback);
return send(callbacks).build();
```

Variables can be injected using your favorite Java String utilities, such as `String.format(script, myValue)`.

To retrieve the data back from the script, add `HiddenValueCallback` to the list of callbacks sent to the user, as follows:

```
HiddenValueCallback hiddenValueCallback = new  
HiddenValueCallback("myHiddenOutcome", "false");
```

The JavaScript needs to add the required data to the `HiddenValueCallback` and submit the form, for example:

```
document.getElementById('myHiddenOutcome').value = "client side  
data";  
document.getElementById("loginButton_0").click();
```

In the process method of the node, retrieve the hidden callback as follows:

```
Optional<String> result =  
context.getCallback(HiddenValueCallback.class)  
    .map(HiddenValueCallback::getValue)  
    .filter(scriptOutput -> !Strings.isNullOrEmpty(scriptOutput));  
  
if (result.isPresent()) {  
    String myClientSideData = result.get();  
}
```

## Handle multiple visits to the same node

Authentication flow can return to the same decision node by using two different methods.

The first method is to route the failure outcome through a [Retry Limit Decision node](#). This node can limit how many times a user can enter incorrect authentication details. In these instances, the user is returned to re-enter their information; for example, back to an earlier [Username Collector node](#).

The second method involves routing directly back to the currently processing node. To achieve this, use the `Action.send()` method, rather than `Action.goTo()`. The `Action.goTo` method passes control onto the next node in the tree. The `Action.send()` method takes a list of callbacks which you can construct in the current node. The return value is an `ActionBuilder`, which can be used to create an `Action`, as follows:

```
ActionBuilder action = Action.send(ImmutableList.of(new  
ChoiceCallback(), new ConfirmationCallback()));
```

A typical example of returning to the same node is a password change screen where the user must enter their current password, new password, and new password

confirmation. The node that processes these callbacks needs to remain on the screen and display an error message if any of the data entered by the user is incorrect. For example, if the new password and password confirmation do not match.

When a `ConfirmationCallback` is invoked on a screen that was produced by `Action.send()`, it will always route back to the node that created it. Once the details are valid, return an `Action` created using `Action.goTo()` and tree processing can continue as normal.

## Handle errors

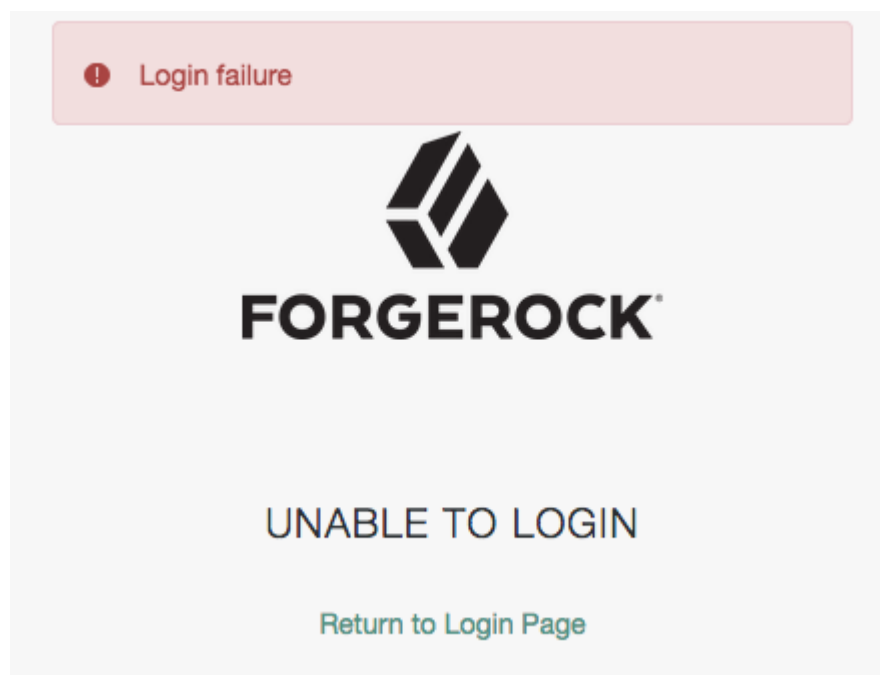
---

This page covers error handling in authentication nodes, including how to report errors to end users and tree administrators, as well as handling unrecoverable errors.

Authentication trees provide a number of ways to output error messages to the user.

### Authentication errors

The most common error to display is a message in the event of an unsuccessful authentication. In an authentication tree, this occurs when the authentication process terminates at the failure node:



### Unrecoverable errors

By default, when a catastrophic error occurs during node processing, a `NodeProcessException` exception should be thrown, which halts the authentication journey immediately, and displays a generic error message. This may not be desirable, as it could create a negative user experience.

Instead, errors that occur during node processing should be caught within the processing block of the node's code, and the user should be routed to an erroneous state outcome. It may be appropriate to have a single error outcome, multiple error outcomes, or no error outcome at all, depending on the node.

It is valuable to store information about the cause of the error in the shared state, in case a node further along the tree processes it. This information should include error text to display to the user. If the shared state is used for this purpose, it is important to document not only the meaning of the various outcomes, but also the keys used to store information in the shared state.

## Configuration errors

You can display error messages to the tree administrator; for example, when a configuration property of a node is required, but not provided.

To automatically display an appropriate error message when required values are missing, annotate your config property with `@RequiredValueValidator`, as follows:

```
@Attribute(order = 300, validators =
    {RequiredValueValidator.class})
Set<String> accountSearchBaseDn();
```

To control the messages displayed on error, ensure there is a `.properties` file under `src/main/resources/org.forgerock.openam.auth.nodes` with the same name as your node class. For more information, see [Internationalization](#).

## Plugin class

---

The plugin class is responsible for informing AM about the details of the customized authentication node. There is little variation between the plugin class for each authentication node, other than the version number and class names within.

Authentication nodes are installed into the product using the AM plugin framework. All AM plugins are created by implementing `org.forgerock.openam.plugins.AmPlugin` interface and registering it using the Java service architecture - placing a file in `META-INF/services`.

For plugins that provide authentication nodes there is an abstract implementation of the `AmPlugin` interface named `org.forgerock.openam.auth.node.api.AbstractNodeAmPlugin`.

The following is an example of the plugin class for an authentication node:

```

public class MyCustomNodePlugin extends AbstractNodeAmPlugin { ①

    private static String currentVersion = "1.0.0"; ②

    @Override
    protected Map<String, Iterable<? extends Class<? extends Node>>>
    getNodesByVersion() {
        return Collections.singletonMap("1.0.0",
Collections.singletonList(MyCustomNode.class)); ③
    }

    @Override
    public String getPluginVersion() {
        return MyCustomNodePlugin.currentVersion;
    }
}

```

- ① Name the plugin class after the core class, and append *Plugin*. For example, `MyCustomNodePlugin`.
- ② Provide a version number for the authentication node.
- ③ Ensure a call to the `getNodesByVersion()` function returns the core classes of the authentication nodes to register. In this example the version is `1.0.0`, and there is just one node being registered as that version.

AM plugins are notified of the following events:

### ***onInstall***

The plugin has been found during AM startup, and is being installed for the first time. It should create all the services and objects it needs.

### ***onStartup(StartupType startupType)***

The plugin is installed and is being started. Any dependency plugins can be relied on as having been started.

The type of startup is provided:

- `FIRST_TIME_INSTALL`. The AM instance has been installed for the first time.
- `FIRST_TIME_DEMO_INSTALL`. The AM deployment has been installed for the first time, using an embedded data store as the config and user stores.
- `NORMAL_STARTUP`. The AM instance is starting from a previously installed state, or is joining an already installed cluster.

### ***onShutdown***

The AM instance is in the process of shutting down cleanly. Any resources the plugin is using should be released and cleaned up.

### ***upgrade(String fromVersion)***

An existing version of the plugin is installed, and a new version has been found during startup. The plugin should make any changes it needs to the services and objects used in the previous version, and create all the services and objects required by the new version.

The version of the plugin being upgraded is provided.

### ***onAmUpgrade(String fromVersion, String toVersion)***

An AM system upgrade is in progress. Any updates needed to accommodate the AM upgrade should be made.

Plugin-specific upgrade should not be made here, as `upgrade` will be called subsequently if the plugin version has also changed.

The AM version being upgraded from, and to, are provided.

The plugin is responsible for maintaining a version number for its content, which is used for triggering appropriate events for installation and upgrade.

For more information, see [amPlugin](#) in the *AM Public API Javadoc*.

## Upgrade nodes and change node configuration

---

Over time, it may become necessary to change the schema of the configuration for your node.

When this happens, the changes must be propagated to the AM configuration system. To ensure an update of the AM configuration, use either of the following methods, depending on the stage of development:

- In the development stage, give your nodes the special version number `0.0.0`. Any AM configuration created by nodes that have this special version number is wiped on each restart of AM.

#### IMPORTANT

If you are using custom nodes with version `0.0.0` in trees, you must remove them from the trees before restarting AM and reinsert them after the restart. If you do not do this, the entire tree cannot be viewed in the UI after the restart.

- After moving to production and switching to semantic versioning, you must write upgrade functions into the node to locate existing configuration and convert it to the new schema.

For information on upgrading schema in production mode, refer to Upgrade simple node configuration schema changes and Upgrade complex node configuration

schema changes.

## Upgrade simple node configuration schema changes

This section explains how to upgrade nodes with simple schema changes. For example, changing an attribute to a compatible type.

When configuration schema changes are simple, call the `PluginTools#upgradeAuthNode(Class)` method in the `upgrade` method of your plugin, as follows:

```
@Override
public void upgrade(String fromVersion) throws PluginException {
    pluginTools.upgradeAuthNode(MyCustomNode.class);
}
```

Examples of simple schema changes include:

- Changing an attribute type to one that is backwards-compatible with any existing values.

For example changing an integer to a string type, or `T` to `Set<T>`.

- Adding a new attribute that has a default value defined.

For example:

```
public class MyCustomNode implements Node {
    public interface Config {
        @Attribute(order = 1)
        String existingAttribute();
        @Attribute(order = 2)
        default Integer newAttribute() {
            return 5;
        }
    }
    // ...
}
```

## Upgrade complex node configuration schema changes

This section explains how to upgrade nodes that are changing the configuration schema such that existing values would clash with the new schema. For example, changing an attribute to an incompatible type.

When configuration schema changes are complex, use the API provided in the `com.sun.identity.sm` package. In this example, version 1.0.0 of a node has the following configuration schema:

```
public interface Config {
    @Attribute(order = 1)
    String name();
}
```

Version 2.0.0 of the node requires the user's given name and family name separately, rather than simply a name string. The config for version 2.0.0 is as follows:

```
public interface Config {
    @Attribute(order = 1)
    String givenName();

    @Attribute(order = 2)
    String familyName();
}
```

To upgrade this example node configuration, find all existing instances of configuration created by the version 1.0.0 node, find the current values for the `name` attribute, and split it on the first space character to use in the two new attributes.

The following code shows how to upgrade the schema of this example node:

```
@Override
public void upgrade(String fromVersion) throws PluginException {
    try {
        SSOToken token = coreWrapper.getAdminToken();
        String serviceName = MyCustomNode.class.getSimpleName();
        ServiceConfigManager configManager = new
        ServiceConfigManager(serviceName, token);

        // Read all the values from all node in all the realms that
will need replacing
        OrganizationConfigManager realmManager = new
        OrganizationConfigManager(token, "/");
        Set<String> realms = ImmutableSet.<String>builder()
            .add("/")
            .addAll(realmManager.getSubOrganizationNames("*", true))
            .build();
        Map<Pair<Realm, String>, String> oldValues = new HashMap<>();
        for (String realm : realms) {
```



```

        ServiceConfig container =
configManager.getOrganizationConfig(realm, null);
        for (String nodeId : container.getSubConfigNames()) {
            ServiceConfig nodeConfig = container.getSubConfig(nodeId);
            String name =
nodeConfig.getAttributes().get("name").iterator().next();
            oldValues.put(Pair.of(Realms.of(realm), nodeId), name);
        }
    }

    // Do the upgrade of the schema
    pluginTools.upgradeAuthNode(MyCustomNode.class);

    // Remove the old value and set the new values
    for (Map.Entry<Pair<Realm, String>, String> nameForUpdate :
oldValues.entrySet()) {
        String realm = nameForUpdate.getKey().getFirst().asPath();
        String nodeId = nameForUpdate.getKey().getSecond();
        String name = nameForUpdate.getValue();
        int spaceIndex = name.indexOf(" ");

        ServiceConfig container =
configManager.getOrganizationConfig(realm, null);
        ServiceConfig nodeConfig = container.getSubConfig(nodeId);
        nodeConfig.removeAttribute("name");
        nodeConfig.setAttributes(ImmutableMap.of(
            "givenName", singleton(name.substring(0, spaceIndex)),
            "familyName", singleton(name.substring(spaceIndex + 1))));
    }
} catch (SSOException | SMSEException | RealmLookupException e) {
    throw new PluginException("Could not upgrade", e);
}
}
super.upgrade(fromVersion);
}

```

## Internationalization

Internationalization (i18n) of content targets both the end user and the node administrator. Messages sent to users and other UI can be internationalized.

Additionally, error messages and administrator-facing UI can be internationalized using the same mechanism for better operator experience.

Internationalized nodes use the locale of the request to find the correct resource bundle, with a default fallback if none is found.

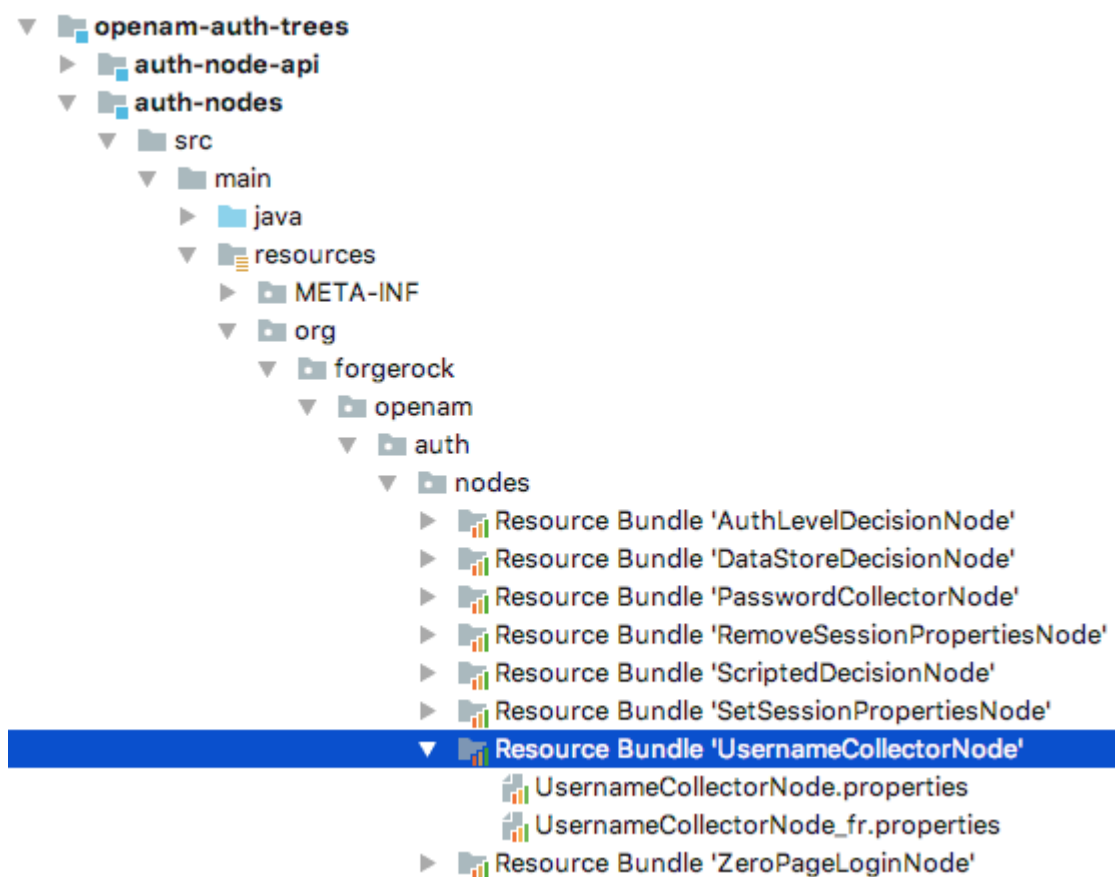
## Localize node UI text

1. Create a Java resource bundle under the `resources` folder in the Maven project for your node.

The path and filename must match that of the core class that will use the translated text.

For example, the resource bundle for the Username Collector node is located in the following path:

`src/main/resources/org/forgerock/openam/auth/nodes/UsernameCollectorNode` .



*Figure 4. Example resource bundle*

2. Add the properties and strings that the node will display to the user.

For example:

```
callback.username=User Name
```

3. Create a `.properties` file in the resource bundle for each language your node will display.

The filename must include the language identifier, as per [rfc5646 - Tags for Identifying Languages](#).

For example, for French translations your `.properties` file could be called `UsernameCollectorNode_fr.properties`.

4. Replicate the properties and translate the values in each `.properties` files.

For example:

```
callback.username=Nom d'utilisateur
```

5. In the core class for your node, specify the path to the resource bundle from which the node will retrieve the translated strings:

```
private static final String BUNDLE =  
    "org/forgerock/openam/auth/nodes/UsernameCollectorNode";
```

6. Define a reference to the bundle using the `getBundleInPreferredLocale` function to enable retrieval of translated strings:

```
ResourceBundle bundle =  
    context.request.locales.getBundleInPreferredLocale(  
        BUNDLE, getClass().getClassLoader());
```

7. Use the `getString` function whenever you need to retrieve a translation from the resource bundle:

```
return send(new  
    NameCallback(bundle.getString("callback.username"))).build();
```

## Build and install nodes

---

This section explains how to build and install authentication nodes for use in authentication trees.

### Build and install a custom authentication node

1. Change to the root directory of the Maven project of the custom nodes.

For example:

```
$ cd /Users/Forgerock/Repositories/am-external/openam-auth-
```

## trees/auth-nodes

2. Run the `mvn clean package` command.

The project will generate a `.jar` file containing your custom nodes. For example, `auth-nodes-version.jar`.

3. Include the custom `.jar` file in the AM `.war` file, as described in [Customize the AM WAR file](#).

### NOTE

Delete or overwrite older versions of the nodes `.jar` file from the `WEB-INF/lib/` folder, to avoid clashes.

### IMPORTANT

If you are using custom nodes with version `0.0.0` in trees, you must remove them from the trees before restarting AM and reinsert them after the restart. If you do not do this, the entire tree cannot be viewed in the UI after the restart.

4. Restart AM for the new nodes to become available.

The custom authentication node is now available in the tree designer to add to authentication trees:

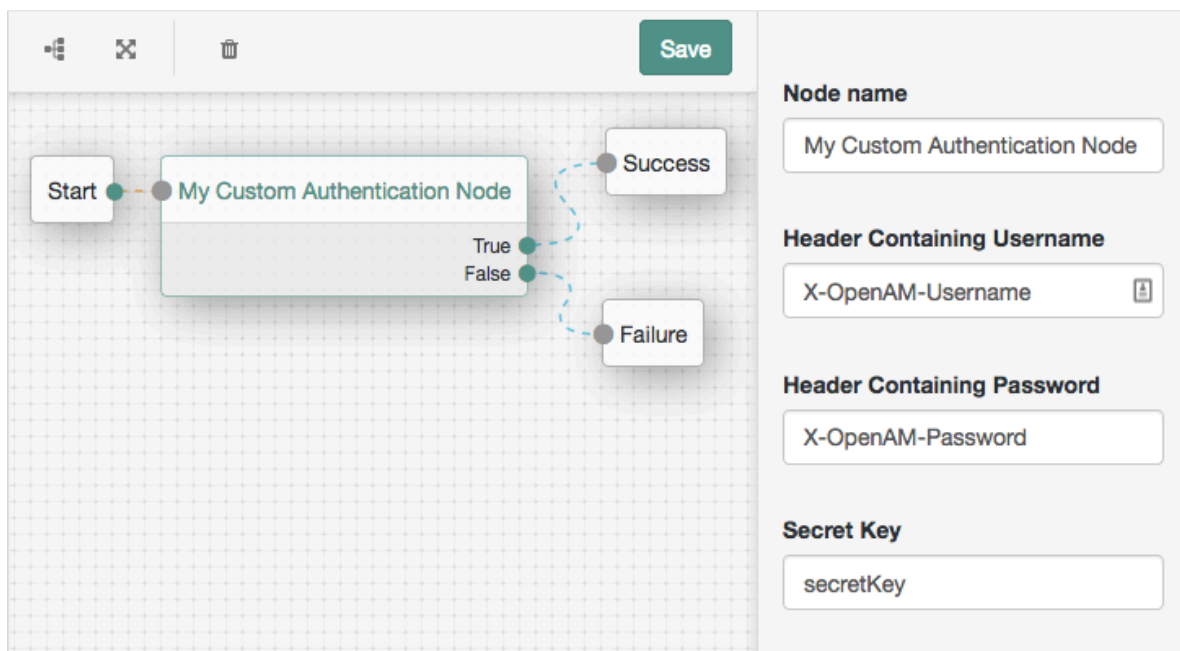


Figure 5. Custom node in a tree

For more information on using the tree designer to manage authentication trees, refer to [Configure authentication trees](#).

For information on upgrading custom nodes, refer to [Upgrade nodes and change node configuration](#).

# Maintain authentication nodes

---

This page covers post-installation tasks relating to authentication nodes, such as testing, debugging, auditing, and performance monitoring.

## Test nodes

You can test authentication nodes in numerous ways. For example, you can use unit tests, functional tests, and perform exploratory or manual testing.

Authentication nodes are well suited to tests that have a high percentage of code coverage. The low number of static dependencies means that unit testing of the node class itself can occur, rather than simply the business logic classes, as was the case for modules. Furthermore, as nodes should be significantly smaller than modules, testing iterations should be much shorter.

### *Unit tests*

Your unit tests should aim for an appropriately high-level percentage of coverage of the code. Unit testing becomes easier with nodes, as most of the business logic is defined by the tree layout, rather than in the nodes themselves.

At minimum, the `process(TreeContext context)` method should be tested to ensure that all appropriate code paths are triggered based on the existence, or lack of, appropriate values in the shared state and callbacks.

The `TreeContext` class and contents have been designed to make sure they are simple to use in unit tests, without the need to resort to mocking.

### *Functional tests*

Functional tests involve deploying the node into an AM instance and testing it using the authentication REST API. They should be written to cover all normal flows through the node.

All the appropriate code paths discovered through unit testing should be functionally tested to ensure that helper, utility, and related mechanisms function as expected.

Additionally, functional tests will ensure that the business logic is correctly called and processed as expected.

#### **TIP**

Mocking expected services may be useful when functionally testing nodes that call out to third-party services.

## Manual testing

Manual testing should occur both during and after node development.

During development, it is expected a node developer will frequently load and reload nodes to ensure they operate as expected, including configuration and execution, as well as any expected error conditions.

After development, manual testing should continue in an exploratory fashion. Simply using a node numerous times can often highlight areas left unpolished, or particular usability issues that may be missed by automated testing.

## Debug nodes

Add debug logging to your custom node to help administrators and support staff investigate any issues which may arise in production.

To add debug logging to a node, obtain a reference to the `amAuth` SLF4J Logger instance.

For example, you can assign the logger to a private field as follows:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

// ...

private final Logger logger = LoggerFactory.getLogger("amAuth");
```

### NOTE

Consider the logging level you use; excessive use of the `error` or `warning` level can cause debug logs to fill, and can have a negative performance impact, if your node is used frequently.

You can also use the SLF4J `varargs` methods to defer string concatenation to SLF4J. This means string concatenation can be skipped if the configured logging level means that your message will not be written.

The following example uses the `debug` level:

```
logger.debug("authLevelSufficient {}", authLevelSufficient);
```

## Audit logging

Audit logging helps administrators to investigate user and system behavior.

AM records all incoming calls as access events. Additionally, in order to capture further details regarding authentication flows, AM records an authentication audit event for each node, and the tree outcome.

A node can provide extra data to be included in the standard audit event which is logged when an authentication node completes.

AM logs an `AM-NODE-LOGIN-COMPLETED` audit event each time an authentication node completes. To add extra information to this audit event, override the node interface method `getAuditEntryDetail`.

For example, the [Retry Limit Decision node](#) overrides this method to record how many retries remain:

```
@Override
public JsonValue getAuditEntryDetail() {
    return json(object(field("remainingRetries",
String.valueOf(retryLimitCount))));
}
```

When this node is processed, it results in an audit event similar to the following:

```
{
  "realm": "/",
  "transactionId": "45453155-cf94-4e23-8ee9-ecdfc9f97e12-1785617",
  "component": "Authentication",
  "eventName": "AM-NODE-LOGIN-COMPLETED",
  "entries": [
    {
      "info": {
        "nodeOutcome": "Retry",
        "treeName": "Example",
        "displayName": "Retry Limit Decision",
        "nodeType": "RetryLimitDecisionNode",
        "nodeId": "bf010b6b-61f8-457e-80f3-c3678e5606d2",
        "authLevel": "0",
        "nodeExtraLogging": {
          "remainingRetries": "2"
        }
      }
    }
  ],
  "timestamp": "2018-08-24T09:43:55.959Z",
  "trackingIds": [
```

```
    "45453155-cf94-4e23-8ee9-ecdfc9f97e12-1785618"
  ],
  "_id": "45453155-cf94-4e23-8ee9-ecdfc9f97e12-1785622"
}
```

The result of the `getAuditEntryDetail` method is stored in the `nodeExtraLogging` field.

## Monitor nodes

You can track authentication flows which complete with success, failure, or timeout as an outcome by using the metrics functionality built-in to AM.

For more information, see [Monitor AM instances](#).

You can also use the following nodes in a tree to create custom metrics:

- [Meter node](#)
- [Timer Start node](#)
- [Timer Stop node](#)

## Troubleshoot node development

---

This page offers solutions to issues that may occur when developing authentication nodes.

1. *I installed my node in AM. Why doesn't it appear in the authentication tree designer?*

The `authNodeName.properties` file for your node must include a `nodeDescription` property to ensure that that your node appears in the authentication tree designer.

AM uses the `nodeDescription` property value as the name of your node.

2. *How do I get new attributes to appear in the node after the service has been loaded once?*

See [Upgrade nodes and change node configuration](#).

3. *What type of exception should I throw so that the framework handles it gracefully?*

To display a custom message to the user, exceptions must be handled inside the node and an appropriate information callback returned.

For more information, see [Handle errors](#).

4. *Do I need multiple projects/jars for multiple nodes?*

No - you can bundle multiple nodes into one plugin, which should be deployed in one single `.jar` file.

See [Build and install nodes](#).



5. *What ForgeRock utilities exist for me to use to assist in the node building experience?*

A number of utilities are available for use in your integrations and custom nodes.

See the [AM Public API Javadoc](#).

6. *Transient State vs Shared State - When should I use one or the other?*

Transient state is used for secret values that should not persist.

See [Store values in shared tree state](#).

7. *If my service collects a username in a different way from the [Username Collector node](#), where do I put the username from the framework to get the principal?*

See [Access an identity's profile](#).

8. *Where do I go for examples of authentication nodes?*

There are many public examples of ForgeRock community nodes at <https://github.com/ForgeRock>.

Examples of community nodes written by third parties can be found on the [Marketplace](#) website.

For source access to the authentication nodes builtin to AM, see [How do I access and build the sample code provided for PingAM?](#) in the *Knowledge Base*.

Was this helpful?  