nevisAuth

Developer Guide

nevisAuth

Developer Guide

Software Version: 4.25.12.1, Date: 21.07.2022

Table of Contents

1. Introduction

nevisAuth is a middleware product of the NEVIS Security Suite. The product provides authentication and authorization services. nevisAuth supports authentication against many back ends, interactive and non-interactive logins and multi-step authentication.

To cover a wide variety of authentication mechanisms, nevisAuth comes bundled with many plug-ins. In addition to those, you can implement custom plug-ins in Java. This allows you to incorporate special requirements or to integrate additional services.

This document is targeted at people who would like to understand and develop new plug-ins. The focus is currently on authentication plug-ins, which allow to integrate new authentication mechanisms into nevisAuth. An authentication plug-in consists of one or multiple related AuthStates, which are the building blocks of an authentication flow.

The **`Requirements`** chapter lists the basic requirements which must be fulfilled to successfully develop authentication plug-ins. The **`Quickstart`** chapter helps you to quickly set up and deploy a new authentication plug-in without going into specific details. The subsequent **`Basic Concepts`** chapter gives an overview of the nevisAuth architecture and looks behind the curtain to help you understand the internals of nevisAuth. The **`Developing AuthStates`** chapter covers everything that is important to program custom authentication plug-ins. Finally, the **`Plug-in Deployment`** chapter contains information about what must be considered when deploying a custom plug-in.

2. Requirements

To develop authentication plug-ins, the following software must be available in your development environment:

- Linux (recommended)
- JDK 1.8 (must)
- nevisAuth SDK (must)
- Maven 3 (must to be able to use the provided project archetypes)

Furthermore, we recommend using an integrated development environment such as Eclipse or IntelliJ. To be able to test your authentication plug-in, you must have access to a NEVIS integration environment which must consist of at least nevisProxy, nevisAuth and nevisLogrend.

3. Quickstart

The nevisAuth SDK is the starting point for the development of custom authentication plug-ins. It is part of every nevisAuth installation and is located at

```
/opt/nevisauth/resources/nevisauth-sdk-<nevisAuth-version>.tar.gz
```

The SDK archive contains all resources required to develop authentication plug-ins for nevisAuth. A Maven archetype is provided to enable fast bootstrapping of nevisAuth authentication plug-in projects. The archetype assists you with the creation of a Maven POM file that already contains the required dependencies to the nevisAuth API and commons library.

The structure of the nevisAuth SDK is as follows:

- **archetypes:** This folder contains a Maven archetype for authentication plug-in projects.
- **docs:** This folder contains documentation for developers.
- **libs:** This folder contains the official API libraries and the corresponding Javadocs.
- **scripts:** This folder contains scripts to set up the SDK.
- **resources:** This folder contains files that are required for the integration of the example authentication plug-in.

The fastest way to get started with your own nevisAuth authentication plug-in is to generate a new Maven project based on the archetype.

3.1. Creating your first Authentication Plug-in

As the first step install the provided Java libraries and the Maven archetype in the Maven repository of the development workstation. If you are working in a Unix/Linux environment, you can automate this by executing the provided script:

```
./scripts/add-dependencies-to-maven-repo.sh
```

Otherwise, you can manually install the necessary Maven dependencies by executing the following commands:

```
mvn install:install-file -Dfile=./libs/nevisauth-authstate-api-<nevisAuth-version>.jar
-DgroupId=ch.nevis.nevisauth -DartifactId=nevisauth-authstate-api -Dversion=<nevisAuth
-version> -Dpackaging=jar
mvn install:install-file -Dfile=./libs/nevisauth-commons-<nevisAuth-version>.jar
-DgroupId=ch.nevis.nevisauth -DartifactId=nevisauth-commons -Dversion=<nevisAuth-version>
-Dpackaging=jar
cd archetypes\AuthPlugin
mvn clean install
```

This will install the required nevisAuth API libraries and the Maven archetype in your local Maven respository.

3.1.1. Creating a Maven Project

To create a new authentication plug-in project, change to the directory where you would like your project to be located:

```
cd ~/projects
```

Then execute:

```
mvn archetype:generate \
 -DarchetypeCatalog=local \
 -DarchetypeGroupId=ch.nevis.nevisauth \
 -DarchetypeArtifactId=auth-plugin-archetype \
 -DarchetypeVersion=<nevisAuth-version> \
 -DgroupId=ch.example \
 -DartifactId=auth-state-example \
 -DinteractiveMode=false
```

As a result, a project with the name `auth-state-example` is created. The resulting Maven project can be imported to various IDEs, e.g., Eclipse and IntelliJ.

Projects created from the archetype already contain an example AuthState. The example AuthState implements authentication by checking the incoming user name and password against a user name/password hash list in a file.

When implementing your own custom plug-in, it is recommended to create the project based on the archetype. After creation, you can replace the example AuthState by your own implementation.

3.2. Packaging and Deploying the Plug-in

The authentication plug-in can be built with the following Maven command:

```
cd auth-state-example
mvn clean package
```

To use the AuthState contained in the authentication plug-in, extract the resulting `ZIP archive` from the `target` directory of the **auth-state-example** directory to the `plugin` directory of a nevisAuth instance.

For example, if your nevisAuth instance is named `default` and runs on the same machine, you can extract it by executing the following command (Note: You may need root access to extract it to this directory):

```
unzip target/auth-state-example-1.0-SNAPSHOT.zip -d /var/opt/nevisauth/default/plugin/
```

To configure nevisAuth to use the AuthState, open the configuration by typing

```
nevisauth config
```

In the XML config, add the following `AuthState` elements below the `Domain` element:

```
<AuthState name="UseridPasswordState" class="ch.example.UseridPasswordFileAuthState"
        classPath="/var/opt/nevisauth/default/plugin/auth-state-example-1.0-SNAPSHOT"
        classLoadStrategy="PARENT_LAST">
    <ResultCond name="ok" next="AuthDone"/>
    <ResultCond name="failed" next="AuthError" />
    <Response value="AUTH_CONTINUE">
      <Gui name="AuthUidPwDialog" label="login.test.label">
        <GuiElem name="lasterror"    type="error"   label="${notes:lasterrorinfo}" value
="${notes:lasterror}"/>
        <GuiElem name="info"         type="info"    label="login.test.text"/>
        <GuiElem name="username" type="text"    label="userid.label" value=
"${notes:loginid}"/>
        <GuiElem name="password" type="pw-text" label="password.label"/>
        <GuiElem name="submit" type="button"        label="submit.button.label" value=
"Login"/>
      </Gui>
    </Response>
    <property name="passwordFileLocation" value=
"/var/opt/nevisauth/default/conf/passwords.txt"/>
</AuthState>

<AuthState name="AuthDone" class="ch.nevis.esauth.auth.states.standard.AuthDone">
  <Response value="AUTH_DONE">
    <Gui name="AuthDoneDialog"/>
  </Response>
</AuthState>

<AuthState name="AuthError" class="ch.nevis.esauth.auth.states.standard.AuthError">
    <Response value="AUTH_ERROR">
        <Gui name="AuthErrorDialog"/>
    </Response>
</AuthState>
```

Make sure the `UseridPasswordState` is referenced by a `Domain` element, e.g.,

```
<Domain name="SSO_TEST" default="true"
  reauthInterval="0"
  inactiveInterval="1800">
  <Entry method="authenticate" state="UseridPasswordState"/>
</Domain>
```

Copy the user credential file from the resources directory into the nevisAuth configuration directory:

```
cp resources/passwords.txt /var/opt/nevisauth/default/conf/passwords.txt
```

The file contains the following credentials for testing the setup:

user name	password
testuser1	password1
testuser2	password2
testuser3	password3

Now, you are ready to restart nevisAuth to read the new configuration:

```
nevisauth restart
```

That's it. You have successfully deployed an AuthState of the type **UseridPasswordFileAuthState** and can try to log in, using the credentials defined in the password file.

4. Basic Concepts

This chapter introduces you to the main concepts of nevisAuth, starting with an overview of the nevisAuth architecture, descending into nevisAuth's core - the `AuthEngine` - and giving insight into the building blocks of an authentication flow - the `AuthStates`. At last, we present how and when a response is generated.

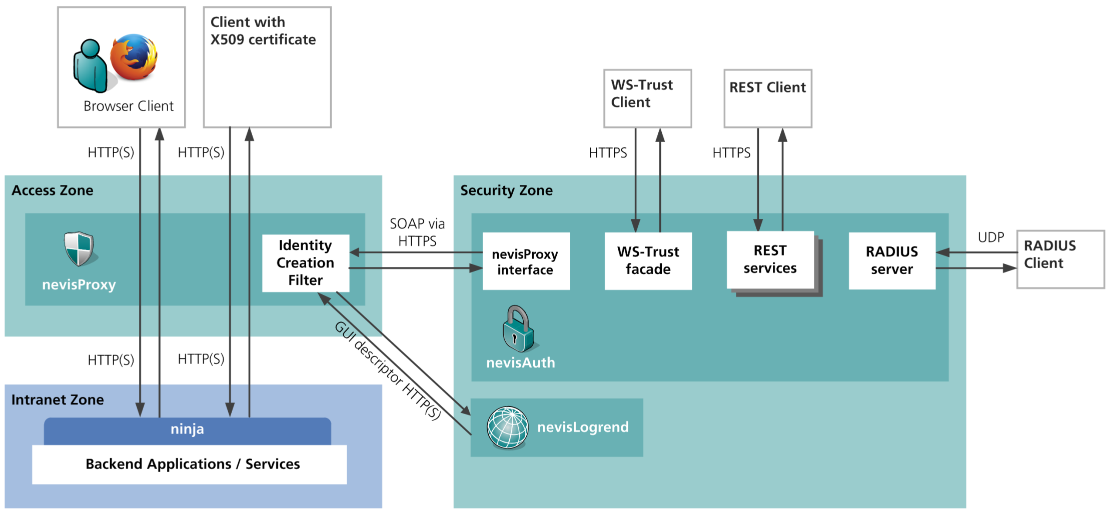
The concepts discussed in this chapter should give you the basic knowledge required for developing your own custom authentication plug-ins.

4.1. nevisAuth Architecture

nevisAuth is a middleware product that provides authentication and authorization services. It supports many authentication back ends and can be configured flexibly to guide the end user through one or more authentication steps.

nevisAuth's clients rely on nevisAuth to authenticate end users. To support different clients, nevisAuth provides different interfaces on the server side. The most common nevisAuth client is nevisProxy, which interacts with nevisAuth through the nevisProxy interface. Other interfaces are, for example, the WS-Trust security token service, which supports WS-Trust clients, the RADIUS service, which provides authentication for RADIUS clients, and the REST services, which can be accessed with any HTTP-based clients.

The following diagram gives an overview of the nevisAuth interfaces and the NEVIS products that are typically involved in the authentication process within the NEVIS Security Suite.



Upon successful authentication of the end user, nevisAuth issues a `proof of authentication` to the client. Depending on the interface through which the request is received, the proof of authentication can take on different forms.

Since most setups with nevisAuth use nevisProxy as a client, the next chapter will familiarize you with specifics about the nevisProxy interface and the proof of authentication issued to nevisProxy.

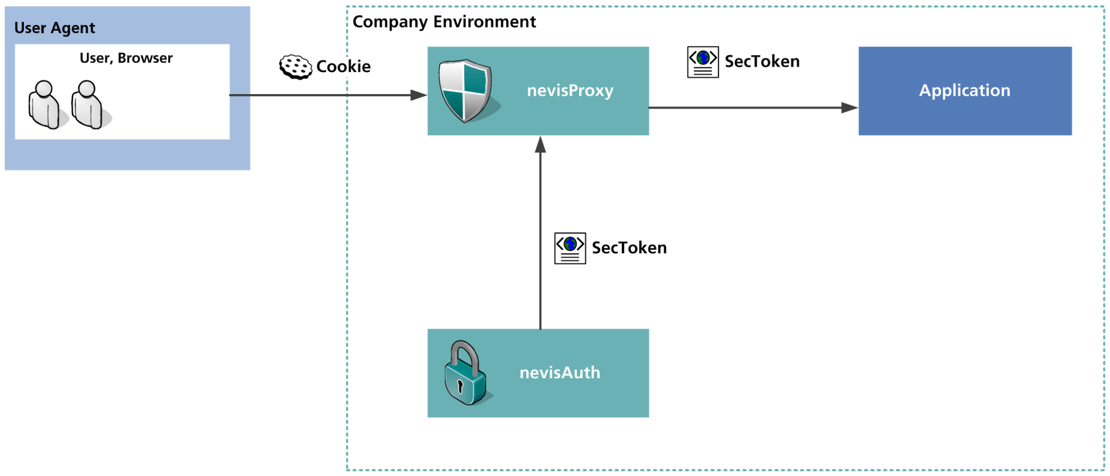
4.1.1. nevisProxy as a Client to nevisAuth

In most cases, nevisAuth is accessed via nevisProxy through the identity creation filter or the security role filter (see the nevisProxy reference guide for details). In this setup, nevisProxy acts as an interceptor for requests to the back-end application(s). Before a request is forwarded to the desired location, the identity creation filter checks whether a user requires authentication and the security role filter verifies that the authorization is sufficient. In case authentication or authorization is required, nevisProxy wraps the original HTTP request, which it received from the user agent, into a SOAP request which will be sent to nevisAuth. GET and POST parameters are forwarded as so called `inArgs` (incoming arguments). Cookies, language settings, IP address of the user agent and other HTTP headers are forwarded as `inCtx` (incoming context).

The request is received by nevisAuth’s nevisProxy interface and processed by the `AuthEngine`. After the request is processed, a response is prepared and sent back to nevisProxy. At that point, nevisProxy decides whether a GUI must be rendered to request more information from the user (e.g., user credentials), whether the authentication process resulted in an error and the request to the protected application must be prohibited, or whether the authentication or logout process was successful.

In case the authentication process was successful, nevisProxy expects a `proof of authentication` from nevisAuth in the form of a security token - a `SecToken`. nevisProxy verifies the content and signature of the SecToken and upgrades the user agent session to an authenticated session. If the verification is successful and the necessary authorization is granted, nevisProxy will allow the initial request to be forwarded to the protected application.

By passing the SecToken to back-end applications, these, in turn, are able to verify the authentication and retrieve the forwarded user identity. The user agent does not receive the SecToken, but instead associates the session with nevisProxy through an HTTP cookie. The following diagram shows the identity passing between the components:



The SecToken is a structured and cryptographically signed XML document that holds user data such as the name and the roles. It enables the receiver to verify that the signed data has not been changed and was generated by a trusted entity. The extensible design of the token also allows to include additional, site-specific attributes that are available during the (site-specific) authentication process.

The security token also holds an expiration time, after which access to the session will no longer be granted and the session will eventually terminate. The proprietary data structure of the SecToken encodes the information configured in the nevisAuth configuration. This typically includes information such as the session ID, user ID, login ID, level of authentication, roles and a timestamp when the signature was computed.

The signature, signature timestamp and session ID are generated by the AuthEngine when the authentication is successful. User ID, login ID and roles must be set in the AuthStates during the authentication flow. The level of authentication is configured in the nevisAuth configuration and set by the AuthEngine if a certain authentication process is passed.

4.2. Service Operations

nevisAuth supports four different authentication and authorization operations: `authenticate`, `stepup`, `unlock` and `logout`. The following table summarizes their functionality:

Operation	Description
authenticate	This operation is called by the client in case the client does not know who the user is yet. The purpose of this operation is to identify and authenticate the user.
stepup	This operation is called by the client in case the client already knows who the user is, but the user requires additional roles to access the desired resources.
unlock	This operation is called by the client in case it requires re-authentication of the user by nevisAuth.
logout	This operation is called by the client upon a logout request initiated by a user.

The AuthEngine processing might differ depending on which operation was triggered by the client. For example, at the end of an authenticate operation, the initial user session is upgraded to an authenticated session, while at the end of a logout operation, the user's session is destroyed.

4.3. AuthEngine

The AuthEngine is responsible for processing authentication and authorization requests. The authentication process starts as soon as a request is received by nevisAuth through one of the nevisAuth interfaces. The AuthEngine transforms the request into a context object and passes it through a set of authentication process steps - the AuthStates. The AuthEngine also identifies whether more information from the user is required, which usually results in the generation of an HTML form. Eventually, the AuthEngine constructs a final response for nevisProxy, which either indicates an error or a successful authentication, session upgrade or logout.

The AuthEngine creates, updates and removes sessions of users. The sessions hold data that are required during the authentication flow, or, beyond that, during a later retrieval of roles or other user attributes. In nevisAuth, sessions are either initial or authenticated. Initial sessions are short-living. They hold data of users who did not yet finish the authentication. Whenever a user finishes an authentication flow in a successful final state, the initial session is upgraded to an authenticated session. On the other hand, if an authentication flow is finished in an error state, the initial session is discarded.

In the following chapters, we introduce the concept of the AuthStates, discuss when and which responses are generated by the AuthEngine and when and how HTML forms are generated.

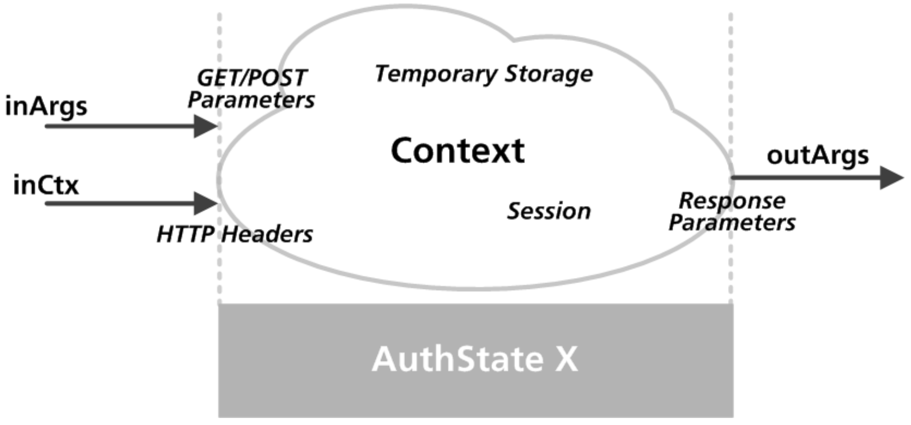
4.4. AuthStates

`AuthStates` are the **building blocks** of the authentication process. Each AuthState has one particular responsibility, e.g. to authenticate a user against a certain authentication back end, to manipulate a request, to cache information, etc. They can be chained together in a directed graph to allow complex configurations for authentication processing.

AuthStates can be separated into two groups: those that are actually authenticating a user and those that support the processing of the request. In a valid nevisAuth setup there must be at least one authenticating AuthState, which is responsible for setting the user id, after it has been verified, into the response.

4.4.1. Data Manipulation in and out of Context

An AuthState has access to the authentication request's context and manipulates it during processing. The context consists of the `inCtx` and `inArgs`, received from the client, the `session` and temporary storage, which can be used to cache information for later usage, and response parameters, which are eventually integrated into the response to the clienti (`outArgs`).



Variables in nevisAuth can be stored and retrieved from different sources. Depending on the source, the lifetime and accessibility of the data varies. While data is usually stored in a user request's context, AuthStates can also store data outside of a user request's context (readable by anybody, e.g., in the background without having a request object available). For that, the out-of-context data service can be used.

The following table lists valid nevisAuth variable sources.

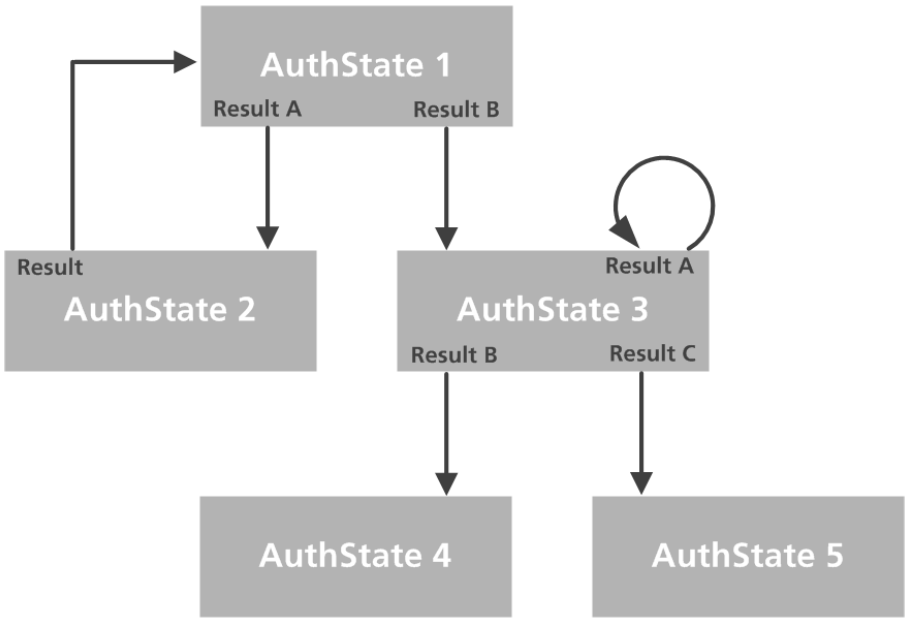
Variable Sources

Source	Description
inargs	The GET and POST parameters forwarded by the client.
inctx	The client's login context variables, such as HTTP headers, required roles, requested language, client address, etc.

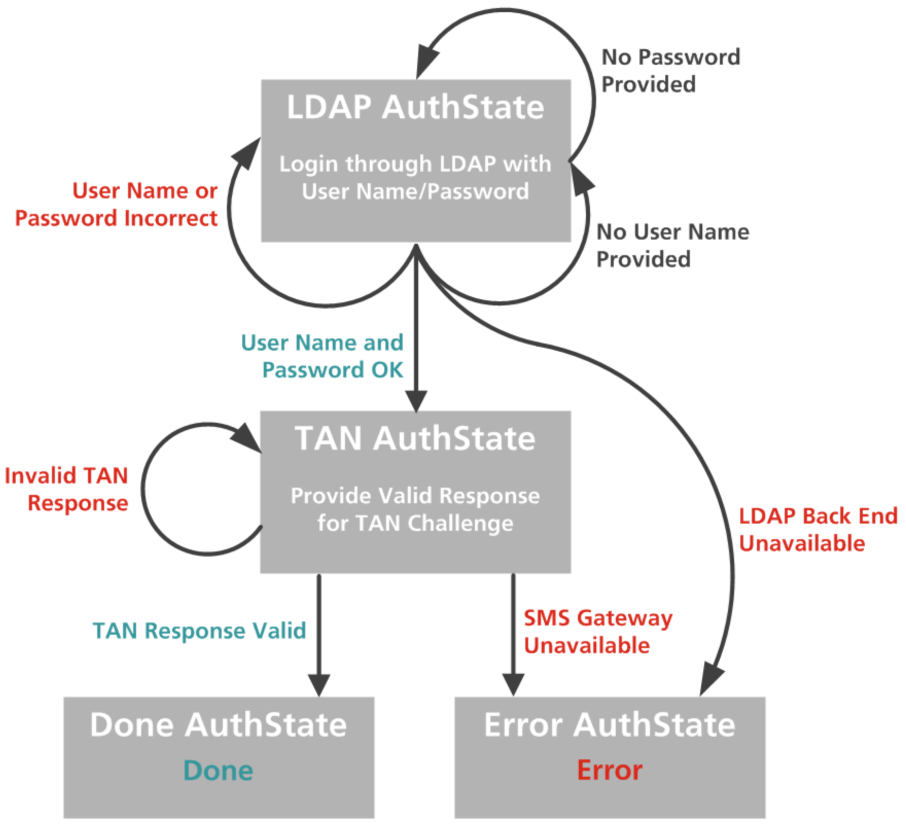
Variable Sources	
sess/session	The session variables. They live as long as the user's session.
notes	Temporary variables. They live only until the next response.
outargs	Variables forwarded to the nevisAuth client (e.g., nevisProxy) for propagation to the back-end application or forward-propagation to the user agent.
request	The request attributes that the AuthEngine supplies. Includes data such as required roles, as well as the language that was chosen for this user agent.
response	The response attributes that previous AuthStates supply. Contains authentication and authorization data, such as user names and roles.
litdict	All variables stored in the literal dictionary files. The value is retrieved according to the currently set language (read-only).
cookie	Pseudo-source for accessing cookies sent by the user agent (read-only).
header	Pseudo-source for reading HTTP headers sent by the user agent (read-only).
oocds	Out-of-context data store. Values read from or written into this source are outside of the user's context, which means that anybody can access them.

4.4.2. Request Processing

In mathematical terms, the request processing model represents a finite state machine of AuthStates. In a correctly set-up instance of nevisAuth, there is always a non-empty set of AuthStates, which is available to process requests. There is always an entry state, at which the processing starts, and one or multiple states at which the processing ends. Each state produces a result.



For a practical example, the following setup shows what a login by user name / password with two-factor authentication could look like:



In this example, the request processing would start at the **LDAP** AuthState, which is marked as the entry state of the AuthState flow. If the user name and password were successfully checked against the LDAP back end, the flow continues at the TAN AuthState. Otherwise the GUI form is rendered (again) and may show an error message indicating that the input was either missing or incorrect.

Continuing at the **TAN** AuthState, nevisAuth initiates the transmission of an SMS to the user's mobile phone, containing a transaction authentication number (TAN). Another GUI form is rendered, prompting the user to provide the TAN in order to prove that he is in control of the claimed mobile phone number.

If the provided TAN equals the sent TAN, the authentication process is successful and the final AuthState **Done** is reached. Otherwise, if the provided TAN was not correct, the GUI form is rendered again and the processing remains in the TAN state. In case a system error occurs, e.g., if the SMS gateway is unavailable, the final AuthState **Error** is reached. In this case the outcome of the processing can not be influenced by the user and therefore the authentication processing must be aborted.

4.4.3. Entry AuthState and Transition Configuration

Once a nevisAuth interface receives a request, the AuthEngine must decide with which AuthState it should start processing. This decision is made based on the configuration and whether a session with the user already exists or not.

If a session already exists, and the previous request did not result in a completed authentication process, the request will be passed to the AuthState that previously failed to continue processing. One reason for previously failing might be that user input was required.

If no session exists, the request processing will start at the configured entry AuthState. The AuthEngine therefore checks whether the configuration contains an entry for the called service operation. As you can see in the configuration below, the service operation is specified by the property 'method' of the 'Entry' XML element. This element is used to identify the first AuthState. In this example, the request is passed to the AuthState **`MyCustomState`** if no session exists and 'authenticate' is called, and to the AuthState **`AnotherState`**, if no session exists and 'unlock' is called.

If a service operation is called, but not configured, the AuthEngine falls back to the entry AuthState that is configured with the 'authenticate' method.

```
<Domain name="SSO_TEST" default="true"
    reauthInterval="0"
    inactiveInterval="1800">
  <Entry method="authenticate" state="MyCustomState"/>
  <Entry method="unlock" state="AnotherState"/>
</Domain
```

To navigate from one AuthState to the next during a request, AuthState transitions are made. Therefore, each AuthState produces a `result`, based on the incoming data and its internal logic. The result of the AuthState is used (among other things) by the AuthEngine to decide which AuthState should process the request next.

The result itself is an arbitrary string value programmatically set by the processing AuthState. The AuthEngine will look in the current AuthState's configuration to find the next AuthState to process the request. The next AuthState is referenced through a name.

A set of triggering conditions helps the AuthEngine decide which AuthState comes next in the process. These triggering conditions are defined in the ResultCond element. Such a configuration might look like this:

```
<AuthState name="MyCustomState" class="ch.my.custom.MyAuthState" final="false">
    <ResultCond name="ok" next="AuthDone"/>
    <ResultCond name="failed" next="AuthError"/>
    ...
</AuthState>
```

Let's assume our AuthState sets the result 'ok'. This prompts the AuthEngine to pass the request to the AuthState with the name `AuthDone` after completing processal with the current AuthState. On the other hand, if the AuthState sets the result 'failed', `AuthError` will be selected as the next AuthState.

There is one implicit transition that does not require (but allows) configuration: `default`. If the result `default` is set by an AuthState, but no transition is configured, the AuthEngine stops processing to prepare a response, possibly indicating that a GUI must be rendered.

4.4.4. AuthState Lifecycle

Every time an AuthState is configured using a `<AuthState>` element, nevisAuth creates exactly one instance of that AuthState. This is done during start-up by instantiating and initializing the AuthState. I.e., AuthStates are multithreaded and nevisAuth shares single instances of AuthStates for multiple requests. In other words, nevisAuth does not create a new instance of the AuthState class for each request. Instead, it reuses the existing ones.

4.5. Response Generation

In the previous chapters we learned that a request is received through one of nevisAuth's interfaces, and passed through the configured AuthStates. A request is processed as long as possible, until it gets stuck in a situation in which it cannot be processed any more. This happens whenever a request hits an AuthState that is configured to show the GUI before processing, or if the AuthState transitions to itself. This occurs, for example, if the user enters a wrong password. At that point, a response is generated by the AuthEngine and handed to the respective nevisAuth interface.

There are three types of responses:

1. `AUTH_ERROR`: Indicates that the AuthEngine or AuthState has detected an error and no user input can circumvent this error.
2. `AUTH_DONE`: Indicates that the AuthEngine has finished the authentication successfully and a SecToken was issued.
3. `AUTH_CONTINUE`: Indicates that nevisAuth requires more input from the user to be able to continue processing.

Depending on the nevisAuth interface the response is handed to, it is mapped to a response according to that interface’s protocol.

4.5.1. GUI Generation

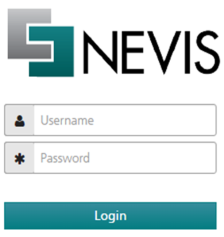
As a general rule, a GUI is always generated if an AuthState responds with an **`AUTH_CONTINUE`** or **`AUTH_ERROR`** response. nevisAuth does not generate GUIs itself. Instead it builds a **`GUI descriptor`** which is sent back to the client. If the client is nevisProxy, it in turn forwards it to nevisLogRend, the login renderer application. The login renderer then responds with an HTML document that implements the GUI described in the GUI descriptor. This rendered GUI will then be forwarded by nevisProxy to the end-user’s user agent.

The GUI descriptor is configured in the AuthState configuration as part of the response element of an AuthState. It guides the login renderer to create a certain GUI by describing the content, but not the layout, of the GUI.

The following code snippet shows an example of the GUI descriptor as part of an AuthState configuration.

GUI Descriptor

```
<Gui name="LoginPage" label="UIDPWDialog">
   <GuiElem name="isiwebuserid" type="text" label="Username" value="" />
   <GuiElem name="isiwebpasswd" type="pw-text" label="Password" value="" />
   <GuiElem name="submit" type="button" label="Login" value="Login" />
</Gui>
```



On the left side, you can see the GUI descriptor in the XML configuration, with the three GUI elements for the user name, password and login button. On the right side you can see how those GUI elements are rendered by nevisLogRend.

5. Technology Stack

nevisAuth is a web application written in Java. We are using the following technologies:

Technology	Version	Description
Linux		Operating System
Java	1.8	Compile and runtime environment. As a runtime environment, newer Java versions can (and are) used, because the JVMs are typically backwards compatible.
Java Servlet API	2.3	The Java Servlet API specifies how HTTP requests are received and how HTTP responses are sent back to the client. nevisAuth implements filters and servlets to handle such requests.
adnjboss	5.1.6.1	The web application server on which the nevisAuth web application is deployed.
wildfly	8	An alternative web application server on which the nevisAuth web application can be installed.
Jax-WS	from JRE 1.8	The web service stack used to retrieve SOAP web service requests and respond with SOAP responses.
Jax-RS	2.6	The web service stack used to accept and respond to REST web service calls.

6. Developing AuthStates

Now that we are familiar with the most important concepts of nevisAuth, we can dive into the actual development.

6.1. Prerequisites

When writing an authentication plug-in, we strongly recommend starting with a Maven project, generated using the archetype provided in the nevisAuth SDK. This will set the correct dependency versions and scopes. How to generate a Maven project based on the provided archetype is described in the chapter Quickstart.

Two libraries are officially delivered as part of the nevisAuth SDK. `nevisauth-authstate-api` and `nevisauth-commons`. The classes contained in `nevisauth-authstate-api` are always provided in a nevisAuth runtime environment. `nevisauth-authstate-api` is thus declared with scope `provided` in Maven projects generated from the archetype. It does not have to be deployed together with the authentication plug-in. Unlike the classes in `nevisauth-authstate-api`, the classes in `nevisauth-commons` are not mandatory to develop an authentication plug-in. But they offer some useful functionality which simplifies development of authentication plug-ins. They are not necessarily provided in a nevisAuth runtime environment and thus are declared with scope `compile` in Maven projects generated from the archetype. `nevisauth-commons` must be deployed together with the authentication plug-in.

Do not use any classes in `ch.nevis.*` or `ch.adnovum.*` other than the classes contained in the libraries officially delivered as part of the nevisAuth SDK. Other classes and libraries are considered to be internal and are subject to change without further notice.

6.2. AuthState Basic Structure

An AuthState is a Java class, which inherits from the abstract class `ch.nevis.esauth.auth.engine.AuthState`. During nevisAuth start-up, for each configured AuthState the `init()` method is called. The AuthState properties are passed from the configuration file to the method, which allows the initialization of the AuthState with configured values.

As mentioned in the chapter Service Operations, there are four operations for different processing contexts: authenticate, stepup, unlock and logout. These operations are also reflected in the AuthState super class as Java methods and can be overriden one-by-one. These methods are called from the `process()` method, which checks which service operation is currently ongoing and then calls the matching implementation. Since, most of the time, an AuthState does not have to differentiate between different operations, it is common practice to override the `process()` method itself.

The `request` and `response` objects constitute the context and are passed as arguments to the AuthState `process()` method.

The minimal structure of the AuthState looks like this:

```
public class MyAuthState extends AuthState {

    @Override
    public void process(AuthRequest request, AuthResponse response) throws
AuthStateException {
        Context context = new Context(request, response);
        // ...
    }

}
```

As described in AuthState Lifecycle, AuthStates are multithreaded. I.e., `process()` is called for every request with a different thread. This means that `process()`, `authenticate()`, `stepup()`, `logout()` and `unlock()` should never override any instance variables, only consume them.

6.3. Reading and Manipulating the Context

The request and response objects, which are passed to the `process()` method, define the context and are used to evaluate whether the user can be authenticated and authorized. By wrapping the incoming request and response objects in a **`Context`** object, you will have access to multiple convenience methods:

```
Context context = new Context(req, res);
```

Let us assume you want to check whether a correct user name and password were provided by the input arguments. You can access the input arguments through the context by referencing the source **`inArgs`** as follows:

```
public class MyAuthState extends AuthState {
    @Override
    public void process(AuthRequest request, AuthResponse response) throws
AuthStateException {
        Context context = new Context(request, response);

        String uid = context.getValue(Scope.INARGS, "username");
        String pwd = context.getValue(Scope.INARGS, "password");
        // ...
    }
}
```

To manipulate the context, e.g., to store data in the session or in the temporary storage, you can use the **`putValue()`** method:

```
public class MyAuthState extends AuthState {
    @Override
    public void process(AuthRequest request, AuthResponse response) throws
AuthStateException {
        // ...
        context.putValue(Scope.SESSION, "returnUri", "https://www.service-provider.com");
        // ...
    }
}
```

6.4. AuthState Result

In the Request Processing chapter, we have learned that the AuthState produces results to help the AuthEngine decide which AuthState to process next. This is done by calling **`setResult()`** on the AuthResponse object.

Let's assume our AuthState sets the following results somewhere in the **`process()`** method. If the request is successful:

```
response.setResult("ok");
```

If it is not successful, it sets:

```
response.setResult("failed");
```

That means that two triggering conditions with the name **`ok`** and **`failed`** must exist in the AuthState configuration and must point to the next AuthState.

Let us revisit our example from the Basic Concepts chapter:

```
<AuthState name="MyCustomState" class="ch.my.custom.MyAuthState" final="false">
  <ResultCond name="ok" next="AuthDone"/>
  <ResultCond name="failed" next="AuthError"/>
  ...
</AuthState>
```

We see that if the AuthState sets **`ok`** as a result, the AuthEngine will decide to pass the request to **`AuthDone`** next. If the AuthState sets **`failed`** as a result, the AuthEngine will decide to pass the request to **`AuthError`** next.

6.5. AuthState Initialization

To allow the flexible use of AuthStates and re-use of code, AuthStates can be initialized with properties that are configured in nevisAuth's configuration file. For example, if you want to have a setup where the name of the input argument is configurable, you need to read the properties specified in the AuthState's configuration. By overriding the **`init()`** method of the abstract super class **`AuthState`**, you will have access to the configuration properties of that instance of the **`AuthState`**. Let's assume you want to read the following configuration:

```
<AuthState name="MyCustomState" class="ch.my.custom.MyAuthState" final="false">
    <ResultCond name="ok" next="AuthDone"/>
    <ResultCond name="failed" next="AuthError"/>
    <Response value="AUTH_CONTINUE">
      ...
    </Response>
    <property name="usernameKey" value="userid" />
</AuthState>
```

During start-up of nevisAuth, a new instance of **`MyAuthState`** is created and initialized with the property **`usernameKey`**.

You can read the config properties that were passed to the **`init()`** method, as the following example demonstrates. By wrapping the properties in a **`Configuration`** object, you will have access to a set of convenience methods.

```
private String usernameKey;

@Override
public void init(Properties cfg) throws BadConfigurationException {
    Configuration configuration = new Configuration(cfg);
    this.usernameKey = configuration.getPropertyAsString("usernameKey", "username");
}
```

In the above example, the variable from which the user name should be taken is determined. If no **`usernameKey`** was configured, the default value **`username`** is used. The variable can then be referenced from other methods like **`process()`**.

Please note that for every time your AuthState is configured using an **`<AuthState>`** element, nevisAuth creates only one instance of your AuthState. This is done during start-up by first instantiating the AuthState and then calling its **`init()`** method. This means that **instance variables in AuthStates are not thread-safe**. nevisAuth does not create a new instance of the AuthState class for each request. It reuses the existing one.

6.5.1. Variable Expressions

To make the AuthState configuration even more flexible, variable expressions can be used. Variable expressions can be written in nevisAuth's own variable expression language or with the Java Unified expression language (JUEL) [1: http://docs.oracle.com/javaee/6/tutorial/doc/gjddd.html].

For nevisAuth's own variable expression language, the syntax for accessing and possibly filtering or modifying values in nevisAuth is as follows:

```
${source:name[:filter[:pattern]]}
```

The source and name define which variable the AuthState loads. The filter and pattern may transform the value of the variable before loading it into memory. Filter and pattern may be undefined, in which case the output of the variable expression is the value of the referenced variable.

A filter may be defined in the form of a regular expression (as defined in http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html). If it is defined, the output of the variable expression will be:

- An empty string, if the value did not match the regular expression.
- The whole matching substring of the value, if the regular expression matches, but does not define any groupings.
- The pattern, if the value matches the regular expression and a pattern is defined.
- The value of the grouping, if the regular expression matches and defines a grouping.

JUEL is a more powerful tool for reading and manipulating variable values. For more details have a look at the tutorial (http://docs.oracle.com/javaee/6/tutorial/doc/gjddd.html) and the nevisAuth Reference Guide chapter **`Java EL expressions`**.

For nevisAuth AuthState development it is irrelevant whether nevisAuth's expression language or JUEL is used, because the mechanism for evaluating any of the expressions is the same.

AuthStates that want to allow the evaluation of variable expressions can pass the expression value to the **`evaluateExpression()`** method. For example, let's assume an AuthState allows the configuration of a property with the name "usernameKey". The same AuthState could be used to fetch the user name from the inargs (i.e., incoming POST or GET parameters) or from the notes scope (which is a short-living store for variables that previous AuthStates prepared). The two examples below show the differences in the configuration:

```xml
<AuthState name="MyCustomState" class="ch.my.custom.MyAuthState" final="false">
    <ResultCond name="ok" next="AuthDone"/>
    <ResultCond name="failed" next="AuthError"/>
    <property name="usernameKey" value="${inargs:username}" />
</AuthState>
```

```xml
<AuthState name="MyCustomState" class="ch.my.custom.MyAuthState" final="false">
    <ResultCond name="ok" next="AuthDone"/>
    <ResultCond name="failed" next="AuthError"/>
    <property name="usernameKey" value="${notes:username}" />
</AuthState>
```

The retrieval of the user name value is carried out by evaluating the configured variable expression. The following code shows how this can be achieved. The expression is stored as a field variable during initialization.

```
private String usernameKey;

@Override
public void init(Properties cfg) throws BadConfigurationException {
    Configuration configuration = new Configuration(cfg);
    this.usernameKey = configuration.getPropertyAsString("usernameKey",
"{inargs:username}");
    ...
}
```

In the `process()` method, you can then evaluate the `usernameKey` variable and use it to retrieve the user name.

```
@Override
public void process(AuthRequest req, AuthResponse res) {
    Context context = new Context(req, res);

    String uid = context.evaluateExpression(usernameKey);
    // ...
}
```

6.6. User Authentication

To mark a request as authenticated, the AuthState flow must reach an AUTH_DONE state at some point. When a request ends in AUTH_DONE, a security token is issued and, therefore, the user id and login id must exist in the session. In the example below, the user id and login id are set in the `AuthResponse`, which results in the values being set in the user's session.

```
response.setLoginId(username);
response.setUserId(username);
```

The login id is set to the value that the user used to authenticate. The user id is set to the value that the user is associated with in the system. Often, login id and user id have the same value, but it is also possible that there is more than one login ID mapping to the same user ID.

6.7. User Authorization

Some resources are only accessible if the user has the expected roles. The `SecToken` therefore also contains roles. In the AuthState, you can add them like this:

```
response.addActualRole(role);
```

6.8. Logging

To write into the standard nevisAuth log file, you can use the Log4j 1.2.16 logging framework. A logger can be instantiated as follows:

```
private final static Logger LOG = Logger.getLogger("MyCustomLogger");
```

Logging can then be done on separate levels, such as:

```
LOG.debug("For verbose logging statements");
LOG.info("For important information");
LOG.warn("For warnings that might influence the systems behaviour negatively");
LOG.error("For serious and unexpected problems that require administrator intervention");
```

6.9. Audit Log

The audit log is a security-relevant chronological record of successful and unsuccessful login attempts. In the nevisAuth context this means that the audit log is written if the AuthEngine ends up in either the AUTH_DONE or the AUTH_ERROR state. The information that is written into the audit log is prepared by the AuthStates, which have been traversed during the attempt to login.

To prepare the audit trail, an AuthState can create a new `AuthMarker` in the code by calling `markAuthenticate()` on the response object:

```
res.markAuthentication(new AuthMarker(this, "X509", AuthenticationType.TOKEN, userId));
```

The parameters that are passed to the AuthMarker constructor are:

- The AuthState that is the initiator of the Audit event.
- A descriptive name of the technology that was used in this AuthState.
- The authentication type, which can be one of the following: USERNAME_PASSWORD, TOKEN, CHALLENGE_RESPONSE, EXTERN, FEDERATION, ONE_TIME_PASSWORD, SELECTION, MUTATION, or NONE.
- The user, who is involved in the request.

6.10. Error Reporting

Error handling is essential in nevisAuth for qualifying problems quickly by reading the nevisAuth log. If something goes severely wrong, nevisAuth should log errors in its logfile using the trace level `error`, such that an operator can identify the source of the problem easily and quickly. AuthState errors can be categorized into two types: Start-up errors and processing errors.

Start-up errors happen during execution of the `init()` method, for example if a mandatory config property is not set. They should be propagated by throwing a `BadConfigurationException`. The AuthEngine will catch `BadConfigurationException`s and log the name of the failed AuthState and its failure cause into the log file, e.g.:

```
ENGINE ERROR:      State 'MyAuthState' failed initialization:
Missing required property 'usernameSource' in AuthState 'MyCustomState' of class
'ch.my.custom.MyAuthState'
```

When the initialization of an AuthState fails, the `LOG.error()` method should not be called directly in the AuthState. Instead it is best practice to throw the `BadConfigurationException` with a descriptive message, since the AuthEngine will log it with the trace level `error` once it is caught.

During processing, the procedure is similar: Error cases are covered by throwing an `AuthStateException` with a descriptive message. Error logging should not be done directly in the AuthState. For example:

```
try {
  // somewhere in here, ScriptException is thrown.
} catch (ScriptException e) {
  throw new AuthStateException("Failed to evaluate script: " + e, e);
}
```

6.11. GUI Generation

For GUI generation, the AuthEngine calls the AuthState method `generate()`. This method implements the construction of the GUI descriptor, which instructs nevisLogRend what data must be displayed in the GUI.

In rare cases, e.g., if a direct response must be prepared instead of a GUI descriptor, the `generate()` method can be overridden. This means that nevisAuth can instruct clients like nevisProxy to send a response generated in nevisAuth to the user agent directly instead of passing the GUI descriptor to nevisLogRend first.

As with `process()`, the `generate()` method is also not thread-safe and should never override any instance variables, only consume them.

In most common cases, the `generate()` method does not have to be overridden, though.

6.12. Access to Configured Keys

In some cases the custom AuthState might require to access some key material to process the request as expected. For example an AuthState might require a certificate to sign a result, another AuthState might need to have access to a public key to decrypt some of the contents received in the request, etc. The configuration of nevisAuth allows to specify certificates in the `esauth4.xml` file. These keys can be accessed using the class `ch.nevis.nevisauth.commons.config.KeyUtil` of the API.

The following example describes how to navigate through all the keys defined in the configuration:

```
package ch.example;

import java.io.PrintStream;
import java.security.PrivateKey;
import java.security.cert.X509Certificate;

import ch.nevis.nevisauth.commons.config.ConfiguredKeyStore;
import ch.nevis.nevisauth.commons.config.KeyObject;
import ch.nevis.nevisauth.commons.config.KeyService;
import ch.nevis.nevisauth.commons.config.KeyUtil;

/**
 * A class that prints all the key material defined in the nevisAuth configuration file.
 */
public final class KeyMaterialPrinter {
    private KeyMaterialPrinter() {
    }

    public static void printKeys(PrintStream out) {
        KeyService keyService = KeyUtil.defaultKeyService();
        keyService.configuredKeyStores().forEach(keyStore -> printKeyStore(out, keyStore
));
    }

    private static void printKeyStore(PrintStream out, ConfiguredKeyStore keyStore) {
        out.print("KeyStore name: ");
        out.println(keyStore.name());
        keyStore.keyObjects().forEach(keyObject -> printKeyObject(out, keyObject));
        out.println();
    }

    private static void printKeyObject(PrintStream out, KeyObject keyObject) {
        out.print("    KeyObject name: ");
        out.println(keyObject.name());
        keyObject.privateKey().ifPresent(privateKey -> printPrivateKey(out, privateKey));
        keyObject.certificates().forEach(certificate -> printCertificate(out,
certificate));
    }

    private static void printPrivateKey(PrintStream out, PrivateKey privateKey) {
        // Print the contents of the private key...
    }

    private static void printCertificate(PrintStream out, X509Certificate certificate) {
        // Print the contents of the certificate...
    }
}
```

6.13. A Complete Example

A complete example of an **`AuthState`** implementation could look as follows:

```
package ch.example;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.UncheckedIOException;
import java.util.Properties;
```

```java
import ch.nevis.esauth.BadConfigurationException;
import ch.nevis.esauth.auth.engine.AuthConst;
import ch.nevis.esauth.auth.engine.AuthRequest;
import ch.nevis.esauth.auth.engine.AuthResponse;
import ch.nevis.esauth.auth.engine.AuthState;
import ch.nevis.nevisauth.commons.config.Configuration;
import ch.nevis.nevisauth.commons.context.Context;
import ch.nevis.nevisauth.commons.context.Scope;

import org.apache.log4j.Logger;


/**
 * This class is an example for how the nevisauth-authstate-api can be used.
 * It implements a use case where a user is authenticated against a password
 * file.
 * The password file should be a delimited text file. Per default the
 * UseridPasswordFileAuthState expects a ':' separated file where the password
 * are SHA-256 hashes in a hexadecimal format. The location of the password
 * file can be configured in the AuthState configuration file so that it
 * gets passed as a property to the {@link #init(Properties)} method.
 * The logic for parsing the file and checking whether a user has provided the
 * correct credentials is extracted into a separate class {@link CredentialProvider}
 *
 **/
public class UseridPasswordFileAuthState extends AuthState {

    /*
     * Logger used to log messages to the esauth4.log file.
     * The Logger's logging level can be parameterized in
     * the log4j.xml file.
     */
    private final static Logger LOG = Logger.getLogger("ApiAuthStates");

    private CredentialProvider credentialProvider;

    private boolean logPassword = false;

    @Override
    public void init(Properties cfg) throws BadConfigurationException {
        Configuration configuration = new Configuration(cfg);

        String passwordFileLocation =  configuration.getPropertyAsString(
"passwordFileLocation");
        logPassword = configuration.getPropertyAsBoolean("logPassword", false);

        try {
            this.credentialProvider = CredentialProvider
.createFromDelimetedUseridPasswordFile(passwordFileLocation, ":");
        } catch (FileNotFoundException e) {
            String errorString = "PasswordFile location: " + passwordFileLocation + " is
invalid.";
            LOG.debug(errorString);
            throw new BadConfigurationException(errorString, e);
        } catch (IOException e) {
            String errorString = "Error while processing: " + passwordFileLocation;
            LOG.debug(errorString);
            throw new UncheckedIOException(errorString, e);
        }

        LOG.info("Initialized UseridPasswordFileAuthState");
    }
    @Override
    public void authenticate(AuthRequest req, AuthResponse res) {
        Context context = new Context(req, res);

        String uid = context.getValue(Scope.INARGS, "username");
        String pwd = context.getValue(Scope.INARGS, "password");

        res.setLoginId(uid);

        if (uid != null && credentialProvider.isUserAuthentic(uid, pwd)) {
            if(logPassword) {
                LOG.info("successfully authenticate user: " + uid + ", password: " + pwd
);
            } else {
                LOG.info("successfully authenticate user: " + uid);
            }
            res.setUserId(uid);
            res.setResult("ok");
        } else{
            LOG.info("user: " + uid + " could not be authenticated");
            res.setError(AuthConst.AUTH_FAILED, "wrong credentials");
            res.setResult("failed");
        }
    }
}
```

7. Plug-In Deployment

In the Quickstart chapter, we already learned how to integrate a simple custom AuthState into our nevisAuth instance. This chapter provides more insight into how to deal with specific situations.

7.1. Plug-in Packaging

A nevisAuth plug-in should bring all its required dependencies, except the `nevisauth-authstate-api`. By default the `pom.xml` generated by the Maven archetype uses the Apache Maven Assembly plug-in (http://maven.apache.org/plugins/maven-assembly-plugin/) to create a `ZIP archive` file containing the plug-in `jar` file and all its transitive `compile` and `runtime` dependencies in the subfolder `lib`. Additionally, the files `README`, `LICENSE`, and `NOTICE` are included if they exist in the project base directory.

7.2. Plug-in Classpaths

nevisAuth plug-ins are loaded when the nevisAuth instance starts. The AuthEngine checks specific paths on the filesystem to find the configured AuthStates and their dependencies.

The paths that are investigated by the AuthEngine are configured in the nevisAuth configuration:

```
<AuthEngine name="AuthEngine"
    classPath="/opt/nevisauth/plugin:/var/opt/nevisauth/default/plugin"
    classLoadStrategy="PARENT_FIRST"
    useLiteralDictionary="true"
    addAutheLevelToSecRoles="true"
    compatLevel="none"
    inputLanguageCookie="LANG"
>
```

The plug-ins that are delivered with the nevisAuth package are located in the /opt/nevisauth/plugin directory. We recommend putting instance-specific plug-ins into the instance directory: /var/opt/nevisauth/<instance-name>/plugin.

The classloading strategy defines how the classes are discovered by the classloader. With **`PARENT_FIRST`**, the AuthEngine classloader first checks whether the class has already been loaded by a parent classloader (which is the container's classloader). With **`PARENT_LAST`**, the AuthEngine classloader will try to discover and load the class, but delegate the class loading to the parent if it cannot be found.

Different strategies have different effects. With **`PARENT_FIRST`**, you reduce the amount of the MetaSpace memory of the Java virtual machine. However, if you are using libraries that also exist in the web application server classpath it is recommended using **`PARENT_LAST`** to avoid classloading conflicts.

7.3. Classpath Conflict Resolution

Similar to the problem of loading classes from the AuthEngine's classpath first vs. loading classes from the web application server's classpath first, different plug-ins may also produce library conflicts. Therefore, it is sometimes necessary to configure a classpath on the AuthState to avoid library dependency confusion. This will lead to a separate classloader being used for that AuthState (an AuthState classloader).

The problem can be illustrated as follows: Imagine an AuthState A, which depends on library_v1 and an AuthState B, which depends on library_v2. If the libraries contain the same classes but implement different functionality or method signatures, nevisAuth may fail during runtime because AuthState A and B expect a different behavior of the library.

This can be solved by configuring a separate classloader per AuthState, as the following example shows:

```
<AuthState name="UseridPasswordState" class="ch.example.UseridPasswordFileAuthState"
        classPath="/var/opt/nevisauth/default/plugin/auth-state-example-1.0-SNAPSHOT"
        classLoadStrategy="PARENT_LAST">
    <ResultCond name="ok" next="AuthDone"/>
    <ResultCond name="failed" next="AuthError" />
    <property name="passwordFileLocation" value=
"/var/opt/nevisauth/default/conf/passwords.txt"/>
</AuthState>
```

In this case, the **`ZIP archive auth-state-example-1.0-SNAPSHOT.zip`** containing the AuthState **`ch.example.UseridPasswordFileAuthState`** must be extracted into the directory **`/var/opt/default/plugin`**. Due to the **`PARENT_LAST`** class loading strategy, the directory **`/var/opt/nevisauth/default/plugin/auth-state-example-1.0-SNAPSHOT`** will be consulted first before checking the AuthEngine's classpath for loading classes.