Authentication Node Development Guide

/ ForgeRock Access Management 6.5

Latest update: 6.5.5

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

Copyright © 2018-2020 ForgeRock AS.

Abstract

Guide to developing authentication nodes for use in ForgeRock® Access Management (AM) authentication trees. AM provides authentication, authorization, entitlement, and federation software.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.

To view a copy of this license, visit https://creativecommons.org/licenses/by-nc-nd/3.0/ or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

ForgeRock® and ForgeRock Identity Platform™ are trademarks of ForgeRock Inc. or its subsidiaries in the U.S. and in other countries. Trademarks are the property of their respective owners.

UNLESS OTHERWISE MUTUALLY AGREED BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

DejaVu Fonts

Bitstream Vera Fonts Copyright

Copyright (c) 2003 by Bitstream, Inc. All Rights Reserved. Bitstream Vera is a trademark of Bitstream, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of the fonts accompanying this license ("Fonts") and associated documentation files (the "Font Software"), to reproduce and distribute the Font Software, including without limitation the rights to use, copy, merge, publish, distribute, and/or sell copies of the Font Software, and to permit persons to whom the Font Software is furnished to do so, subject to the following conditions:

The above copyright and trademark notices and this permission notice shall be included in all copies of one or more of the Font Software typefaces.

The Font Software may be modified, altered, or added to, and in particular the designs of glyphs or characters in the Fonts may be modified and additional glyphs or characters may be added to the Fonts, only if the fonts are renamed to names not containing either the words "Bitstream" or the word "Vera".

This License becomes null and void to the extent applicable to Fonts or Font Software that has been modified and is distributed under the "Bitstream Vera" names.

The Font Software may be sold as part of a larger software package but no copy of one or more of the Font Software typefaces may be sold by itself.

THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL BITSTREAM OR THE GNOME FOUNDATION BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.

Except as contained in this notice, the names of Gnome, the Gnome Foundation, and Bitstream Inc., shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Font Software without prior written authorization from the Gnome Foundation or Bitstream Inc., respectively. For further information, contact: fonts at gnome dot org.

Arev Fonts Copyright

Copyright (c) 2006 by Tavmjong Bah. All Rights Reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of the fonts accompanying this license ("Fonts") and associated documentation files (the "Font Software"), to reproduce and distribute the modifications to the Bitstream Vera Font Software, including without limitation the rights to use, copy, merge, publish, distribute, and/or sell copies of the Font Software, and to permit persons to whom the Font Software is furnished to do so, subject to the following conditions:

The above copyright and trademark notices and this permission notice shall be included in all copies of one or more of the Font Software typefaces.

The Font Software may be modified, altered, or added to, and in particular the designs of glyphs or characters in the Fonts may be modified and additional glyphs or characters may be added to the Fonts, only if the fonts are renamed to names not containing either the words "Tavmjong Bah" or the word "Arev".

This License becomes null and void to the extent applicable to Fonts or Font Software that has been modified and is distributed under the "Tavmjong Bah Arev" names.

The Font Software may be sold as part of a larger software package but no copy of one or more of the Font Software typefaces may be sold by itself.

THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL TAVMJONG BAH BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.

Except as contained in this notice, the name of Tavmjong Bah shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Font Software without prior written authorization from Tavmjong Bah. For further information, contact: tavmjong @ free . fr.

FontAwesome Copyright

Copyright (c) 2017 by Dave Gandy, https://fontawesome.com/.

This Font Software is licensed under the SIL Open Font License, Version 1.1. See https://opensource.org/licenses/OFL-1.1.

Table of Contents

Preface iv
1. About Authentication Nodes 1
    Types of Nodes 2
    Differences Between Modules and Nodes 4
    Converting Modules to Nodes 5
    Determining the Functionality of a Node 5
2. Preparing for Development 7
    The Maven Project for an Authentication Node 12
3. The Node Class 16
    The Meta Data Annotation 20
    The Node Interface 21
    The Config Interface 21
    Injecting Objects Into a Node Instance 26
    The Action Interface 28
    Handling Errors 35
4. The Plugin Class 37
    Upgrading Nodes and Configuration Changes 38
5. Internationalization 42
6. Building and Installing 44
7. Maintaining Authentication Nodes 46
    Testing 46
    Debugging 47
    Auditing 48
    Monitoring 49
8. Troubleshooting 50
A. Getting Support 52
Glossary 53

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

Preface

This guide provides guidance and best practice for developing and maintaining authentication nodes in AM.

For information on configuring and using authentication trees, see "About Authentication Trees" in the *Authentication and Single Sign-On Guide*.

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.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

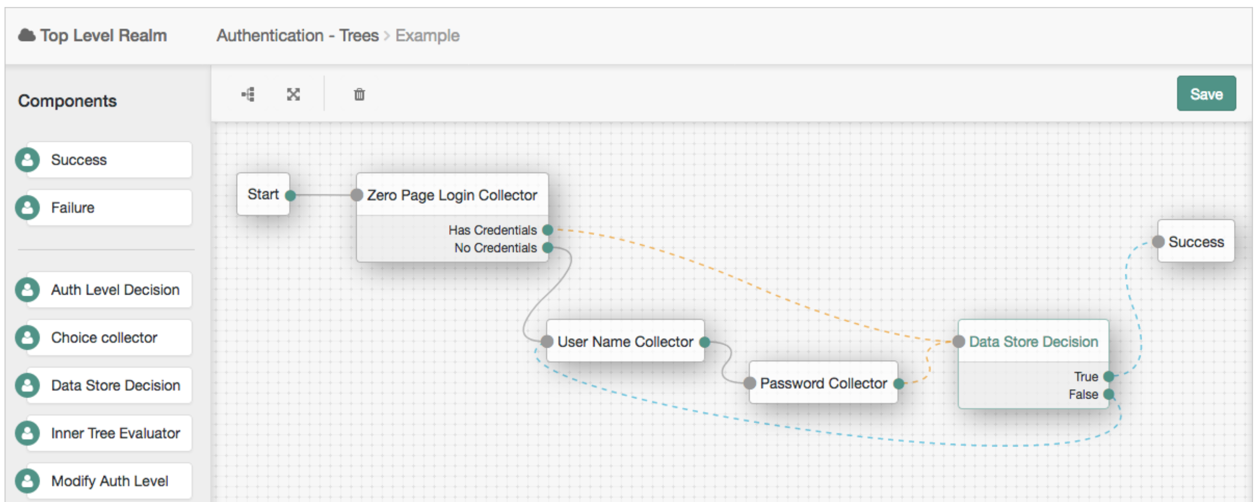
Chapter 1
About Authentication Nodes

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 within chains.

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

Example Authentication Tree



Nodes are designed to be single-responsibility, and where appropriate should be loosely coupled to 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.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

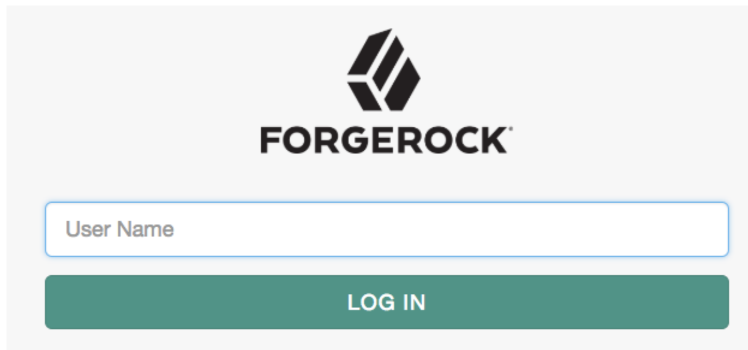
Types of Nodes

This section covers the two most common types of authentication node.

Collector Nodes

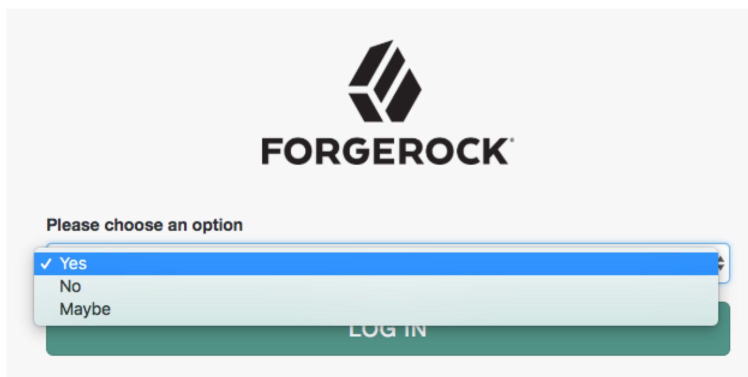
Collector nodes capture data from a user during the authentication process. This data is often captured by a *callback* that is rendered in the UI as a text field, drop-down list, or other form component.

Examples of collector nodes includes the Username Collector Node and Password Collector Node.



Collector nodes may perform some 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.



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.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

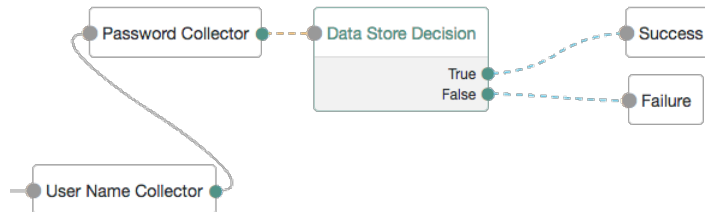
Decision Nodes

Decision nodes retrieve the state produced by one or more nodes, perform some processing on it, optionally store some derived information in the shared state, and provide one or more outcomes depending on the result.

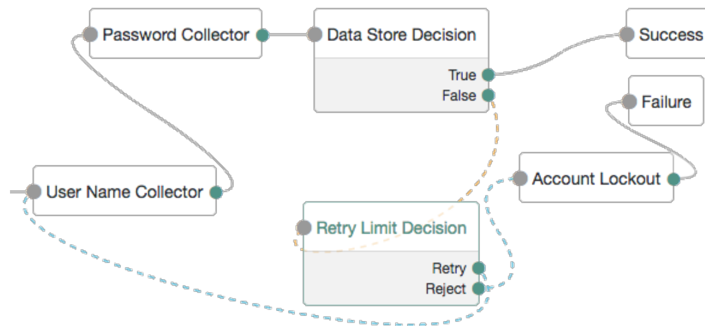
The simplest decision node returns a boolean outcome - `true`, or `false`.

Complex nodes may have additional outcomes. For example the LDAP Decision Node offers additional outcomes of *Locked* and *Expired*. 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.



Note

Nodes May Have Prerequisite Stages

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

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.

Differences Between Modules and Nodes

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 and 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.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

- Nodes are easier to test due to their smaller code size and their specific functionality.

Converting 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 "Sharing Configuration Between Nodes".

Determining the Functionality of a Node

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

Before creating a set of nodes, a node developer must first understand the level of granularity they 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, the developer may create nodes that take values from the shared state and save it to the user's profile.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

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

The following guidelines help a node developer to 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 then 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.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

Chapter 2
Preparing for Development

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

> **Tip**
>
> For information about customizing post-authentication hooks for a tree, see "Creating Post-Authentication Hooks for Trees" in the *Authentication and Single Sign-On Guide*.

To prepare for creating or modifying authentication nodes, complete the following procedures:

- "To Prepare an Environment For Building Custom Authentication Nodes"
- "To Set Up a Maven Project For Building Custom Authentication Nodes"

To Prepare an Environment For Building Custom Authentication Nodes

Complete the following steps to set up your environment for building custom authentication nodes:

1. Ensure your BackStage account is part of a subscription:

    a. In a browser, navigate to the ForgeRock BackStage website and log in or register for an account.

    b. Confirm or request that your account is added to a subscription. For more details, see "Getting access to product support" in the *ForgeRock Knowledge Base*.

2. Install Apache Maven 3.2.5 or later, and Oracle JDK or OpenJDK 1.8.

    > **Tip**
    >
    > To verify the installed versions, run the **mvn --version** command:

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

```
$ mvn --version
Apache Maven 3.2.5 (12a6b3acb947671f09b81f49094c53f426d8cea1; 2014-12-14T17:29:23+00:00)
Maven home: /usr/local/apache-maven/apache-maven-3.2.5 Java version: 1.8.0_121, vendor: Oracle
 Corporation
Java home: /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/jre Default locale:
 en_US,
platform encoding: UTF-8 OS name: "mac os x", version: "10.11.6", arch: "x86_64", family: "mac"
```

3. Configure Maven to be able to access the ForgeRock repositories by adding your BackStage credentials to the Maven `settings.xml` file. For details, see How do I access the ForgeRock protected Maven repositories? in the *ForgeRock Knowledge Base*.

   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/repo/private-releases</url>
          <releases>
              <enabled>true</enabled>
              <checksumPolicy>fail</checksumPolicy>
          </releases>
          <snapshots>
              <enabled>false</enabled>
              <checksumPolicy>warn</checksumPolicy>
          </snapshots>
      </repository>
  </repositories>
  </profile>
</profiles>
<activeProfiles>
  <activeProfile>forgerock</activeProfile>
</activeProfiles>
```

To Set Up a Maven Project For Building Custom Authentication Nodes

ForgeRock provides a Maven archetype for creating 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.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

Note

Ensure you have completed the steps in "To Prepare an Environment For Building Custom Authentication Nodes" before proceeding.

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

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

   a. In a terminal window, navigate to a folder in which to create the new Maven project. For example:

   ```
   $ cd ~/Repositories
   ```

   b. 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 that is registered through a plugin is stored with the name returned as a result of calling `Class.getSimpleName()`. AM does *not* 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 that the name does not collide with the names of existing nodes.

   For example:

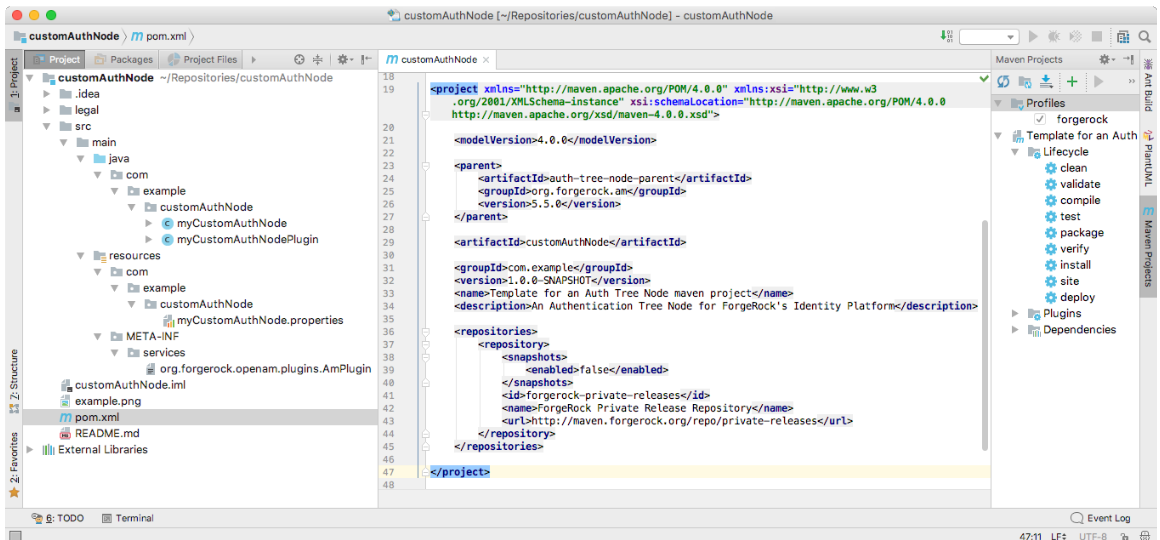
Copyright © 2018-2020 ForgeRock AS. All rights reserved.

```
$ 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=6.5.0 \
  -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.

The project resembles the following:

Node Project Created by Using the Archetype



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

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

a. If you do not already have a ForgeRock BackStage account, get one from https://backstage.forgerock.com. You cannot clone the `am-external` repository without a BackStage account.

b. Clone the `am-external` repository:

```
$ git clone https://myBackStageID@stash.forgerock.org/scm/openam/am-external.git
```

Tip

URL encode your BackStage ID if it contains special characters. For example **:** becomes `%3A` and **@** becomes `%40`.

Enter your BackStage password when prompted to do so.

c. Check out the `release/6.5.0` branch:

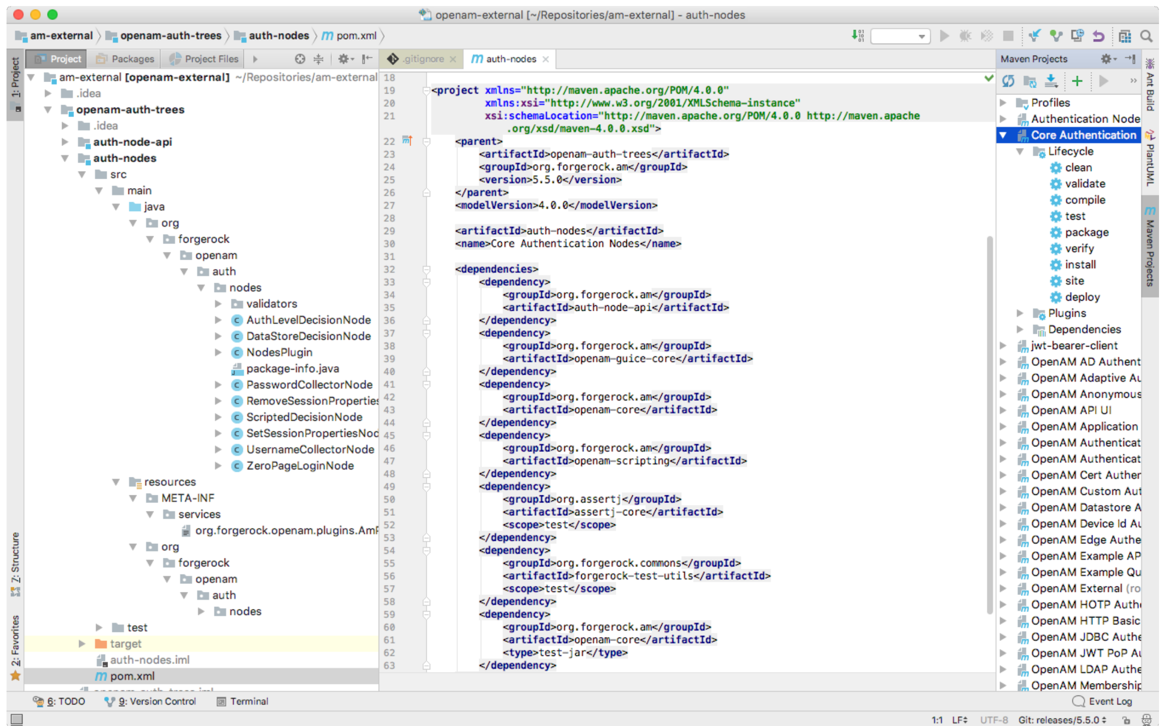
```
$ cd am-external
$ git checkout releases/6.5.0
```

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

The project resembles the following:

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

Node Project Cloned from ForgeRock



The Maven Project for an Authentication Node

When configuring your 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 6.5 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:

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

- 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.

The Maven project for an authentication node contains the following files:

`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>6.5.5</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>
```


Copyright © 2018-2020 ForgeRock AS. All rights reserved.

```
  </dependencies>

  <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;resolution:=optional</
Import-Package>
              </manifestEntries>
            </transformer>
          </transformers>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

***`authNodeName`*`.java`**

Core class for the custom authentication node. See "*The Node Class*".

***`authNodeName`*`Plugin.java`**

Plugin class for the custom authentication node. See "*The 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.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

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

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

Chapter 3
The 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.

Conceptually, the `Node` class is structured as follows:

1. The *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.

2. Any private constants

3. The *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 it’s configuration.

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

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

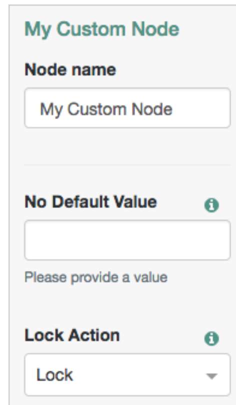
The `config` interface configures values using methods. To provide no default value to the tree administrator, provide the method's signature but not the implementation. To provide a default value to the tree administrator, mark the method as `default` and provide both a method and a value. For example:

```
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:



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.

4. The *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.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

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

5. The *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.

6. Any private methods

7. Optionally, a custom outcome provider

The following example is the `SetSessionPropertiesNode` class, taken from the Set Session Properties Node:

```
/*
 * The contents of this file are subject to the terms of the Common Development and
 * Distribution License (the License). You may not use this file except in compliance with the
 * License.
 *
 * You can obtain a copy of the License at legal/CDDLv1.0.txt. See the License for the
 * specific language governing permission and limitations under the License.
 *
 * When distributing Covered Software, include this CDDL Header Notice in each file and include
 * the License file at legal/CDDLv1.0.txt. If applicable, add the following below the CDDL
 * Header, with the fields enclosed by brackets [] replaced by your own identifying
 * information: "Portions copyright [year] [name of copyright owner]".
 *
 * Copyright 2017-2018 ForgeRock AS.
 */

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;
```


Copyright © 2018-2020 ForgeRock AS. All rights reserved.

```
import org.forgerock.openam.auth.node.api.TreeContext;
import org.forgerock.openam.auth.nodes.validators.SessionPropertyValidator;
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)  ❶
public class SetSessionPropertiesNode extends SingleOutcomeNode {  ❷

    /**
     * Configuration for the node.
     */
    public interface Config {  ❸
        /**
         * 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  ❹
    public SetSessionPropertiesNode(@Assisted Config config) {
        this.config = config;
    }

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

Key:

❶ The `@Node.Metadata` annotation. See "The Meta Data Annotation".
❷ Implementing the `Node` interface. See "The Node Interface".
❸ Implementing the `Config` interface. See "The Config Interface".

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

4. Injecting the `Node` instance. See "Injecting Objects Into a Node Instance".
5. Creating an `Action` instance. See "The Action Interface".

The Meta Data 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` 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.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 "The Config Interface".

`configValidator`

An optional class name used to validate the provided configuration.

For example, the following is the `@Node.Metadata` annotation from the Set Session Properties Node:

```
@Node.Metadata(outcomeProvider = SingleOutcomeNode.OutcomeProvider.class,
  configClass = SetSessionPropertiesNode.Config.class)
```

For more information on the `@Node.Metadata` annotation, see the Node.Metadata annotation type in the *AM 6.5.5 Public API Javadoc*.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

The 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 *AM 6.5.5 Public 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 *AM 6.5.5 Public 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 *AM 6.5.5 Public API Javadoc*.

The 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:

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

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

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

    @Attribute(order = 3)
    default YourCustomEnum action() {
        return YourCustomEnum.LockScreen; ❸
    };
}
```

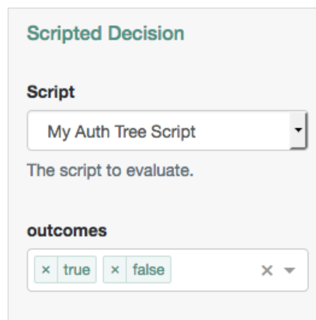
Key:

❶ 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()`.

❷ 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.

❸ 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:



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 6.5.5 Public API Javadoc*.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

Sharing 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.

Creating 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 "Allowing Multiple Instances of an Auxiliary Service".

Referencing 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 get instance methods on that class, for example:

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

Reinstalling 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 then 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:

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

```
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 "Upgrading Nodes and Configuration Changes".

Allowing 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.getOrDefault(Constants.ORGANIZATION_NAME, "/");
           }
```


Copyright © 2018-2020 ForgeRock AS. All rights reserved.

```
            try {
                return InjectorHolder.getInstance(AnnotatedServiceRegistry.class)
                        .getRealmInstances(MyAuxService.class, Realms.of(realmName))
                        .stream()
                        .collect(Collectors.toMap(MyAuxService::id, MyAuxService::id));
            } catch (SSOException | SMSException | RealmLookupException e) {
                LoggerFactory.getLogger("amAuth").error("Couldn't load realm {}", realmName, e);
                throw new IllegalStateException("Couldn't load realm that was passed", e);
            }
        }
    }
    // ...
}
```

Obtaining Configuration of Services Built-in to AM

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.

Obtaining the 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:

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

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

Obtaining the 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();
```

Injecting 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.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

@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*.

Using 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
    }
}
```


Copyright © 2018-2020 ForgeRock AS. All rights reserved.

```
@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.

The Action Interface

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

The `Action` interface uses the following fields:

`callbacks`

A list of the callbacks that have been requested by the node. This list may be `null`.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

`outcome`

The result of the node.

`sessionProperties`

A map of properties that will be added to the final session if the authentication tree completes successfully.

Use the `putSessionProperty(String key, String value)` and `removeSessionProperty(String key)` methods to add or remove entries from the map.

`sharedState`

A JSON representation of the shared state - the properties set so far by nodes in the tree.

See "Storing Values in Shared Tree State".

`transientState`

A JSON representation of any transient state - properties with security or secret values set by previous nodes in the tree. Transient tree state is not persisted to the authentication session, and is only available until the authentication flow reaches the next node requiring user interaction.

See "Storing Values in Shared Tree State".

The methods provided within the `Action` interface are as follows:

`goTo()`

Specify the exit path to take, and move on to the next node in the tree.

For example:

```
return goTo(false).build();
```

See the goTo method in the *AM 6.5.5 Public API Javadoc*.

`send()`

Send the specified callbacks to the user for them to interact with.

For example, the Username Collector Node uses the following code to send the `NameCallback` callback to the user to request the `USERNAME` value:

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

See the send methods in the *AM 6.5.5 Public API Javadoc*.

`sendingCallbacks()`

Returns true if the action is a request for input from the user.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

See the sendingCallbacks method in the *AM 6.5.5 Public API Javadoc*.

For example, the following is the `Action` implementation from the Auth Level Decision Node:

```
@Override
public Action process(TreeContext context) throws NodeProcessException {
    JsonValue authLevel = context.sharedState.get(AUTH_LEVEL);
    boolean authLevelSufficient = !authLevel.isNull()
                                  && authLevel.asInteger()
                                  >= config.authLevelRequirement();
    return goTo(authLevelSufficient).build();
}
```

For more information, see the Action class in the *AM 6.5.5 Public API Javadoc*.

Storing 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 "*Implementing Sessions*" in the *Authentication and Single Sign-On Guide*.

To Set and Get Values in Tree State

- Use the following code examples to set and get values in shared tree state:
  - To store a value in shared tree state, create a copy of the existing tree state, insert the new value into the copy, and then replace the existing state with the copy before returning the `Action` from the `process` method.

    For example:

    ```
    public Action process(TreeContext context) {
      JsonValue copyState = context.sharedState.copy().put(USERNAME, name);
      goToNext().replaceSharedState(copyState).build();
    }
    ```

  - To get a value from the shared tree state, use the following code:

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

```
public Action process(TreeContext context) {
  context.sharedState.get(USERNAME).asString()
}
```

Storing Secret Values in Transient Tree State

Authentication state should always be stored in the `TreeContext` `sharedState` object, which is then passed in as a parameter to the `process` method. AM ensures that this shared state is made available to downstream nodes.

To avoid storing sensitive information, such as passwords, a distinction is made between authentication state which can be written to CTS or to a JWT, and state which should only be available for the duration of the current HTTP request.

Sensitive information, such as passwords, should be stored in the `TreeContext` `transientState`.

Important

Note that `sharedState` and `transientState` should be updated by using the `replaceSharedState` and `replaceTransientState` methods on `Action.Builder` respectively. For example:

```
Action action = goToNext()
  .replaceTransientState(context.transientState.copy().put(PASSWORD, pwd))
  .build();
```

Accessing 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' `sharedState`, 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 read an identity's profile from a realm, 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.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

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(ISAuthConstants.AUTH_SERVICE_NAME,
            AccessController.doPrivileged(AdminTokenAction.getInstance()))
        .getOrganizationConfig(realm);

    Set<String> realmAliasAttrs = serviceConfig.getAttributes()
        .get(ISAuthConstants.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 neccessary.

Reading Values from an Identity's Profile

- To read individual specific attribute values from a profile, after obtaining the profile, use the `AMIdentity#getAttribute(String name)` method.

Writing Values to an Identity's Profile

- To 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();
  ```

Creating and Managing Callbacks

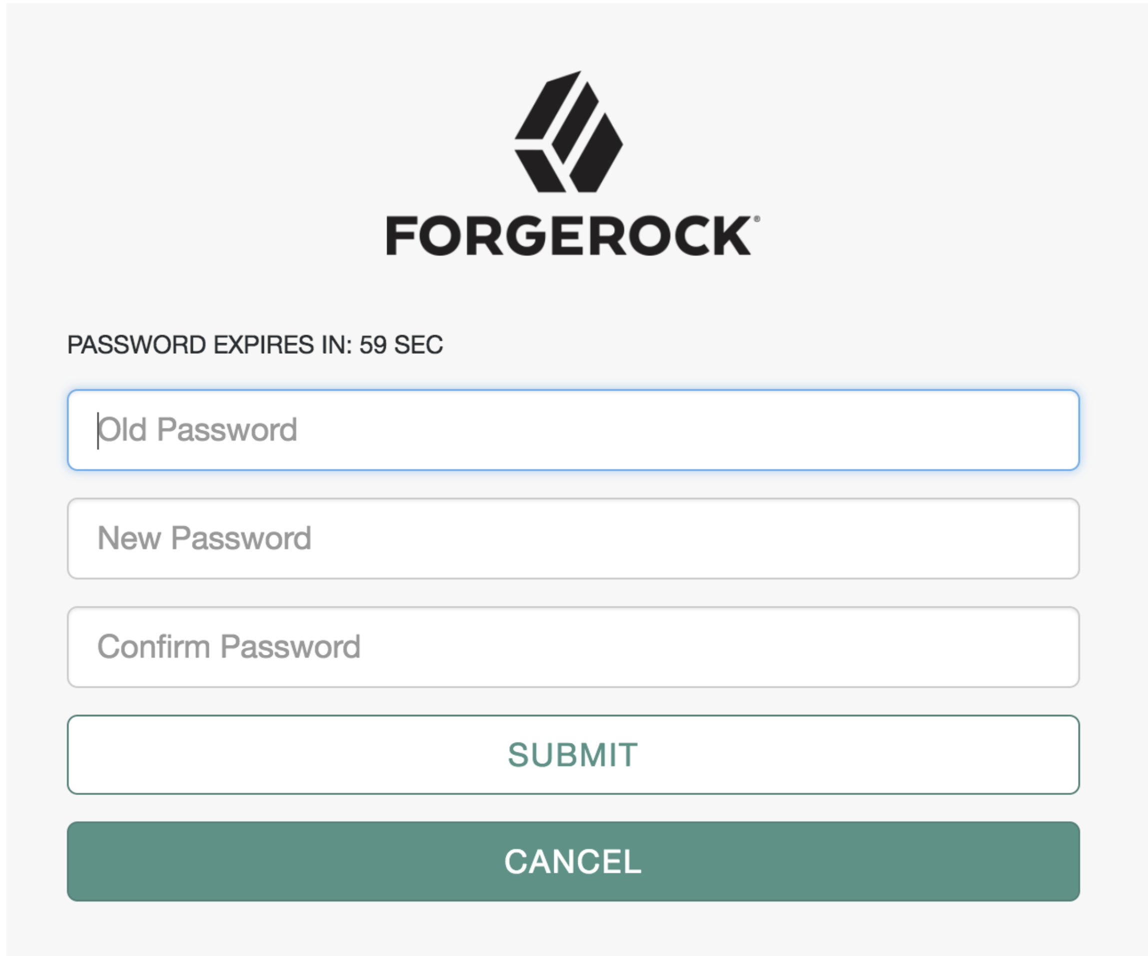
Callbacks are the method that nodes use to obtain interaction with the authenticating user.

Calling the `getCallbacks()` method on a `TreeContext` - the sole argument to the `process()` method of a node - returns a list of all the callbacks that have just been displayed in the UI.

Callbacks must implement the `javax.security.auth.callback.Callback` interface, although there are already many convenient existing implementations in the package and you may not need to create your own.

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

Copyright © 2018-2020 ForgeRock AS. All rights reserved.



In order to process responses to callbacks, it is necessary to know which callback is at which position 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:

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

```
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.

Sending and Executing 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 favourite 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();
}
```

Handling Multiple Visits to the Same Node

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

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

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.

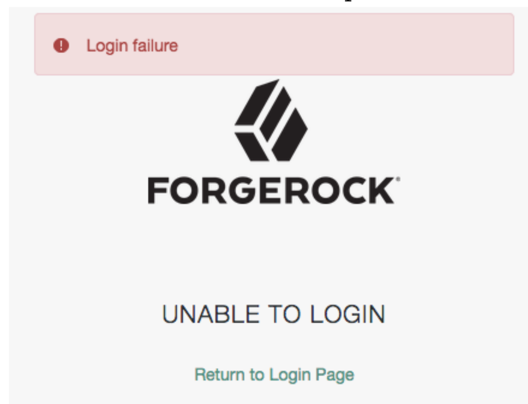
Handling Errors

This section 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:



Copyright © 2018-2020 ForgeRock AS. All rights reserved.

Unrecoverable Errors

When errors occur during node processing, for example an LDAP server connection is down, throw a `NodeProcessException` exception, with the desired message.

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*".

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

Chapter 4
The 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 then 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 { ❶

  private static String currentVersion = "1.0.0"; ❷

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

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

Key:

❶ Name the plugin class after the core class, and append *Plugin*, for example `MyCustomNodePlugin`.

❷ Provide a version number for the authentication node.

❸ 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.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

`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 6.5.5 Public API Javadoc*.

Upgrading Nodes and Configuration Changes

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:

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

1. 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.

   Any instances of nodes with version `0.0.0` that are used within trees will need to be removed and re-inserted into the affected trees, so that the updated schema is applied and available to those tree.

2. 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, see "Upgrading Simple Node Configuration Schema Changes" and "Upgrading Complex Node Configuration Schema Changes".

Upgrading 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 the type of an attribute 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;
    }
  }
  // ...
}
```

Upgrading 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.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

When configuration schema changes are complex, use the API provided in the `com.sun.identity.sm` package, as detailed below.

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()) {
```


Copyright © 2018-2020 ForgeRock AS. All rights reserved.

```
      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 | SMSException | RealmLookupException e) {
      throw new PluginException("Could not upgrade", e);
  }
  super.upgrade(fromVersion);
}
```


Copyright © 2018-2020 ForgeRock AS. All rights reserved.

Chapter 5
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.

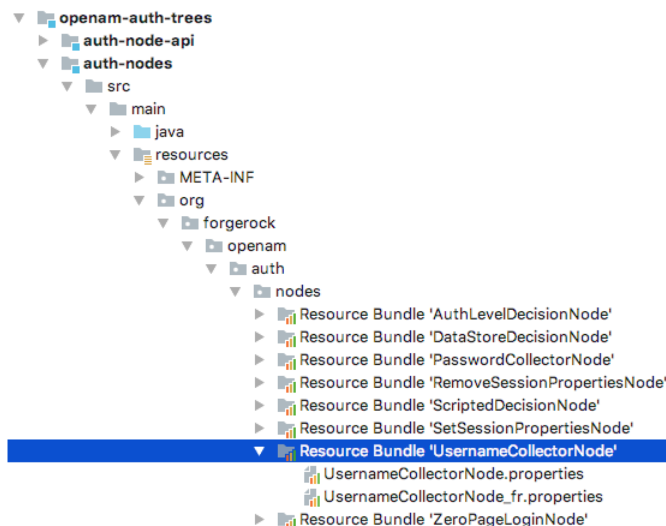
To Localize Node UI Text

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`

   Example Resource Bundle



Copyright © 2018-2020 ForgeRock AS. All rights reserved.

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();
```


Copyright © 2018-2020 ForgeRock AS. All rights reserved.

Chapter 6
Building and Installing

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

To Build and Install Custom Authentication Nodes

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-6.5.5.jar`.

3. Copy the `.jar` file to the `WEB-INF/lib/` folder where AM is deployed:

   ```
   $ cp auth-nodes-6.5.5.jar /path/to/tomcat/webapps/openam/WEB-INF/lib/
   ```

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

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:

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

Custom Node within a Tree

For more information on using the tree designer to manage authentication trees, see "Configuring Authentication Trees" in the *Authentication and Single Sign-On Guide*.

For information on upgrading custom nodes, see "Upgrading Nodes and Configuration Changes".

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

Chapter 7

Maintaining Authentication Nodes

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

Testing

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 which have a high percentage of code coverage. The low amount of static dependencies mean 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.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

> **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.

Debugging

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 - over use of the `error` or `warning` level can cause debug logs to fill, and even affect performance 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);
```


Copyright © 2018-2020 ForgeRock AS. All rights reserved.

Auditing

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.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

Monitoring

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 "Monitoring Services" in the *Setup and Maintenance Guide*.

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

- Meter Node
- Timer Start Node
- Timer Stop Node

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

Chapter 8
Troubleshooting

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

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

A: 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.

Q: How do I get new attributes to appear in the node after the service has been loaded once?

A: See "Upgrading Nodes and Configuration Changes".

Q: What type of exception should I throw so that the framework handles it gracefully?

A: 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 "Handling Errors".

Q: Do I need multiple projects/jars for multiple nodes?

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

See "*Building and Installing*".

Q: What ForgeRock utilities exist for me to use to assist in the node building experience?

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

See the *AM 6.5.5 Public API Javadoc*.

Q: Transient State vs Shared State - When should I use one or the other?

A: Transient state is used for secret values that should not persist. See "Storing Values in Shared Tree State".

Q: 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?

A: See "Accessing an Identity's Profile".

Q: Where do I go for examples of authentication nodes?

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

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

Examples of community nodes written by third parties may be found on the ForgeRock Marketplace.

For source access to the authentication nodes builtin to AM, see How do I access and build the sample code provided for AM/OpenAM (All versions)? in the *ForgeRock Knowledge Base*.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

Appendix A. Getting Support

ForgeRock provides support services, professional services, training through ForgeRock University, and partner services to assist you in setting up and maintaining your deployments. For a general overview of these services, see https://www.forgerock.com.

ForgeRock has staff members around the globe who support our international customers and partners. For details on ForgeRock's support offering, including support plans and service level agreements (SLAs), visit https://www.forgerock.com/support.

ForgeRock publishes comprehensive documentation online:

- The ForgeRock Knowledge Base offers a large and increasing number of up-to-date, practical articles that help you deploy and manage ForgeRock software.

  While many articles are visible to community members, ForgeRock customers have access to much more, including advanced information for customers using ForgeRock software in a mission-critical capacity.

- ForgeRock product documentation, such as this document, aims to be technically accurate and complete with respect to the software documented. It is visible to everyone and covers all product features and examples of how to use them.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

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.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

	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

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

	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

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

	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.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

	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.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.

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.

Copyright © 2018-2020 ForgeRock AS. All rights reserved.