PicketLink Reference Documentation

PicketLink [http://www.jboss.org/picketlink]

PicketLink Reference Documentation

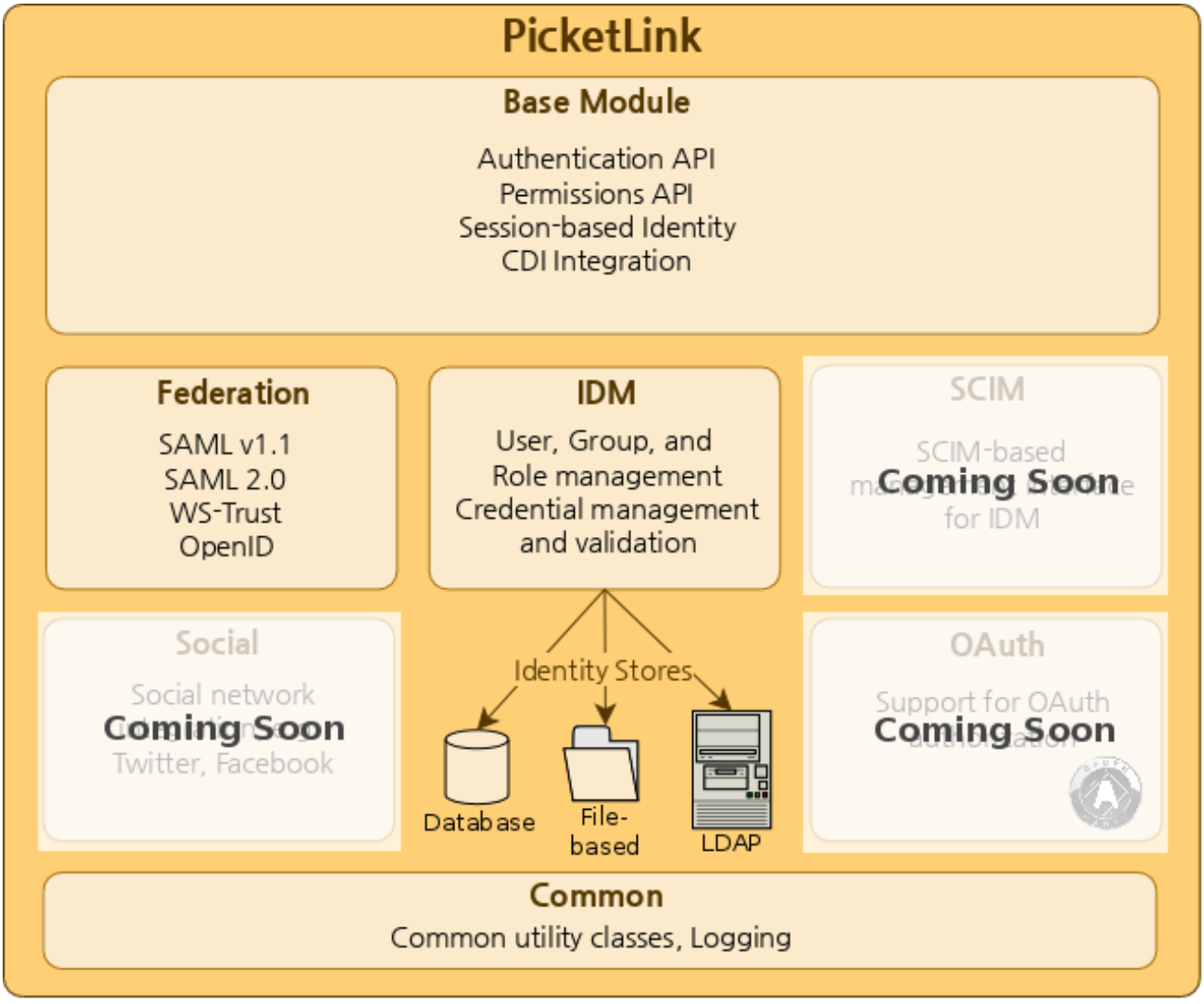
by

Version 2.5.3.Beta1

Chapter 1. Overview

1.1. What is PicketLink?

PicketLink is an Application Security Framework for Java EE applications. It provides features for authenticating users, authorizing access to the business methods of your application, managing your application's users, groups, roles and permissions, plus much more. The following diagram presents a high level overview of the PicketLink modules and the main features provided by those modules:



1.2. Where do I get started?

Depending on exactly which PicketLink features you'd like to use, getting started can be as simple as adding the PicketLink jar libraries to your project (see Section 1.5, “Maven Dependencies” below for more info) and writing a few lines of code. To get started using PicketLink Identity Management to manage the users and other identity objects in your application, you can head

straight to Section 3.2, “Getting Started - The 5 Minute Guide”. If you don't wish to use PicketLink IDM but would like to use PicketLink for authentication in your Java EE application but control the authentication process yourself then head to Section 2.3.1, “A Basic Authenticator” for simplistic example which you may adapt for your own application. If you wish to use SAML SSO then you can head to Chapter 11, *Federation*.

Here's some additional resources also to help you get started:

1.2.1. QuickStarts

Please head to Chapter 12, *PicketLink Quickstarts* for more information about our quickstarts, covering some useful usecases and most of PicketLink features.

1.2.2. API Documentation

The latest version of the PicketLink API documentation can be found at http://docs.jboss.org/picketlink/2/latest/api/. This handy reference describes the user-facing classes and methods provided by the PicketLink libraries.

1.3. Modules

1.3.1. Base module

The base module provides the integration framework required to use PicketLink within a Java EE application. It defines a flexible authentication API that allows pluggable authentication mechanisms to be easy configured, with a sensible default authentication policy that delegates to the identity management subsystem. It provides session-scoped authentication tracking for web applications and other session-capable clients, plus a customisable permissions SPI that supports a flexible range of authorization mechanisms for object-level security. It is the "glue" that integrates all of the PicketLink modules together to provide a cohesive API, and also provides *CDI* producers to allow you to inject the PicketLink API objects directly into your application's beans.

The base module libraries are as follows:

- `picketlink-api` - API for PicketLink's base module.
- `picketlink-impl` - Internal implementation classes for the base API.

1.3.2. Identity Management

The Identity Management module defines the base identity model; a collection of interfaces and classes that represent the identity constructs (such as users, groups and roles) used throughout PicketLink (see the Identity Management chapter for more details). As such, it is a required module and must always be included in any application deployments that use PicketLink for security. It also provides a uniform API for managing the identity objects within your application. The Identity Management module has been designed with minimal dependencies and may be used in a Java SE environment, however the recommended environment is Java EE in conjunction with the base module.

Libraries are as follows:

- `picketlink-idm-api` - PicketLink's Identity Management (IDM) API. This library defines the Identity Model central to all of PicketLink, and all of the identity management-related interfaces.
- `picketlink-idm-impl` - Internal implementation classes for the IDM API.

1.3.3. Federation

The Federation module is an optional module that implements a number of Federated Identity standards, such as SAML (both version 1.1 and 2.0), WS-Trust and OpenID.

1.4. License

PicketLink is licensed under the Apache License Version 2, the terms and conditions of which can be found at apache.org [http://www.apache.org/licenses/LICENSE-2.0.html].

1.5. Maven Dependencies

The PicketLink libraries are available from the Maven Central Repository. The dependencies can be easily configured in your Maven-based project by using the PicketLink Bill of Materials(BOM). A BOM is a very handy tool to properly manage your dependencies, their versions and keep your project in sync with the libraries supported and distributed by a specific PicketLink version.

For most applications using Java EE 6.0, the *picketlink-javaee-6.0* BOM can be used to define the PicketLink and Java EE 6.0 specification APIs dependencies to your project.

```
<properties>
  <!-- PicketLink dependency versions -->
  <version.picketlink.javaee.bom>2.5.3.Beta1</version.picketlink.javaee.bom>
</properties>

<dependencyManagement>
  <dependencies>
    <!-- Dependency Management for PicketLink and Java EE 6.0. -->
    <dependency>
      <groupId>org.picketlink</groupId>
      <artifactId>picketlink-javaee-6.0</artifactId>
      <version>${version.picketlink.javaee.bom}</version>
      <scope>import</scope>
      <type>pom</type>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Once the BOM is included, required PicketLink dependencies may be added to the section of your pom.xml. When using a BOM you don't need to specify their versions because this is automatically managed, the version in use depends on the BOM's version. For example, the configuration above

is defining a version 2.5.3.Beta1 of the *picketlink-javaee-6.0* BOM, which means you'll be using version 2.5.3.Beta1 of the PicketLink libraries.

For a typical application, it is suggested that you include the following PicketLink dependencies:

```
<dependencies>
<!-- Import the PicketLink API, we deploy this as library with the application,
        and can compile against it -->
  <dependency>
    <groupId>org.picketlink</groupId>
    <artifactId>picketlink-api</artifactId>
  </dependency>

  <!-- Import the PicketLink implementation, we deploy this as library with the application,
        but don't compile against it -->
  <dependency>
    <groupId>org.picketlink</groupId>
    <artifactId>picketlink-impl</artifactId>
    <scope>runtime</scope>
  </dependency>
</dependencies>
```

PicketLink also provides a specific BOM with the Apache Deltaspike(core and security modules) dependencies if you want to use it in conjunction with PicketLink.

```
<properties>
  <!-- PicketLink dependency versions -->
  <version.picketlink.javaee.bom>2.5.3.Beta1</version.picketlink.javaee.bom>
</properties>

<dependencyManagement>
  <dependencies>
    <!-- Dependency Management for PicketLink and Apache Deltaspike. -->
    <dependency>
      <groupId>org.picketlink</groupId>
      <artifactId>picketlink-javaee-6.0-with-deltaspike</artifactId>
      <version>${version.picketlink.javaee.bom}</version>
      <scope>import</scope>
      <type>pom</type>
    </dependency>
  </dependencies>
</dependencyManagement>
```

The identity management library is a required dependency of the base module and so will be automatically included. If you *don't* wish to use the base module and want to use the PicketLink IDM library on its own, then only add the following dependencies:

```
<dependencies>
  <dependency>
      <groupId>org.picketlink</groupId>
      <artifactId>picketlink-idm-api</artifactId>
```

```
      <scope>compile</scope>
  </dependency>

  <dependency>
      <groupId>org.picketlink</groupId>
      <artifactId>picketlink-idm-impl</artifactId>
      <scope>runtime</scope>
  </dependency>
```

If you wish to use PicketLink's Identity Management features and want to include the default database schema (see the Identity Management chapter for more details) then configure the following dependency also:

```
  <dependency>
    <groupId>org.picketlink</groupId>
    <artifactId>picketlink-idm-simple-schema</artifactId>
  </dependency>
```

Another way to configure the PicketLink dependencies (without using the PicketLink BOM) is to manually define them in your project's *pom.xml* file like so:

```
<properties>
  <picketlink.version>2.5.3.Beta1</picketlink.version>
</properties>

<dependencies>
  <dependency>
      <groupId>org.picketlink</groupId>
      <artifactId>picketlink-idm-api</artifactId>
      <scope>compile</scope>
      <version>${picketlink.version}</version>
  </dependency>

  <dependency>
      <groupId>org.picketlink</groupId>
      <artifactId>picketlink-idm-impl</artifactId>
      <scope>runtime</scope>
      <version>${picketlink.version}</version>
  </dependency>
</dependencies>
```

This last option may be more suitable for projects that don't use Java EE (for example in a Java SE environment).

> **Note**
>
> We strongly recommend using a BOM to configure your project with the PicketLink dependencies. This can avoid some very common and tricky issues like keep the

versions in the project using the artifacts in sync with the versions distributed in a release.

1.6. PicketLink Installer

The PicketLink Installer is a simple Apache Ant script that applies all the necessary changes to your JBoss Enterprise Application Platform 6.1 installation, including:

- Updates the PicketLink module with the latest libraries.
- Installs the PicketLink Subsystem.
- Configures some of the PicketLink Quickstarts. Specially the PicketLink Federation examples. What means you can start using them to get a picture of the SAML Single Sign-On and other features provided by PicketLink Federation. !

Important

The installer is not a required step in order to get you started with PicketLink. But if you want the PicketLink Subsystem installed and the PicketLink module updated (in order to avoid ship the libraries inside your deployment) in your JBoss Enterprise Application Platform installation, it can be very useful.

The installer can be obtained from http://downloads.jboss.org/picketlink/2/2.5.3.Beta1/picketlink-1.1.4.Final-installer.zip. Once you've downloaded, extract the ZIP file, enter the directory that was created and execute the following command:

```
user@host picketlink-installer-1.1.4.Final]$ ant
```

Now you should be prompted for the full path of your JBoss Application Server installation.

```
prepare:
     [echo]
                                          [echo]
  ###############################################################################
     [echo]       Welcome to the PicketLink Installer
     [echo]
                                          [echo]
  This installer will update your JBoss Enterprise Application Platform 6 installation with the
     [echo]       following libraries and their dependencies:
     [echo]
     [echo]       - PicketLink Core 2.5.3.Beta1
     [echo]       - PicketLink Identity Management 2.5.3.Beta1
     [echo]       - PicketLink Federation 2.5.3.Beta1
     [echo]       - PicketLink Federation Quickstarts 2.1.8.Final
```

```
     [echo]       - PicketLink Subsystem 1.1.1.Final
     [echo]
     [echo]       New modules will be added to your installation.
                                          [echo]
  ###############################################################################
     [echo]
    [input] Please enter the path to your JBoss Enterprise Application Platform 6 installation:
```

And it is done !

> **Note**
>
> There is also a specific installer for JBoss Enterprise Application Platform 6.2. You can download it from http://downloads.jboss.org/picketlink/2/2.5.3.Beta1/picketlink-1.2.2.Final-installer.zip

1.7. Help us improve the docs!

We're always looking for ways to improve this documentation. If you think that we can enhance the way that any of PicketLink's features or concepts are explained, or even if you just spot a typo or grammatical error then please let us know on the PicketLink forums (you can find a link to the forums at www.picketlink.org [http://www.picketlink.org]). We appreciate any and all feedback that is provided.

Chapter 2. Authentication

2.1. Overview

Authentication is the act of verifying the identity of a user. PicketLink offers an extensible authentication API that allows for significant customization of the authentication process, while also providing sensible defaults for developers that wish to get up and running quickly. It also supports both synchronous and asynchronous user authentication, allowing for both a traditional style of authentication (such as logging in with a username and password), or alternatively allowing authentication via a federated identity service, such as OpenID, SAML or OAuth. This chapter will endeavour to describe the authentication API and the authentication process in some detail, and is a good place to gain a general overall understanding of authentication in PicketLink. However, please note that since authentication is a cross-cutting concern, various aspects (for example Identity Management-based authentication and Federated authentication) are documented in other chapters of this book.

2.2. Authentication API - the `Identity` bean

The `Identity` bean (which can be found in the `org.picketlink` package) is central to PicketLink's security API. This bean represents the authenticated user for the current session, and provides many useful methods for controlling the authentication process and querying the user's assigned privileges. In terms of authentication, the `Identity` bean provides the following methods:

```
AuthenticationResult login();

void logout();

boolean isLoggedIn();

Account getAccount();
```

The `login()` method is the *primary* point of entry for the authentication process. Invoking this method will cause PicketLink to attempt to authenticate the user based on the credentials that they have provided. The `AuthenticationResult` type returned by the `login()` method is a simple enum that defines the following two values:

```
public enum AuthenticationResult {
    SUCCESS, FAILED
}
```

If the authentication process is successful, the `login()` method returns a result of `SUCCESS`, otherwise it returns a result of `FAILED`. The default implementation of the `Identity` bean is

a `@SessionScoped` CDI bean, which means that once a user is authenticated they will stay authenticated for the duration of the session.

> **Note**
>
> One significant point to note is the presence of the `@Named` annotation on the `Identity` bean, which means that its methods may be invoked directly from the view layer (if the view layer, such as JSF, supports it) via an EL expression.

One possible way to control the authentication process is by using an action bean, for example the following code might be used in a JSF application:

```
public @RequestScoped @Named class LoginAction {

    @Inject Identity identity;

    public void login() {
        AuthenticationResult result = identity.login();
        if (AuthenticationResult.FAILED.equals(result)) {
            FacesContext.getCurrentInstance().addMessage(null,
                new FacesMessage(
                   "Authentication was unsuccessful.  Please check your username and password " +
                        "before trying again."));
        }
    }
}
```

In the above code, the `Identity` bean is injected into the action bean via the CDI `@Inject` annotation. The `login()` method is essentially a wrapper method that delegates to `Identity.login()` and stores the authentication result in a variable. If authentication was unsuccessful, a `FacesMessage` is created to let the user know that their login failed. Also, since the bean is `@Named` it can be invoked directly from a JSF control like so:

```
<h:commandButton value="LOGIN" action="#{loginAction.login}"/>
```

The `Identity.isLoggedIn()` method may be used to determine whether there is a user logged in for the current session. It is typically used as an authorization check to control either an aspect of the user interface (for example, not displaying a menu item if the user isn't logged in), or to restrict certain business logic. While logged in, the `getAccount()` method can be used to retrieve the currently authenticated account (i.e. the user). If the current session is not authenticated, then `getAccount()` will return `null`. The following example shows both the `isLoggedIn()` and `getAgent()` methods being used inside a JSF page:

```
<ui:fragment rendered="#{identity.loggedIn}">Welcome, #{identity.account.loginName}
```

> **Note**
>
> If you're wondering what an `Account` is, it is simply a representation of the external entity that is interacting and authenticating with your application. The `Account` interface is actually the superclass of the `User` and `Agent` - see the Identity Management chapter for more details.

The `logout()` method allows the user to log out, thereby clearing the authentication state for their session. Also, if the user's session expires (for example due to inactivity) their authentication state will also be lost requiring the user to authenticate again.

The following JSF code example demonstrates how to render a log out button when the current user is logged in:

```
<ui:fragment rendered="#{identity.loggedIn}">
    <h:form>
        <h:commandButton value="Log out" action="#{identity.logout}"/>
    </h:form>
</ui:fragment>
```

While it is the `Identity` bean that controls the overall authentication process, the actual authentication "business logic" is defined by the `Authenticator` interface:

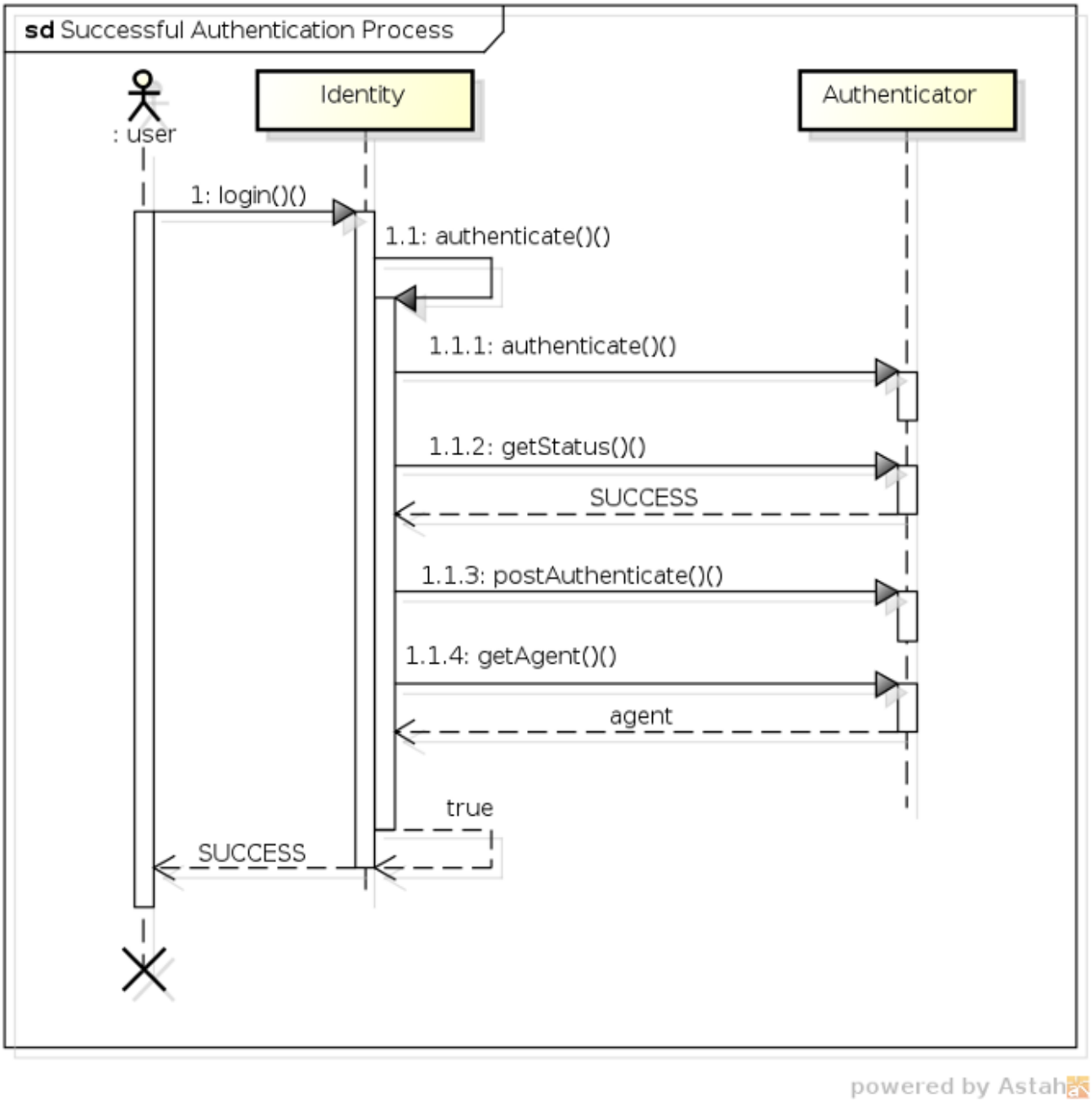
```
public interface Authenticator {
    public enum AuthenticationStatus {
        SUCCESS,
        FAILURE,
        DEFERRED
    }

    void authenticate();

    void postAuthenticate();

    AuthenticationStatus getStatus();

    Account getAccount();
}
```

During the authentication process, the `Identity` bean will invoke the methods of the *active* `Authenticator` (more on this in a moment) to perform user authentication. The `authenticate()` method is the most important of these, as it defines the actual authentication logic. After `authenticate()` has been invoked by the `Identity` bean, the `getStatus()` method will reflect the authentication status (either `SUCCESS`, `FAILURE` or `DEFERRED`). If the authentication process was a success, the `getAccount()` method will return the authenticated `Account` object and

the `postAuthenticate()` method will be invoked also. If the authentication was not a success, `getAccount()` will return `null`.

2.3. The Authentication Process

Now that we've looked at all the individual pieces, let's take a look at how they all work together to process an authentication request. For starters, the following sequence diagram shows the class interaction that occurs during a successful authentication:



- *1* - The user invokes the `login()` method of the `Identity` bean.

- *1.1* - The `Identity` bean (after performing a couple of validations) invokes its own `authenticate()` method.
- *1.1.1* - Next the `Identity` bean invokes the `Authenticator` bean's `authenticate()` method (which has a return value of `void`).
- *1.1.2* - To determine whether authentication was successful, the `Identity` bean invokes the `Authenticator`'s `getStatus()` method, which returns a `SUCCESS`.
- *1.1.3* - Upon a successful authentication, the `Identity` bean then invokes the `Authenticator`'s `postAuthenticate()` method to perform any post-authentication logic.
- *1.1.4* - The `Identity` bean then invokes the `Authenticator`'s `getAccount()` method, which returns an `Account` object representing the authenticated agent, which is then stored as a private field in the `Identity` bean.

The authentication process ends when the `Identity.authenticate()` method returns a value of `true` to the `login()` method, which in turn returns an authentication result of `SUCCESS` to the invoking user.

2.3.1. A Basic Authenticator

Let's take a closer look at an extremely simple example of an `Authenticator`. The following code demonstrates an `Authenticator` implementation that simply tests the username and password credentials that the user has provided against hard coded values of `jsmith` for the username, and `abc123` for the password, and if they match then authentication is deemed to be a success:

```
@PicketLink
public class SimpleAuthenticator extends BaseAuthenticator {

    @Inject DefaultLoginCredentials credentials;

    @Override
    public void authenticate() {
        if ("jsmith".equals(credentials.getUserId()) &&
                "abc123".equals(credentials.getPassword())) {
            setStatus(AuthenticationStatus.SUCCESS);
            setAccount(new User("jsmith"));
        } else {
            setStatus(AuthenticationStatus.FAILURE);
            FacesContext.getCurrentInstance().addMessage(null, new FacesMessage(
                "Authentication Failure - The username or password you provided were invalid."));
        }
    }
}
```

The first thing we can notice about the above code is that the class is annotated with the `@PicketLink` annotation. This annotation indicates that this bean should be used for the authentication process. The next thing is that the authenticator class extends something called `BaseAuthenticator`. This abstract base class provided by PicketLink implements the

`Authenticator` interface and provides implementations of the `getStatus()` and `getAccount()` methods (while also providing matching `setStatus()` and `setAccount()` methods), and also provides an empty implementation of the `postAuthenticate()` method. By extending `BaseAuthenticator`, our `Authenticator` implementation simply needs to implement the `authenticate()` method itself.

We can see in the above code that in the case of a successful authentication, the `setStatus()` method is used to set the authentication status to `SUCCESS`, and the `setAccount()` method is used to set the user (in this case by creating a new instance of `User`). For an unsuccessful authentication, the `setStatus()` method is used to set the authentication status to `FAILURE`, and a new `FacesMessage` is created to indicate to the user that authentication has failed. While this code is obviously meant for a JSF application, it's possible to execute whichever suitable business logic is required for the view layer technology being used.

One thing that hasn't been touched on yet is the following line of code:

```
@Inject DefaultLoginCredentials credentials;
```

This line of code injects the credentials that have been provided by the user using CDI's `@Inject` annotation, so that our `Authenticator` implementation can query the credential values to determine whether they're valid or not. We'll take a look at credentials in more detail in the next section.

> **Note**
>
> You may be wondering what happens if you don't provide an `Authenticator` bean in your application. If this is the case, PicketLink will automatically authenticate via the identity management API, using a sensible default configuration. See the Identity Management chapter for more information.

2.3.2. Multiple Authenticator Support

If your application needs to support multiple authentication methods, you can provide the authenticator selection logic within a producer method annotated with `@PicketLink`, like so:

```
@RequestScoped
@Named
public class AuthenticatorSelector {

    @Inject Instance<CustomAuthenticator> customAuthenticator;
    @Inject Instance<IdmAuthenticator> idmAuthenticator;

    private String authenticator;

    public String getAuthenticator() {
```

```
        return authenticator;
    }

    public void setAuthenticator(String authenticator) {
        this.authenticator = authenticator;
    }

    @Produces
    @PicketLink
    public Authenticator selectAuthenticator() {
        if ("custom".equals(authenticator)) {
            return customAuthenticator.get();
        } else {
            return idmAuthenticator.get();
        }
    }
}
```

This `@Named` bean exposes an `authenticator` property that can be set by a user interface control in the view layer. If its value is set to "custom" then `CustomAuthenticator` will be used, otherwise `IdmAuthenticator` (the `Authenticator` used to authenticate using the identity management API) will be used instead. This is an extremely simple example but should give you an idea of how to implement a producer method for authenticator selection.

2.3.3. Credentials

Credentials are something that provides evidence of a user's identity; for example a username and password, an X509 certificate or some kind of biometric data such as a fingerprint. PicketLink has extensive support for a variety of credential types, and also makes it relatively simple to add custom support for credential types that PicketLink doesn't support out of the box itself.

In the previous section, we saw a code example in which a `DefaultLoginCredentials` (an implementation of the `Credentials` interface that supports a user ID and a credential value) was injected into the `SimpleAuthenticator` bean. The most important thing to know about the `Credentials` interface in relation to writing your own custom `Authenticator` implementation is that *you're not forced to use it*. *However*, while the `Credentials` interface is mainly designed for use with the Identity Management API (which is documented in a separate chapter) and its methods would rarely be used in a custom `Authenticator`, PicketLink provides some implementations which are suitably convenient to use as such, `DefaultLoginCredentials` being one of them.

So, in a custom `Authenticator` such as this:

```
public class SimpleAuthenticator extends BaseAuthenticator {

    @Inject DefaultLoginCredentials credentials;

    // code snipped
}
```

The credential injection is totally optional. As an alternative example, it is totally valid to create a request-scoped bean called `UsernamePassword` with simple getters and setters like so:

```
public @RequestScoped class UsernamePassword {
    private String username;
    private String password;

    public String getUsername() { return username; }
    public String getPassword() { return password; }

    public void setUsername(String username) { this.username = username; }
    public void setPassword(String password) { this.password = password; }
}
```

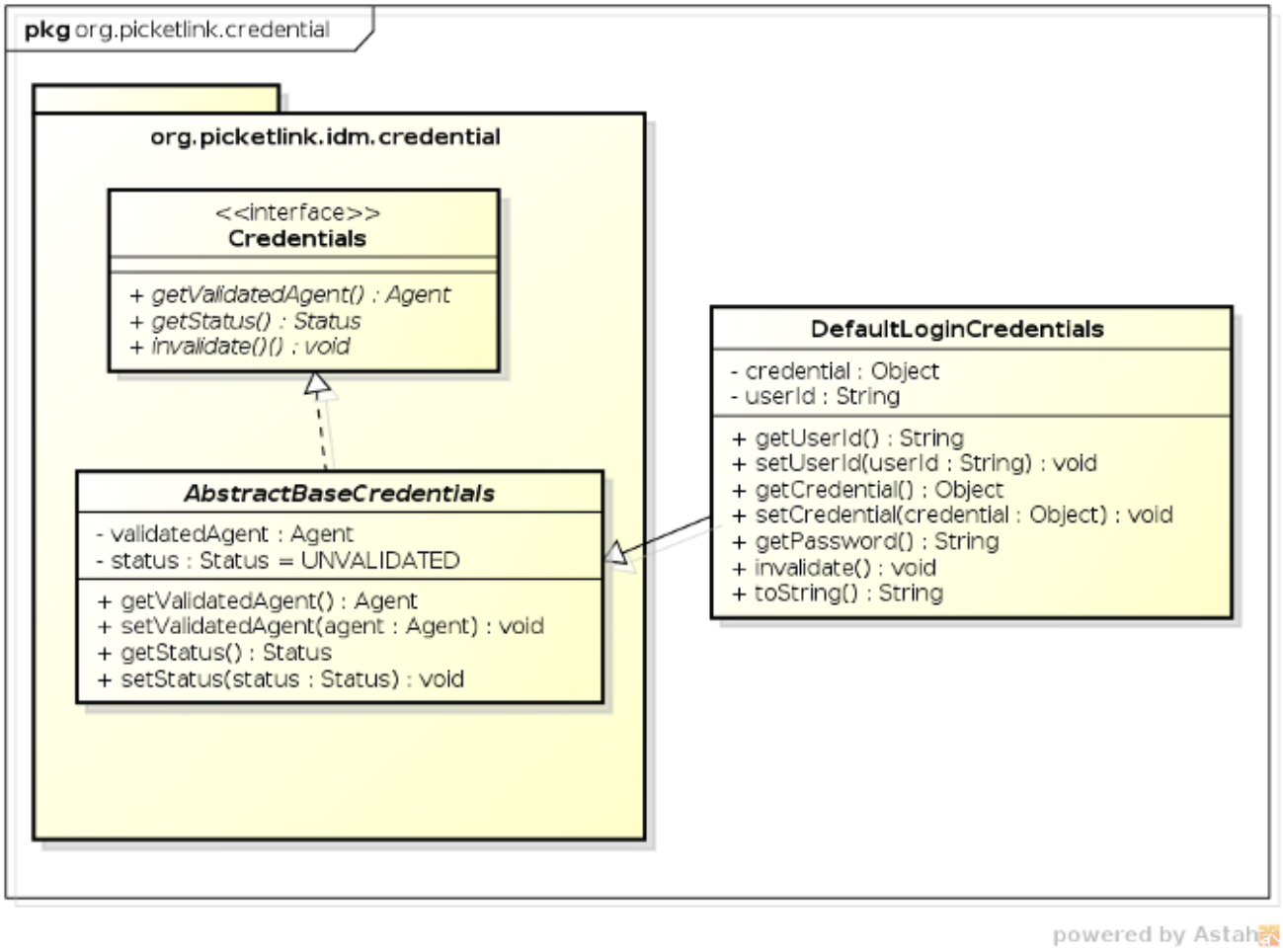
And then inject that into the `Authenticator` bean instead:

```
public class SimpleAuthenticator extends BaseAuthenticator {

    @Inject UsernamePassword usernamePassword;

    // code snipped
}
```

Of course it is not recommended that you actually do this, however this simplistic example serves adequately for demonstrating the case in point.

2.3.4. DefaultLoginCredentials

The `DefaultLoginCredentials` bean is provided by PicketLink as a convenience, and is intended to serve as a general purpose `Credentials` implementation suitable for a variety of use cases. It supports the setting of a `userId` and `credential` property, and provides convenience methods for working with text-based passwords. It is a request-scoped bean and is also annotated with `@Named` so as to make it accessible directly from the view layer.



A view technology with support for EL binding (such as JSF) can access the `DefaultLoginCredentials` bean directly via its bean name, `loginCredentials`. The following code snippet shows some JSF markup that binds the controls of a login form to `DefaultLoginCredentials`:

```
<div class="loginRow">
  <h:outputLabel for="name" value="Username" styleClass="loginLabel"/>
  <h:inputText id="name" value="#{loginCredentials.userId}"/>
</div>

<div class="loginRow">
  <h:outputLabel for="password" value="Password" styleClass="loginLabel"/>
  <h:inputSecret id="password" value="#{loginCredentials.password}" redisplay="true"/>
</div>
```

Chapter 3. Identity Management - Overview

3.1. Introduction

PicketLink Identity Management (IDM) is a fundamental module of PicketLink, with all other modules building on top of the IDM component to implement their extended features. It features provide a rich and extensible API for managing the identities (such as users, groups and roles) of your applications and services. It also supports a flexible system for identity partitioning, allowing it to be used as a complete security solution in simple web applications and/or as an Identity Provider (IDP) in more complex multi-domain scenarios. It also provides the core Identity Model API classes (see below) upon which an application's identity classes are defined to provide the security structure for that application.

The Identity Management features of PicketLink are accessed primarily via the following three interfaces:

- `PartitionManager` is used to manage identity *partitions*, which are essentially a *container* for a set of identity objects. The `PartitionManager` interface provides a set of CRUD methods for creating, reading, updating and deleting partitions, as well as methods for creating an `IdentityManager` or `RelationshipManager` (more on these next). A typical Java EE application with simple security requirements will likely not be required to access the `PartitionManager` API directly.

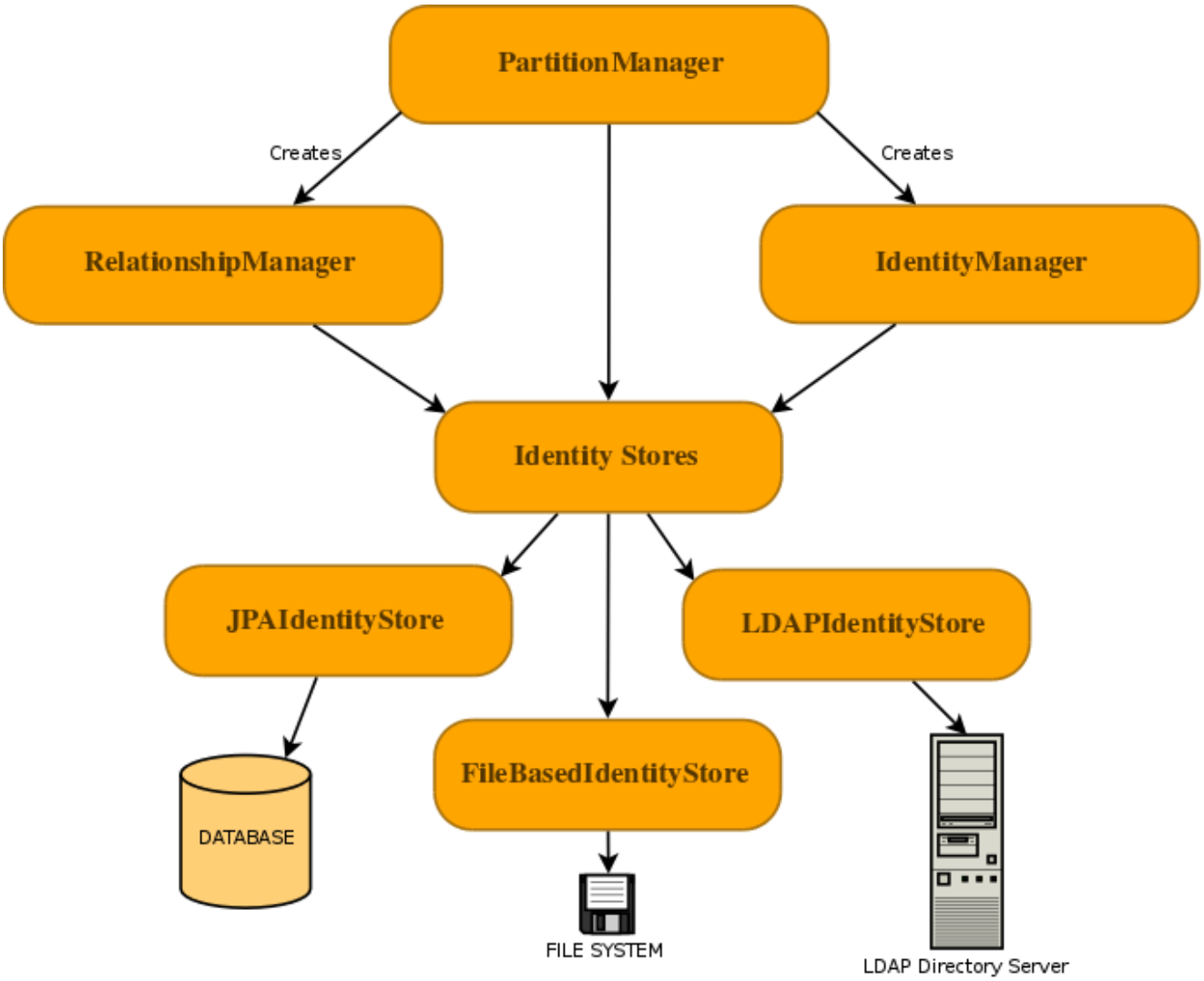
- `IdentityManager` is used to manage *identity objects* within the scope of a partition. Some examples of identity objects are users, groups and roles, although PicketLink is not limited to just these. Besides providing the standard set of CRUD methods for managing and locating identity objects, the `IdentityManager` interface also defines methods for managing credentials and for creating identity queries which may be used to locate identities by their properties and attributes.

- `RelationshipManager` is used to manage *relationships* - a relationship is a typed association between two or more identities, with each identity having a definitive meaning within the relationship. Some examples of relationships that may already be familiar are group memberships (where a user is a member of a particular group) or granted roles (where a user is assigned to a role to afford them a certain set of privileges). The `RelationshipManager` provides CRUD methods for managing relationships, and also for creating a relationship query which may be used to locate relationships between identities based on the relationship type and participating identity object/s.

Note

In case you are wondering why a separate `RelationshipManager` interface is required for managing relationships between identites, it is because PicketLink supports relationships between identities belonging to separate partitions; therefore the scope of a `RelationshipManager` instance is not constrained to a single partition in the same way as the `IdentityManager`.

Interaction with the backend store that provides the persistent identity state is performed by configuring one or more `IdentityStore`s. PicketLink provides a few built-in `IdentityStore` implementations for storing identity state in a database, file system or LDAP directory server, and it is possible to provide your own custom implementation to support storing your application's identity data in other backends, or extend the built-in implementations to override their default behaviour.



3.1.1. Injecting the Identity Management Objects

In a Java EE environment PicketLink provides a set of producer methods for injecting the primary identity management objects into your CDI beans. The following table outlines which IDM classes may be injected, and the CDI scope of each of the beans.

Table 3.1. Identity Management Objects

Class Name	Scope
`org.picketlink.idm.PartitionManager`	`@ApplicationScoped`
`org.picketlink.idm.IdentityManager`	`@RequestScoped`
`org.picketlink.idm.RelationshipManager`	`@RequestScoped`

3.1.2. Interacting with PicketLink IDM During Application Startup

Since the `IdentityManager` and `RelationshipManager` beans are request scoped beans (as per the above table) it is not possible to inject them directly into a `@Startup` bean as there is no request scope available at application startup time. Instead, if you wish to use the IDM API within a `@Startup` bean in your Java EE application you may inject the `PartitionManager` (which is application-scoped) from which you can then get references to the `IdentityManager` and `RelationshipManager`:

```
@Singleton
@Startup
public class InitializeSecurity {
    @Inject private PartitionManager partitionManager;

    @PostConstruct
    public void create() {
        // Create an IdentityManager
        IdentityManager identityManager = partitionManager.createIdentityManager();

        User user = new User("shane");
        identityManager.add(user);

        Password password = new Password("password");
        identityManager.updateCredential(user, password);

        // Create a RelationshipManager
       RelationshipManager relationshipManager = partitionManager.createRelationshipManager();

        // Create some relationships
    }
}
```

3.1.3. Configuring the Default Partition

Since PicketLink has multiple-partition support, it is important to be able to control the associated partition for the injected `IdentityManager`. By default, PicketLink will injected an `IdentityManager` for the *default Realm* (i.e. the `org.picketlink.idm.model.basic.Realm` partition with a name of *default*). If your application has basic security requirements then this might well be adequate, however if you wish to override this default behaviour then simply provide a producer method annotated with the `@PicketLink` qualifier that returns the default partition for your application:

```
@ApplicationScoped
public class DefaultPartitionProducer {
    @Inject PartitionManager partitionManager;

    @Produces
    @PicketLink
    public Partition getDefaultPartition() {
        return partitionManager.getPartition(Tier.class, "warehouse.dispatch");
    }
}
```

3.2. Getting Started - The 5 Minute Guide

If you'd like to get up and running with IDM quickly, the good news is that PicketLink will provide a default configuration that stores your identity data on the file system if no other configuration is available. This means that if you have the PicketLink libraries in your project, you can simply inject the `PartitionManager`, `IdentityManager` or `RelationshipManager` beans into your own application and start using them immediately:

```
@Inject PartitionManager partitionManager;
@Inject IdentityManager identityManager;
@Inject RelationshipManager relationshipManager;
```

Once you have injected an `IdentityManager` you can begin creating users, groups and roles for your application:

> **Note**
>
> The following code examples make use of the classes provided as part of the *basic identity model* - see Chapter 5, *Identity Management - Basic Identity Model* for more information.

```
User user = new User("jdoe");
user.setFirstName("Jane");
```

```
user.setLastName("Doe");
identityManager.add(user);

Group group = new Group("employees");
identityManager.add(group);

Role admin = new Role("admin");
identityManager.add(admin);
```

Use the `RelationshipManager` to create relationships, such as role assignments and group memberships:

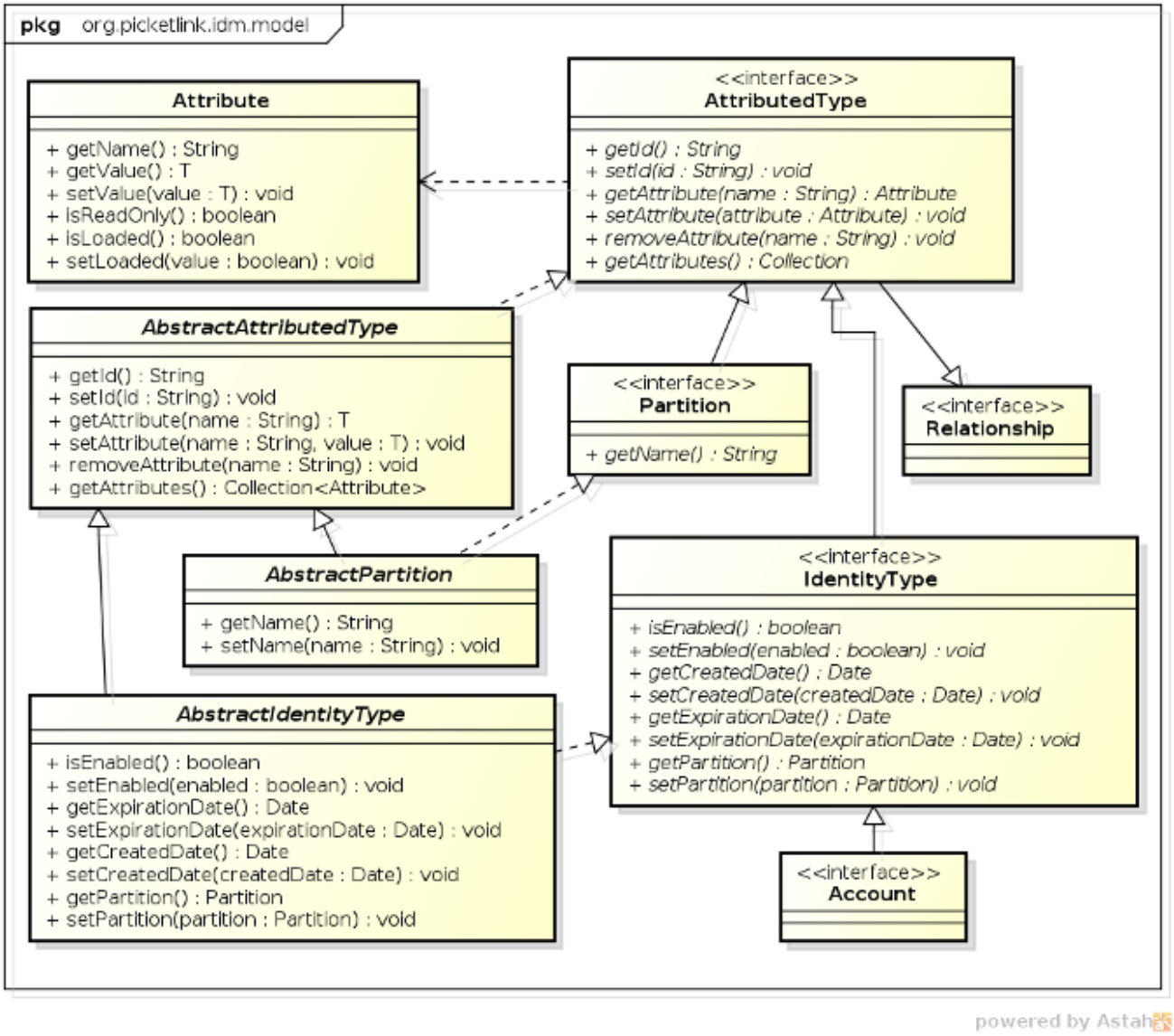
```
// Grant the admin role to the user
relationshipManager.add(new Grant(user, admin));

// Add the user to the employees group
relationshipManager.add(new GroupMembership(user, group));
```

The static methods provided by the `org.picketlink.idm.model.basic.BasicModel` class are based on the basic identity model and may be used to lookup various identity objects, or test whether certain relationships exist. These methods accept either an `IdentityManager` or `RelationshipManager` object as a parameter.

```
// Lookup the user by their username
User user = BasicModel.getUser(identityManager, "jdoe");

// Test if the user has the admin role
boolean isAdmin = BasicModel.hasRole(relationshipManager, user, role);

// Test if the user is a member of the employee group
boolean isEmployee = BasicModel.isMember(relationshipManager, user, group);
```

3.3. Identity Model

The Identity Model is a set of classes that define the security structure of an application. It may consist of identity objects such as users, groups and roles; relationships such as group and role memberships; and partitions such as realms or tiers. The classes found in the `org.picketlink.idm.model` package define the base types upon which the identity model is built upon:



- `AttributedType` is the base interface for the identity model. It declares a number of methods for managing a set of attribute values, plus `getId()` and `setId()` methods for setting a unique identifier value.
- `Attribute` is used to represent an attribute value. An attribute has a name and a (generically typed) value, and may be marked as read-only. Attribute values that are expensive to load (such as large binary data) may be lazy-loaded; the `isLoaded()` method may be used to determine whether the Attribute has been loaded or not.
- `Partition` is the base interface for partitions. Since each partition must have a name it declares a `getName()` method.
- `Relationship` is the base interface for relationships. Besides the base methods defined by the `AttributedType` interface, relationship implementations have no further contractual requirements, however they will define methods that return the identities and attribute values in accordance with the relationship type.

Which Identity Model Should My Application Use?

- `IdentityType` is the base interface for Identity objects. It declares properties that indicate whether the identity object is enabled or not, optional created and expiry dates, plus methods to read and set the owning `Partition`.
- `Account` is the base interface for identities that are capable of authenticating. Since the authentication process may not depend on one particular type of attribute (not all authentication is performed with a username and password) there are no hard-coded property accessors defined by this interface. It is up to each application to define the `Account` implementations required according to the application's requirements.
- `AbstractAttributedType` is an abstract base class for creating `AttributedType` implementations.
- `AbstractPartition` is an abstract base class that implements the base methods of the `Partition` interface, to simplify the development of partition implementations.
- `AbstractIdentityType` is an abstract base class that implements the base methods of the `IdentityType` interface, to simplify the development of identity objects.

3.3.1. Which Identity Model Should My Application Use?

The base identity types listed above do not define an identity model implementation themselves, so they cannot be used directly to service the security requirements of an application. Instead, an application must either define its own identity model (by providing implementations of whichever identity objects are required by the application, such as user, group or role classes) or by using a pre-prepared model. PicketLink provides a *basic* identity model (more details can be found in Chapter 5, *Identity Management - Basic Identity Model*) which provides a basic set of identity objects, however in case the basic identity model is insufficient, it is quite simple to define a custom model as we'll see in the next section.

3.4. Creating a Custom Identity Model

A custom identity model typically consists of two types of objects - the *identity* objects which define the security constructs of an application, and the *relationships* which define how the identity objects interract with each other to establish a meaningful security policy. PicketLink treats both types of object in an abstract manner, so it is up to the developer to create meaning for these objects and their relationships within the context of their own application. This section will describe how to create new identity objects and customize their properties and attributes, while the following section will complete the picture by describing how custom relationships are created.

Let's start by looking at the source for some of the identity objects in the basic model, starting with the `Agent` and `User` objects:

```
public class Agent extends AbstractIdentityType implements Account {
    private String loginName;

    public Agent() { }
```

```
    public Agent(String loginName) {
        this.loginName = loginName;
    }

    @AttributeProperty
    @Unique
    public String getLoginName() {
        return loginName;
    }

    public void setLoginName(String loginName) {
        this.loginName = loginName;
    }
}
```

```
public class User extends Agent {
    private String firstName;
    private String lastName;
    private String email;

    public User() { }

    public User(String loginName) {
        super(loginName);
    }

    @AttributeProperty
    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    @AttributeProperty
    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    @AttributeProperty
    public String getEmail() {
        return this.email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

The Agent class is intended to represent a third party entity that may authenticate against an application, whether human (a user) or non-human (an external or remote process). Because Agent implements the Account marker interface, it is making a declaration that this identity object is capable of authenticating. To support the typical username/password authentication method, the Agent class defines a loginName property, however since the Account interface enforces no particular method of authentication (instead of a using username for authentication your application may require a certificate or fingerprint) this property is arbitrary.

The User class represents a human user and extends Agent to add the human-specific properties *firstName*, *lastName* and *email*. Since human users are also capable of authenticating it will also inherit the loginName property from the Agent.

3.4.1. The @AttributeProperty Annotation

In the code above we can see that the getter methods of the identity objects are annotated with @AttributeProperty. This annotation (from the org.picketlink.idm.model.annotation package) is used to indicate that the property of the identity object should be persisted by the configured identity store when creating or updating the identity object. If this annotation was missing, then the property value would be null when loading the identity object from the identity store.

In this example, the annotation is placed on the getter method however it is also valid to place it on the corresponding field value.

3.4.2. The @Unique Annotation

In the above code listing for the Agent class, we can also see that there is a @Unique annotation on the getLoginName() getter method (in addition to the @AttributeProperty annotation). This special annotation (also from the org.picketlink.idm.model.annotation package) is used to indicate to PicketLink that a unique constraint must be enforced on the property value - i.e. no two Agent objects (or their subclasses) may return the same value for getLoginName().

3.5. Creating Custom Relationships

One of the strengths of PicketLink is its ability to support custom relationship types. This extensibility allows you, the developer to create specific relationship types between two or more identities to address the domain-specific requirements of your own application.

> **Note**
>
> Please note that custom relationship types are not supported by all IdentityStore implementations - see the Identity Store section above for more information.

To create a custom relationship type, we start by creating a new class that implements the Relationship interface. To save time, we also extend the AbstractAttributedType abstract class which takes care of the identifier and attribute management methods for us:

```
public class Authorization extends AbstractAttributedType implements Relationship {

}
```

The next step is to define which identities participate in the relationship. Once we create our identity property methods, we also need to annotate them with the `org.picketlink.idm.model.annotation.RelationshipIdentity` annotation. This is done by creating a property for each identity type.

```
private User user;
private Agent application;

@RelationshipIdentity
public User getUser() {
  return user;
}

public void setUser(User user) {
  this.user = user;
}

@RelationshipIdentity
public Agent getApplication() {
  return application;
}

public void setApplication(Agent application) {
  this.application = application;
}
```

We can also define some attribute properties, using the `@RelationshipAttribute` annotation:

```
private String accessToken;

@RelationshipAttribute
public String getAccessToken() {
  return accessToken;
}

public void setAccessToken(String accessToken) {
  this.accessToken = accessToken;
}
```

3.6. Partition Management

PicketLink has been designed from the ground up to support a system of *partitioning*, allowing the identity objects it manages to be separated into logical groupings. Partitions may be used to split identities into separate *realms*, allowing an application to serve multiple organisations (for

example in a SaaS architecture) or to support a multi-*tier* application allowing each tier to define its own set of identity objects (such as groups or roles). PicketLink's architecture also allows you to define your own custom partition types, allowing more complex use cases to be supported.

The `PartitionManager` interface provides the following methods for managing partitions:

```
public interface PartitionManager extends Serializable {

    <T extends Partition> T getPartition(Class<T> partitionClass, String name);

    <T extends Partition> List<T> getPartitions(Class<T> partitionClass);

    <T extends Partition> T lookupById(final Class<T> partitionClass, String id);

    void add(Partition partition);

    void add(Partition partition, String configurationName);

    void update(Partition partition);

    void remove(Partition partition);
}
```

To create a new `Partition` object you may use either of the `add()` methods. If a `configurationName` parameter value isn't provided (see Chapter 7, *Identity Management - Configuration* for more information), then the newly created `Partition` will use the default configuration.

```
// Create a new Realm partition called "acme"
partitionManager.add(new Realm("acme"));
```

```
// Create a new Tier partition called "sales" using the named configuration "companyAD"
partitionManager.add(new Tier("sales"), "companyAD");
```

Each new `Partition` object created will be automatically assigned a unique identifier value, which can be accessed via its `getId()` method:

```
Realm realm = new Realm("acme");
partitionManager.add(realm);
String partitionId = realm.getId();
```

Partitions may be retrieved using either their name or their unique identifier value. Both methods require the exact partition class to be provided as a parameter:

```
Realm realm = partitionManager.getPartition(Realm.class, "acme");
```

```
Tier tier = partitionManager.lookupById(Tier.class, tierId);
```

It is also possible to retrieve all partitions for a given partition class. In this case you can retrieve all partitions for a given type or all of them:

```
List<Realm> realms = partitionManager.getPartitions(Realm.class);
List<Partition> allPartitions = partitionManager.getPartitions(Partition.class);
```

Since `Partition` objects all implement the `AttributedType` interface, it is also possible to set arbitrary attribute values:

```
realm.setAttribute(new Attribute<Date>("created", new Date()));
```

After making changes to an existing `Partition` object, the `update()` method may be used to persist those changes:

```
partitionManager.update(realm);
```

A `Partition` object may also be removed with the `remove()` method:

> **Warning**
>
> Removing a `Partition` object is permanent, and will also remove all identity objects that exist within that partition!

```
partitionManager.remove(realm);
```

3.6.1. Creating Custom Partitions

Creating a custom partition type is extremely simple. PicketLink provides an abstract base class called `AbstractPartition` (see above) which makes creating a custom partition class a trivial exercise - simply extend the `AbstractPartition` class and then add any additional property getter/setter methods that you might require. Let's take a look at the built-in `Realm` class to see how little code it requires to create a custom partition:

```
@IdentityPartition(supportedTypes = {IdentityType.class})
public class Realm extends AbstractPartition {
    public Realm() {
        super(null);
```

```
    }

    public Realm(String name) {
        super(name);
    }
}
```

The `@IdentityPartition` annotation must be present on the partition class - the `supportedTypes` member is used to configure which identity types may be stored in this partition. Any identity object (or subclass) specified by `supportedTypes` is valid. There is also a `unsupportedTypes` member which may be used to specify identity types which *may not* be stored in the partition. This value can be used to *trim* unsupported classes (and their subclasses) off the `supportedTypes`.

Chapter 4. Identity Management - Credential Validation and Management

4.1. Authentication

> **Note**
>
> While the IDM module of PicketLink provides authentication features, for common use cases involving standard username and password based authentication in a Java EE environment, PicketLink provides a more streamlined method of authentication. Please refer to Chapter 2, *Authentication* for more information.

PicketLink IDM provides an authentication subsystem that allows user credentials to be validated thereby confirming that an authenticating user is who they claim to be. The `IdentityManager` interface provides a single method for performing credential validation, as follows:

```
void validateCredentials(Credentials credentials);
```

The `validateCredentials()` method accepts a single `Credentials` parameter, which should contain all of the state required to determine who is attempting to authenticate, and the credential (such as a password, certificate, etc) that they are authenticating with. Let's take a look at the `Credentials` interface:

```
public interface Credentials {
    public enum Status {
        UNVALIDATED, IN_PROGRESS, INVALID, VALID, EXPIRED
    };

   Account getValidatedAccount();

   Status getStatus();

   void invalidate();
}
```

- The `Status` enum defines the following values, which reflect the various credential states:
  - `UNVALIDATED` - The credential is yet to be validated.

- `IN_PROGRESS` - The credential is in the process of being validated.
- `INVALID` - The credential has been validated unsuccessfully
- `VALID` - The credential has been validated successfully
- `EXPIRED` - The credential has expired

- `getValidatedAccount()` - If the credential was successfully validated, this method returns the `Account` object representing the validated user.
- `getStatus()` - Returns the current status of the credential, i.e. one of the above enum values.
- `invalidate()` - Invalidate the credential. Implementations of `Credential` should use this method to clean up internal credential state.

Let's take a look at a concrete example - `UsernamePasswordCredentials` is a `Credentials` implementation that supports traditional username/password-based authentication:

```
public class UsernamePasswordCredentials extends AbstractBaseCredentials {

    private String username;

    private Password password;

    public UsernamePasswordCredentials() { }

    public UsernamePasswordCredentials(String userName, Password password) {
        this.username = userName;
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

    public UsernamePasswordCredentials setUsername(String username) {
        this.username = username;
        return this;
    }

    public Password getPassword() {
        return password;
    }

    public UsernamePasswordCredentials setPassword(Password password) {
        this.password = password;
        return this;
    }

    @Override
    public void invalidate() {
        setStatus(Status.INVALID);
        password.clear();
    }
```

```
}
```

The first thing we may notice about the above code is that the `UsernamePasswordCredentials` class extends `AbstractBaseCredentials`. This abstract base class implements the basic functionality required by the `Credentials` interface. Next, we can see that two fields are defined; `username` and `password`. These fields are used to hold the username and password state, and can be set either via the constructor, or by their associated setter methods. Finally, we can also see that the `invalidate()` method sets the status to `INVALID`, and also clears the password value.

Let's take a look at an example of the above classes in action. The following code demonstrates how we would authenticate a user with a username of "john" and a password of "abcde":

```
Credentials creds = new UsernamePasswordCredentials("john",
    new Password("abcde"));
identityManager.validate(creds);
if (Status.VALID.equals(creds.getStatus())) {
  // authentication was successful
}
```

We can also test if the credentials that were provided have expired (if an expiry date was set). In this case we might redirect the user to a form where they can enter a new password.

```
Credentials creds = new UsernamePasswordCredentials("john",
    new Password("abcde"));
identityManager.validate(creds);
if (Status.EXPIRED.equals(creds.getStatus())) {
  // password has expired, redirect the user to a password change screen
}
```

4.2. Managing Credentials

Updating user credentials is even easier than validating them. The `IdentityManager` interface provides the following two methods for updating credentials:

```
void updateCredential(Account account, Object credential);
void updateCredential(Account account, Object credential, Date effectiveDate, Date expiryDate);
```

Both of these methods essentially do the same thing; they update a credential value for a specified `Account`. The second overloaded method however also accepts `effectiveDate` and `expiryDate` parameters, which allow some temporal control over when the credential will be valid. Use cases for this feature include implementing a strict password expiry policy (by providing an expiry date), or creating a new account that might not become active until a date in the future (by providing an effective date). Invoking the first overloaded method will store the credential with an effective date of the current date and time, and no expiry date.

> **Note**
>
> One important point to note is that the `credential` parameter is of type `java.lang.Object`. Since credentials can come in all shapes and sizes (and may even be defined by third party libraries), there is no common base interface for credential implementations to extend. To support this type of flexibility in an extensible way, PicketLink provides an SPI that allows custom credential handlers to be configured that override or extend the default credential handling logic. Please see the next section for more information on how this SPI may be used.

Let's take a look at a couple of examples. Here's some code demonstrating how a password can be assigned to user "jsmith":

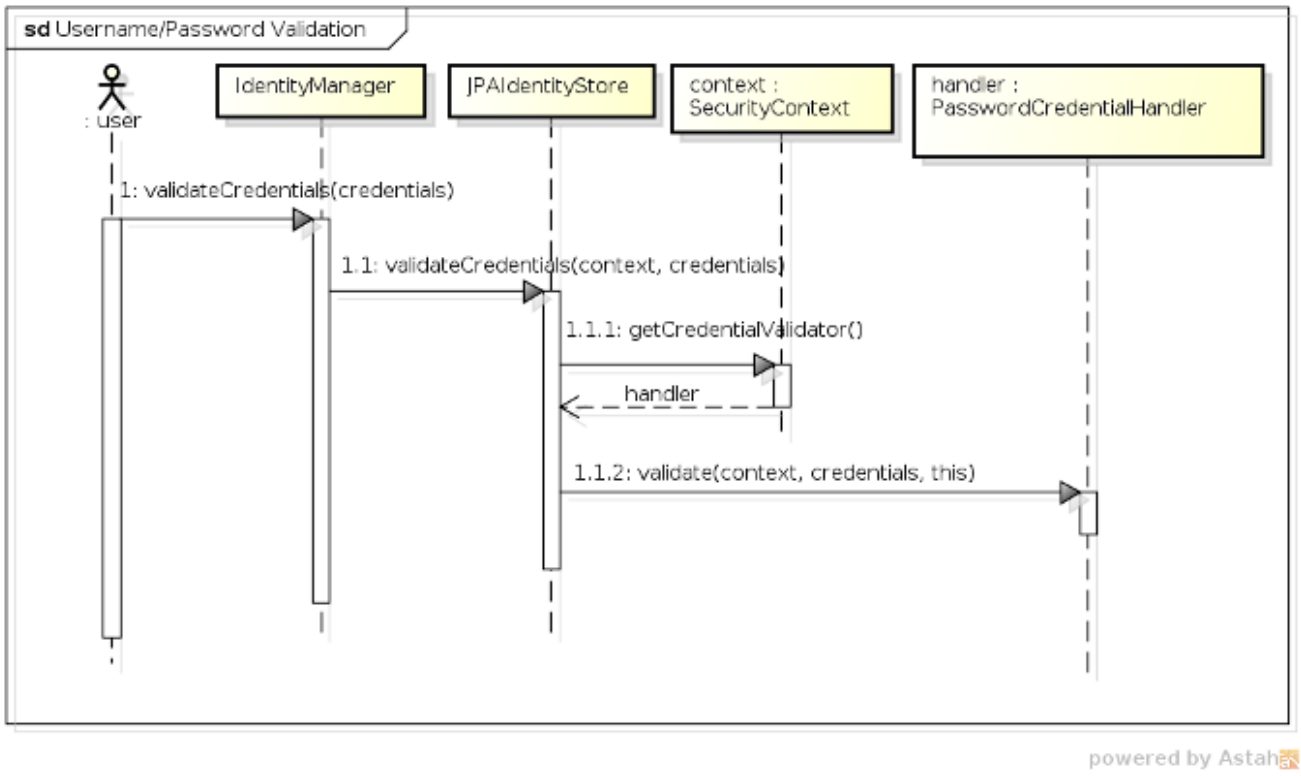
```
User user = BasicModel.getUser(identityManager, "jsmith");
identityManager.updateCredential(user, new Password("abcd1234"));
```

This example creates a digest and assigns it to user "jdoe":

```
User user = BasicModel.getUser(identityManager, "jdoe");
Digest digest = new Digest();
digest.setRealm("default");
digest.setUsername(user.getLoginName());
digest.setPassword("abcd1234");
identityManager.updateCredential(user, digest);
```

4.3. Credential Handlers

For `IdentityStore` implementations that support multiple credential types, PicketLink provides an optional SPI to allow the default credential handling logic to be easily customized and extended. To get a better picture of the overall workings of the Credential Handler SPI, let's take a look at the sequence of events during the credential validation process when validating a username and password against `JPAIdentityStore`:



- *1* - The user (or some other code) first invokes the `validateCredentials()` method on `IdentityManager`, passing in the `Credentials` instance to validate.
- *1.1* - After looking up the correct `IdentityStore` (i.e. the one that has been configured to validate credentials) the `IdentityManager` invokes the store's `validateCredentials()` method, passing in the `IdentityContext` and the credentials to validate.
- *1.1.1* - In `JPAIdentityStore`'s implementation of the `validateCredentials()` method, the `IdentityContext` is used to look up the `CredentialHandler` implementation that has been configured to process validation requests for usernames and passwords, which is then stored in a local variable called `handler`.
- *1.1.2* - The `validate()` method is invoked on the `CredentialHandler`, passing in the security context, the credentials value and a reference back to the identity store. The reference to the identity store is important as the credential handler may require it to invoke certain methods upon the store to validate the credentials.

The `CredentialHandler` interface declares three methods, as follows:

```
public interface CredentialHandler {
    void setup(IdentityStore<?> identityStore);

    void validate(IdentityContext context, Credentials credentials,
                  IdentityStore<?> identityStore);

    void update(IdentityContext context, Account account, Object credential,
                IdentityStore<?> identityStore, Date effectiveDate, Date expiryDate);
```

```
}
```

The `setup()` method is called once, when the `CredentialHandler` instance is first created. Credential handler instantiation is controlled by the `CredentialHandlerFactory`, which creates a single instance of each `CredentialHandler` implementation to service all credential requests for that handler. Each `CredentialHandler` implementation must declare the types of credentials that it is capable of supporting, which is done by annotating the implementation class with the `@SupportsCredentials` annotation like so:

```
@SupportsCredentials(
  credentialClass = { UsernamePasswordCredentials.class, Password.class },
  credentialStorage = EncodedPasswordStorage.class
)
public class PasswordCredentialHandler implements CredentialHandler {
```

Since the `validate()` and `update()` methods receive different parameter types (`validate()` takes a `Credentials` parameter value while `update()` takes an `Object` that represents a single credential value), the `@SupportsCredentials` annotation must contain a complete list of all types supported by that handler.

Similarly, if the `IdentityStore` implementation makes use of the credential handler SPI then it also must declare which credential handlers support that identity store. This is done using the `@CredentialHandlers` annotation; for example, the following code shows how `JPAIdentityStore` is configured to be capable of handling credential requests for usernames and passwords, X509 certificates and digest-based authentication:

```
@CredentialHandlers({ PasswordCredentialHandler.class,
          X509CertificateCredentialHandler.class, DigestCredentialHandler.class })
public class JPAIdentityStore implements IdentityStore<JPAIdentityStoreConfiguration>,
                                         CredentialStore {
```

4.3.1. The `CredentialStore` interface

For `IdentityStore` implementations that support multiple credential types (such as `JPAIdentityStore` and `FileBasedIdentityStore`), the implementation may choose to also implement the `CredentialStore` interface to simplify the interaction between the `CredentialHandler` and the `IdentityStore`. The `CredentialStore` interface declares methods for storing and retrieving credential values within an identity store, as follows:

```
public interface CredentialStore {
   void storeCredential(IdentityContext context, Account account,
                        CredentialStorage storage);
   <T extends CredentialStorage> T retrieveCurrentCredential(IdentityContext context,
                                                 Account account, Class<T> storageClass);
```

```
   <T extends CredentialStorage> List<T> retrieveCredentials(IdentityContext context,
                                                Account account, Class<T> storageClass);
}
```

4.3.2. The `CredentialStorage` interface

The `CredentialStorage` interface is essentially used to represent the state required to validate an account's credentials, and is persisted within the identity store. The base interface is quite simple and only declares two methods - `getEffectiveDate()` and `getExpiryDate()`:

```
public interface CredentialStorage {
   @Stored Date getEffectiveDate();
   @Stored Date getExpiryDate();
}
```

The most significant thing to note above is the usage of the `@Stored` annotation. This annotation is used to mark the properties of the `CredentialStorage` implementation that should be persisted. The only requirement for any property values that are marked as `@Stored` is that they are serializable (i.e. they implement the `java.io.Serializable` interface). The `@Stored` annotation may be placed on either the getter method or the field variable itself. An implementation of `CredentialStorage` will typically declare a number of properties (in addition to the `effectiveDate` and `expiryDate` properties) annotated with `@Stored`. Here's an example of one of a `CredentialStorage` implementation that is built into PicketLink - `EncodedPasswordStorage` is used to store a password hash and salt value:

```
public class EncodedPasswordStorage implements CredentialStorage {

    private Date effectiveDate;
    private Date expiryDate;
    private String encodedHash;
    private String salt;

    @Override @Stored
    public Date getEffectiveDate() {
        return effectiveDate;
    }

    public void setEffectiveDate(Date effectiveDate) {
        this.effectiveDate = effectiveDate;
    }

    @Override @Stored
    public Date getExpiryDate() {
        return expiryDate;
    }

    public void setExpiryDate(Date expiryDate) {
        this.expiryDate = expiryDate;
    }
```

```
    @Stored
    public String getEncodedHash() {
        return encodedHash;
    }

    public void setEncodedHash(String encodedHash) {
        this.encodedHash = encodedHash;
    }

    @Stored
    public String getSalt() {
        return this.salt;
    }

    public void setSalt(String salt) {
        this.salt = salt;
    }

}
```

4.4. Built-in Credential Handlers

PicketLink provides built-in support for the following credential types:

> **Warning**
>
> Not all built-in `IdentityStore` implementations support all credential types. For example, since the `LDAPIdentityStore` is backed by an LDAP directory server, only password credentials are supported. The following table lists the built-in `IdentityStore` implementations that support each credential type.

Table 4.1. Built-in credential types

Credential type	Description	Supported by

`org.picketlink.idm.credential.Password`	A plain text-based password	`JPAIdentityStore`
		`FileBasedIdentityStore`
		`LDAPIdentityStore`

`org.picketlink.idm.credential.Digest`	Used for digest-based authentication	`JPAIdentityStore` `FileBasedIdentityStore`
`java.security.cert.X509Certificate`	Used for X509 certificate based authentication	`JPAIdentityStore` `FileBasedIdentityStore`
`org.picketlink.idm.credential.TOTPCredential`	Used for Time-based One-time Password authentication	`JPAIdentityStore` `FileBasedIdentityStore`

The next sections will describe each of these built-in types individually. Configuration parameters are set at initialization time - see Section 7.1.8.1, “Passing parameters to Credential Handlers” for details.

4.4.1. Username/Password-based Credential Handler

This credential handlers supports a username/password based authentication.

Credentials can be updated as follows:

```
User user = BasicModel.getUser(identityManager, "jsmith");
identityManager.updateCredential(user, new Password("abcd1234"));
```

In order to validate a credential you need to the following code:

```
UsernamePasswordCredentials credential = new UsernamePasswordCredentials();

Password password = new Password("abcd1234");

credential.setUsername("jsmith");
credential.setPassword(password);

identityManager.validateCredentials(credential);

if (Status.VALID.equals(credential.getStatus()) {
  // successful validation
} else {
  // invalid credential
}
```

4.4.1.1. Configuration Parameters

The following table describes all configuration parameters supported by this credential handler:

Table 4.2. Configuration Parameters

Parameter	Description
`PasswordCredentialHandler.PASSWORD_ENCODER`	It must be a `org.picketlink.idm.credential.encoder.PasswordEncoder` sub-type. It defines how passwords are encoded. Defaults to SHA-512.
`PasswordCredentialHandler.SECURE_RANDOM_PROVIDER`	It must be a `org.picketlink.common.random.SecureRandomProvider` sub-type. It defines how `SecureRandom` are created in order to be used to generate random numbers to salt passwords. Defaults to SHA1PRNG with a default seed.
`PasswordCredentialHandler.RENEW_RANDOM_NUMBER_GENERATOR_INTERVAL`	To increase the security of generated salted passwords, `SecureRandom` instances can be renewed from time to time. This option

DIGEST-based Credential Handler

Parameter	Description
	defines the time in milliseconds. Defaults to disabled, what means that a single instance is used during the life-time of the application.
`PasswordCredentialHandler.ALGORITHM_RANDOM_NUMBER`	Defines the algorithm to be used by the default `SecureRandomProvider`. Defaults to SHA1PRNG.
`PasswordCredentialHandler.KEY_LENGTH_RANDOM_NUMBER`	Defines the key length of seeds when using the default `SecureRandomProvider`. Defaults to 0, which means it is disabled.
`PasswordCredentialHandler.LOGIN_NAME_PROPERTY`	This option defines the name of the property used to lookup the `Account` object using the provided login name. It has a default value of `loginName` and can be overridden if the credential handler is to be used to authenticate an `Account` type that uses a different property name.
`PasswordCredentialHandler.SUPPORTED_ACCOUNT_TYPES`	This option defines any additional `Account` types that are supported by the credential handler. If no value is specified and/or no identity instances of the specified types are found then the credential handler's fall back behaviour is to attempt to lookup either an `Agent` or `User` (from the `org.picketlink.idm.model.basic` package) identity. The property value is expected to be an array of `Class<? extends Account>` objects.

4.4.2. DIGEST-based Credential Handler

This credential handlers supports a DIGEST based authentication.

Credentials can be updated as follows:

```
User user = BasicModel.getUser(identityManager, "jsmith");
Digest digest = new Digest();

digest.setRealm("PicketLink Realm");
digest.setUsername(user.getLoginName());
digest.setPassword("abcd1234");

identityManager.updateCredential(user, digest);
```

In order to validate a credential you need to the following code:

```
User user = BasicModel.getUser(identityManager, "jsmith");

Digest digest = new Digest();

digest.setRealm("PicketLink Realm");
digest.setUsername(user.getLoginName());
digest.setPassword("abcd1234");

digest.setDigest(DigestUtil.calculateA1(user.getLoginName(), digest.getRealm(), digest.getPassword().toCharArray()

DigestCredentials credential = new DigestCredentials(digestPassword);

identityManager.validateCredentials(credential);

if (Status.VALID.equals(credential.getStatus()) {
// successful validation
} else {
// invalid credential
}
```

4.4.3. X509-based Credential Handler

This credential handlers supports a X509 certificates based authentication.

Credentials can be updated as follows:

```
User user = BasicModel.getUser(identityManager, "jsmith");

java.security.cert.X509Certificate clientCert = // get user certificate

identityManager.updateCredential(user, clientCert);
```

In order to validate a credential you need to the following code:

```
User user = BasicModel.getUser(identityManager, "jsmith");

java.security.cert.X509Certificate clientCert = // get user certificate
X509CertificateCredentials credential = new X509CertificateCredentials(clientCert);

identityManager.validateCredentials(credential);

if (Status.VALID.equals(credential.getStatus()) {
// successful validation
} else {
// invalid credential
}
```

In some cases, you just want to trust the provided certificate and only check the existence of the principal:

```
User user = BasicModel.getUser(identityManager, "jsmith");

java.security.cert.X509Certificate clientCert = // get user certificate
X509CertificateCredentials credential = new X509CertificateCredentials(clientCert);

// trust the certificate and only check the principal existence
credential.setTrusted(true);

identityManager.validateCredentials(credential);

if (Status.VALID.equals(credential.getStatus()) {
// successful validation
} else {
// invalid credential
}
```

4.4.4. Time-based One Time Password Credential Handler

This credential handlers supports a username/password based authentication.

Credentials can be updated as follows:

```
User user = BasicModel.getUser(identityManager, "jsmith");

TOTPCredential credential = new TOTPCredential("abcd1234", "my_totp_secret");

identityManager.updateCredential(user, credential);
```

Users can have multiple TOTP tokens, one for each device. You can provide configure tokens for a specific user device as follows:

```
User user = BasicModel.getUser(identityManager, "jsmith");

TOTPCredential credential = new TOTPCredential("abcd1234", "my_totp_secret");

credential.setDevice("My Cool Android Phone");

identityManager.updateCredential(user, credential);
```

In order to validate a credential you need to the following code:

```
User user = BasicModel.getUser(identityManager, "jsmith");

TOTPCredentials credential = new TOTPCredentials();
```

```
credential.setUsername(user.getLoginName());
credential.setPassword(new Password("abcd1234"));

TimeBasedOTP totp = new TimeBasedOTP();

// let's manually generate a token based on the user secret
String token = totp.generate("my_totp_secret");

credential.setToken(token);

// if you want to validate the token for a specific device
// credential.setDevice("My Cool Android Phone");

identityManager.validateCredentials(credential);

if (Status.VALID.equals(credential.getStatus()) {
  // successful validation
} else {
  // invalid credential
}
```

4.4.4.1. Configuration Parameters

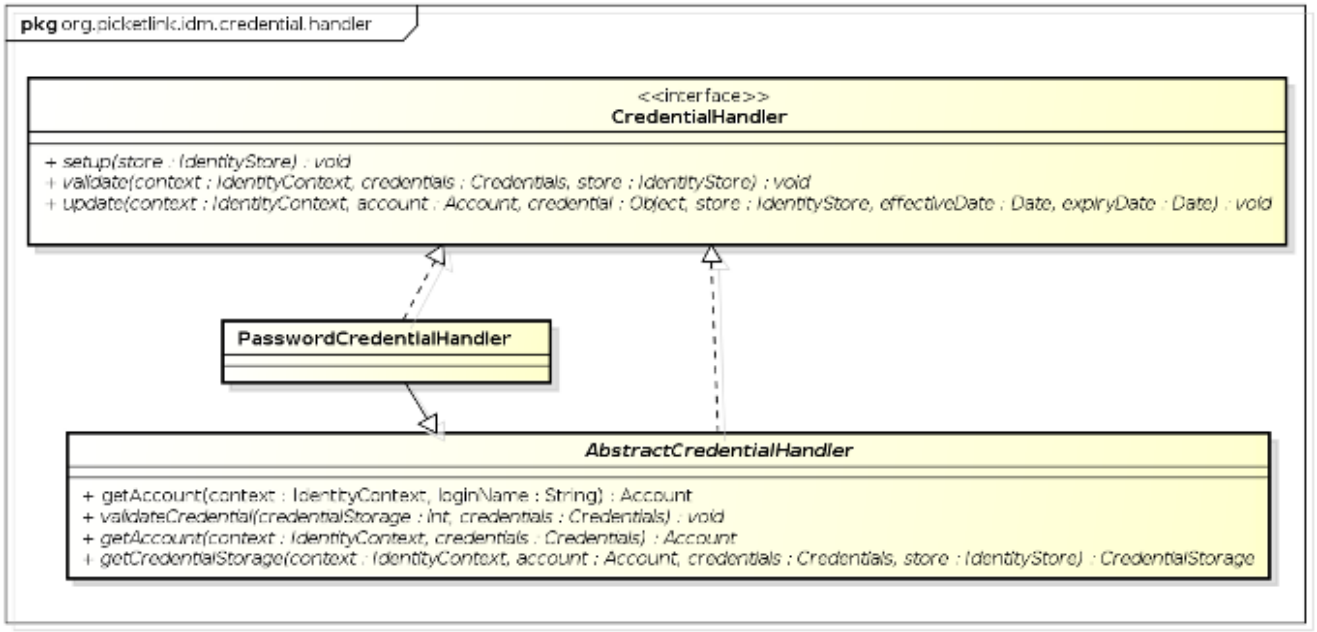
The following table describes all configuration parameters supported by this credential handler:

Table 4.3. Configuration Parameters

Parameter	Description
`TOTPCredentialHandler.ALGORITHM`	The encryption algorithm. Defaults to HmacSHA1.
`TOTPCredentialHandler.INTERVAL_SECONDS`	The number of seconds a token is valid. Defaults to 30 seconds.
`TOTPCredentialHandler.NUMBER_DIGITS`	The number of digits for a token. Defaults to 6 digits.
`TOTPCredentialHandler.DELAY_WINDOW`	the number of previous intervals that should be used to validate tokens. Defaults to 1 interval of 30 seconds.

4.5. Implementing a Custom `CredentialHandler`

In this section we'll dissect the `PasswordCredentialHandler` to learn how to create a custom credential handler. The `AbstractCredentialHandler` abstract class is provided to simplify the process of creating a new credential handler, and is also used by `PasswordCredentialHandler` as a base class:



Let's start by looking at the class declaration for `PasswordCredentialHandler`:

```
@SupportsCredentials(
        credentialClass = {UsernamePasswordCredentials.class, Password.class},
        credentialStorage = EncodedPasswordStorage.class)
public class PasswordCredentialHandler<S extends CredentialStore<?>,
    V extends UsernamePasswordCredentials,
    U extends Password>
    extends AbstractCredentialHandler<S, V, U> {
```

The `@SupportsCredentials` annotation is used to declare exactly which credential classes are supported by the credential handler (as indicated by the `credentialClass` annotation member). The supported credential classes include both credentials used to authenticate via the `validate()` method (i.e. a class that implements the `org.picketlink.idm.credential.Credentials` interface, in this example `UsernamePasswordCredentials`) and the actual encapsulated credential value (e.g. `org.picketlink.idm.credential.Password`). These supported credential classes are also reflected in the generic type declaration of the class itself - the `V` and `U` types in the code above. The `credentialStorage` annotation member declares the storage class used to persist the necessary state for the credential. The storage class must implement the `org.picketlink.idm.credential.storage.CredentialStorage` interface.

The `setup()` method is executed only once and is used to perform any required initialization for the credential handler. In the case of `PasswordCredentialHandler`, the `setup()` method reads the configuration properties (made available from `store.getConfig().getCredentialHandlerProperties()`) and uses those property values to initialize the state required for encoding password values.

```
@Override
public void setup(S store) {
    super.setup(store);

    Map<String, Object> options = store.getConfig().getCredentialHandlerProperties();

    if (options != null) {
        Object providedEncoder = options.get(PASSWORD_ENCODER);

        if (providedEncoder != null) {
            if (PasswordEncoder.class.isInstance(providedEncoder)) {
                this.passwordEncoder = (PasswordEncoder) providedEncoder;
            } else {
                throw new SecurityConfigurationException("The password encoder [" +
                    providedEncoder + "] must be an instance of " +
                    PasswordEncoder.class.getName());
            }
        }

        Object renewRandomNumberGeneratorInterval = options.get(
            RENEW_RANDOM_NUMBER_GENERATOR_INTERVAL);

        if (renewRandomNumberGeneratorInterval != null) {
            this.renewRandomNumberGeneratorInterval = Integer.valueOf(
                renewRandomNumberGeneratorInterval.toString());
        }

        Object secureRandomProvider = options.get(SECURE_RANDOM_PROVIDER);

        if (secureRandomProvider != null) {
            this.secureRandomProvider = (SecureRandomProvider) secureRandomProvider;
        } else {
            Object saltAlgorithm = options.get(ALGORITHM_RANDOM_NUMBER);

            if (saltAlgorithm == null) {
                saltAlgorithm = DEFAULT_SALT_ALGORITHM;
            }

            Object keyLengthRandomNumber = options.get(KEY_LENGTH_RANDOM_NUMBER);

            if (keyLengthRandomNumber == null) {
                keyLengthRandomNumber = Integer.valueOf(0);
            }

            this.secureRandomProvider = new DefaultSecureRandomProvider(
                saltAlgorithm.toString(),
                Integer.valueOf(keyLengthRandomNumber.toString()));
        }
    }

    this.secureRandom = createSecureRandom();
}
```

The credential validation logic is defined by the `validateCredential()` method. This method (which the parent `AbstractCredentialHandler` class declares as an abstract method) checks the validity of the credential value passed in and either returns `true` if the credential is valid or

`false` if it is not. The `validateCredential()` method is a convenience method which delegates much of the boilerplate code required for credential validation to `AbstractCredentialHandler`, allowing the subclass to simply define the bare minimum code required to validate the credential. If you were to implement a `CredentialHandler` without using `AbstractCredentialHandler` as a base class, you would instead need to implement the `validate()` method which in general requires a fair bit more code.

```
@Override
protected boolean validateCredential(final CredentialStorage storage,
    final V credentials) {
    EncodedPasswordStorage hash = (EncodedPasswordStorage) storage;

    if (hash != null) {
        String rawPassword = new String(credentials.getPassword().getValue());
        return this.passwordEncoder.verify(saltPassword(rawPassword,
            hash.getSalt()), hash.getEncodedHash());
    }

    return false;
}
```

The `update()` method (in contrast to `validateCredential`) *is* defined by the `CredentialHandler` interface itself, and is used to persist a credential value to the backend identity store. In `PasswordCredentialHandler` this method creates a new instance of `EncodedPasswordStorage`, a `CredentialStorage` implementation that represents a password's hash and salt values. The salt value in this implementation is randomly generated using the configured property values, and then used to encode the password hash. This value is then stored by calling the `store.storeCredential()` method.

```
@Override
public void update(IdentityContext context, Account account, U password, S store,
                   Date effectiveDate, Date expiryDate) {

    EncodedPasswordStorage hash = new EncodedPasswordStorage();

    if (password.getValue() == null || isNullOrEmpty(password.getValue().toString())) {
        throw MESSAGES.credentialInvalidPassword();
    }

    String rawPassword = new String(password.getValue());

    String passwordSalt = generateSalt();

    hash.setSalt(passwordSalt);
    hash.setEncodedHash(this.passwordEncoder.encode(saltPassword(rawPassword,
        passwordSalt)));

    if (effectiveDate != null) {
        hash.setEffectiveDate(effectiveDate);
    }
```

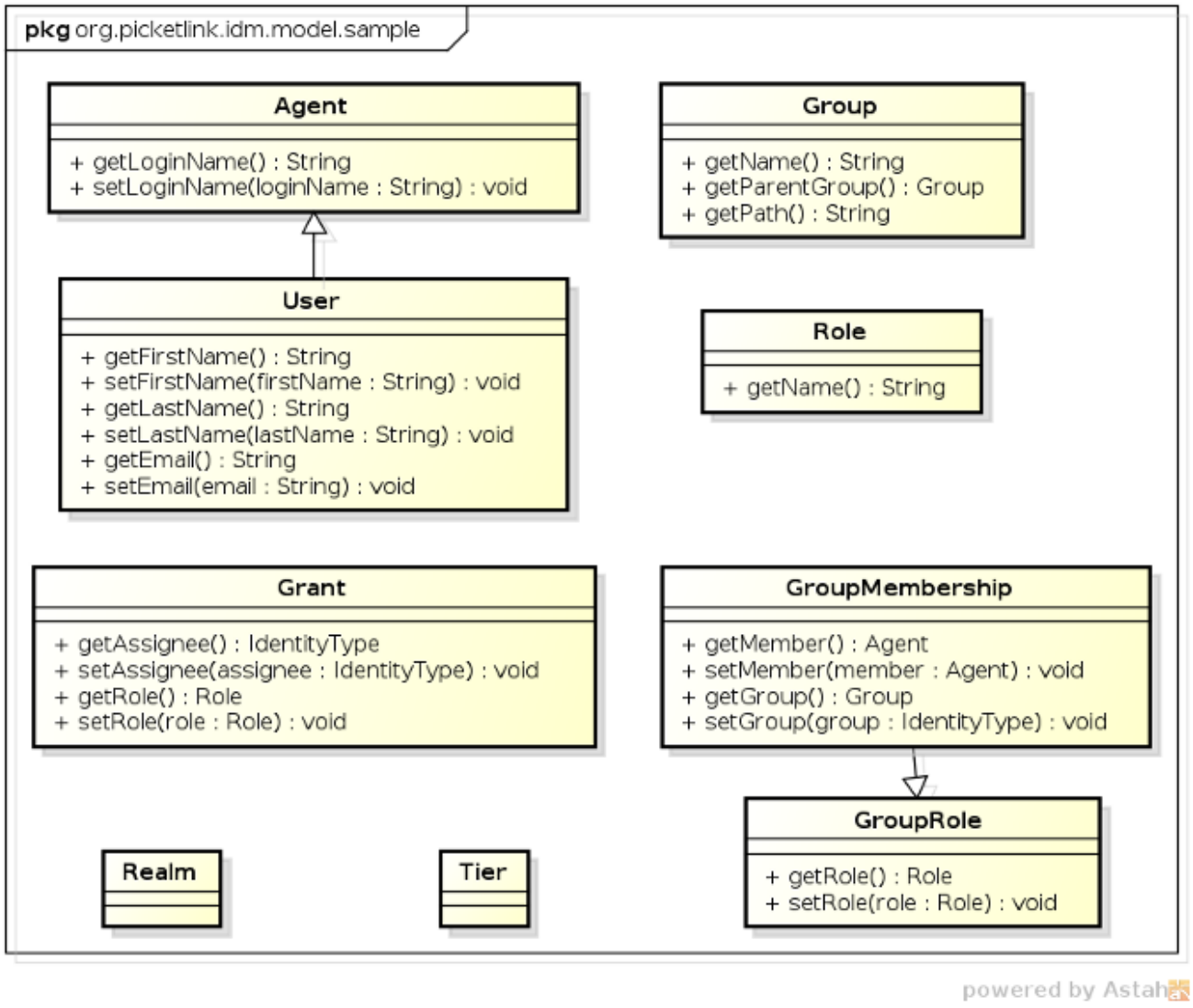
```
    hash.setExpiryDate(expiryDate);

    store.storeCredential(context, account, hash);
}
```

Chapter 5. Identity Management - Basic Identity Model

5.1. Basic Identity Model

For the sake of convenience, PicketLink provides a basic identity model that consists of a number of core interfaces which define a set of fundamental identity types which might be found in a typical application. The usage of this identity model is entirely optional; for an application with basic security requirements the basic identity model might be more than sufficient, however for a more complex application or application with custom security requirements it may be necessary to create a custom identity model.

The following class diagram shows the classes and interfaces in the `org.picketlink.idm.model.basic` package:



Utility Class for the Basic Identity Model

- `Agent` represents a unique entity that may access the services secured by PicketLink. In contrast to a user which represents a human, `Agent` is intended to represent a third party non-human (i.e. machine to machine) process that may authenticate and interact with your application or services. It declares methods for reading and setting the `Agent`'s login name.

- `User` represents a human user that accesses your application and services. In addition to the login name property defined by its parent interface `Agent`, the `User` interface declares a number of other methods for managing the user's first name, last name and e-mail address.
- `Group` is used to manage collections of identity types. Each `Group` has a name and an optional parent group.
- `Role` is used in various relationship types to designate authority to another identity type to perform various operations within an application. For example, a forum application may define a role called *moderator* which may be assigned to one or more `User`s or `Group`s to indicate that they are authorized to perform moderator functions.
- `Grant` relationship represents the assignment of a `Role` to an identity.
- `GroupMembership` relationship represents a `User` (or `Agent`) membership within a `Group`.
- `GroupRole` relationship represents the the assignment of a specific `Role` within a `Group` to a `User` or `Agent`. The reason this relationship extends the `GroupMembership` relationship is simply so it inherits the `getMember()` and `getGroup()` methods - being assigned to a `GroupRole` *does not* mean that the `User` (or `Agent`) that was assigned the group role also becomes a member of the group.
- `Realm` is a partition type, and may be used to store any `IdentityType` objects (including `Agent`s, `User`s, `Group`s or `Role`s.
- `Tier` is a specialized partition type, and may be only used to store `Group` or `Role` objects specific to an application.

5.1.1. Utility Class for the Basic Identity Model

PicketLink also provides an utility class with some very useful and common methods to manipulate the basic identity model. Along the documentation you'll find a lot of examples using the the following class: `org.picketlink.idm.model.basic.BasicModel`. If you're using the basic identity model, this helper class can save you a lot of code and make your application even more simple.

The list below summarizes some of the functionalities provided by this class:

- Retrieve `User` and `Agent` instances by login name.
- Retrieve `Role` and `Group` instances by name.

- Add users as group members, grant roles to users. As well check if an user is member of a group or has a specific role.

One import thing to keep in mind is that the `BasicModel` helper class is only suitable if you're using the types provided by the basic identity model, only. If you are using custom types, even if those are sub-types of any of the types provided by the basic identity model, you should handle those custom types directly using the PicketLink IDM API.

As an example, let's suppose you have a custom type which extends the `Agent` type.

```
SalesAgent salesAgent = BasicModel.getUser(identityManager, "someSalesAgent");
```

The code above will never return a `SalesAgent` instance. The correct way of doing that is using the Query API directly as follows:

```
public SaleAgent findSalesAgent(String loginName) {
    List<SaleAgent> result = identityManager
                             .createIdentityQuery(SaleAgent.class)
                             .setParameter(SaleAgent.LOGIN_NAME, loginName)
                             .getResultList();
    return result.isEmpty() ? null : result.get(0);
}
```

> **Warning**
>
> Please note that the `BasicModel` helper class is only suitable for use cases where only the types provided by the basic identity model are used. If your application have also custom types, they need to be handled directly using the PicketLink IDM API.

5.2. Managing Users, Groups and Roles

PicketLink IDM provides a number of basic implementations of the identity model interfaces for convenience, in the `org.picketlink.idm.model.basic` package. The following sections provide examples that show these implementations in action.

5.2.1. Managing Users

The following code example demonstrates how to create a new user with the following properties:

- Login name - *jsmith*
- First name - *John*
- Last name - *Smith*

- E-mail - *jsmith@acme.com*

```
User user = new User("jsmith");
user.setFirstName("John");
user.setLastName("Smith");
user.setEmail("jsmith@acme.com");
identityManager.add(user);
```

Once the `User` is created, it's possible to look it up using its login name:

```
User user = BasicModel.getUser(identityManager, "jsmith");
```

User properties can also be modified after the User has already been created. The following example demonstrates how to change the e-mail address of the user we created above:

```
User user = BasicModel.getUser(identityManager, "jsmith");
user.setEmail("john@smith.com");
identityManager.update(user);
```

Users may also be deleted. The following example demonstrates how to delete the user previously created:

```
User user = BasicModel.getUser(identityManager, "jsmith");
identityManager.remove("jsmith");
```

5.2.2. Managing Groups

The following example demonstrates how to create a new group called *employees*:

```
Group employees = new Group("employees");
```

It is also possible to assign a parent group when creating a group. The following example demonstrates how to create a new group called *managers*, using the *employees* group created in the previous example as the parent group:

```
Group managers = new Group("managers", employees);
```

To lookup an existing `Group`, the `getGroup()` method may be used. If the group name is unique, it can be passed as a single parameter:

```
Group employees = BasicModel.getGroup(identityManager, "employees");
```

If the group name is not unique, the parent group must be passed as the second parameter (although it can still be provided if the group name is unique):

```
Group managers = BasicModel.getGroup(identityManager, "managers", employees);
```

It is also possible to modify a `Group`'s name and other properties (besides its parent) after it has been created. The following example demonstrates how to disable the "employees" group we created above:

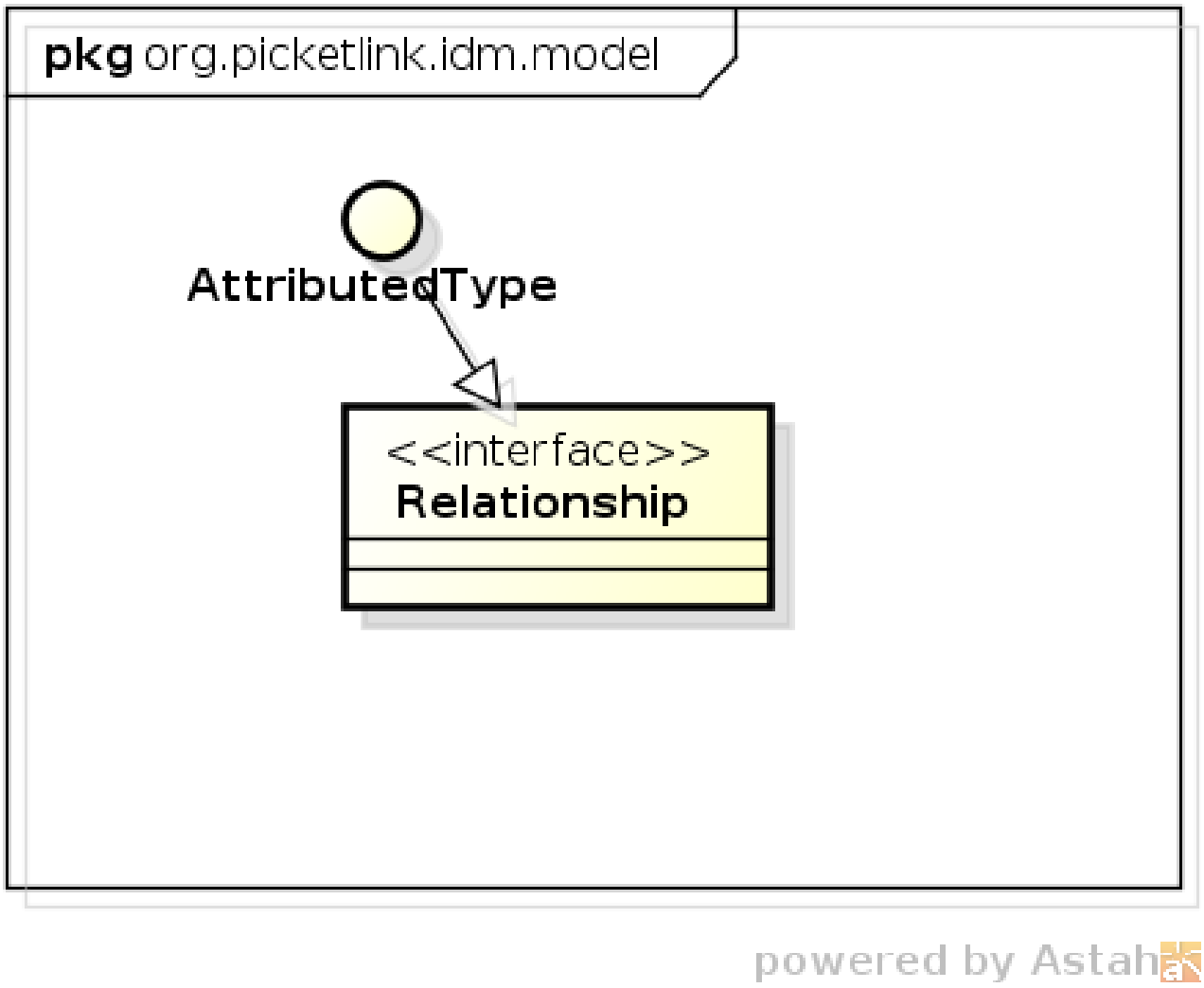
```
Group employees = BasicModel.getGroup(identityManager, "employees");
employees.setEnabled(false);
identityManager.update(employees);
```

To remove an existing group, we can use the `remove()` method:

```
Group employees = BasicModel.getGroup(identityManager, "employees");
identityManager.remove(employees);
```

5.3. Managing Relationships

Relationships are used to model *typed associations* between two or more identities. All concrete relationship types must implement the marker interface `org.picketlink.idm.model.Relationship`:



The `IdentityManager` interface provides three standard methods for managing relationships:

```
void add(Relationship relationship);
void update(Relationship relationship);
void remove(Relationship relationship);
```

- The `add()` method is used to create a new relationship.
- The `update()` method is used to update an existing relationship.

> **Note**
>
> Please note that the identities that participate in a relationship cannot be updated themselves, however the attribute values of the relationship can be updated. If

you absolutely need to modify the identities of a relationship, then delete the relationship and create it again.

- The `remove()` method is used to remove an existing relationship.

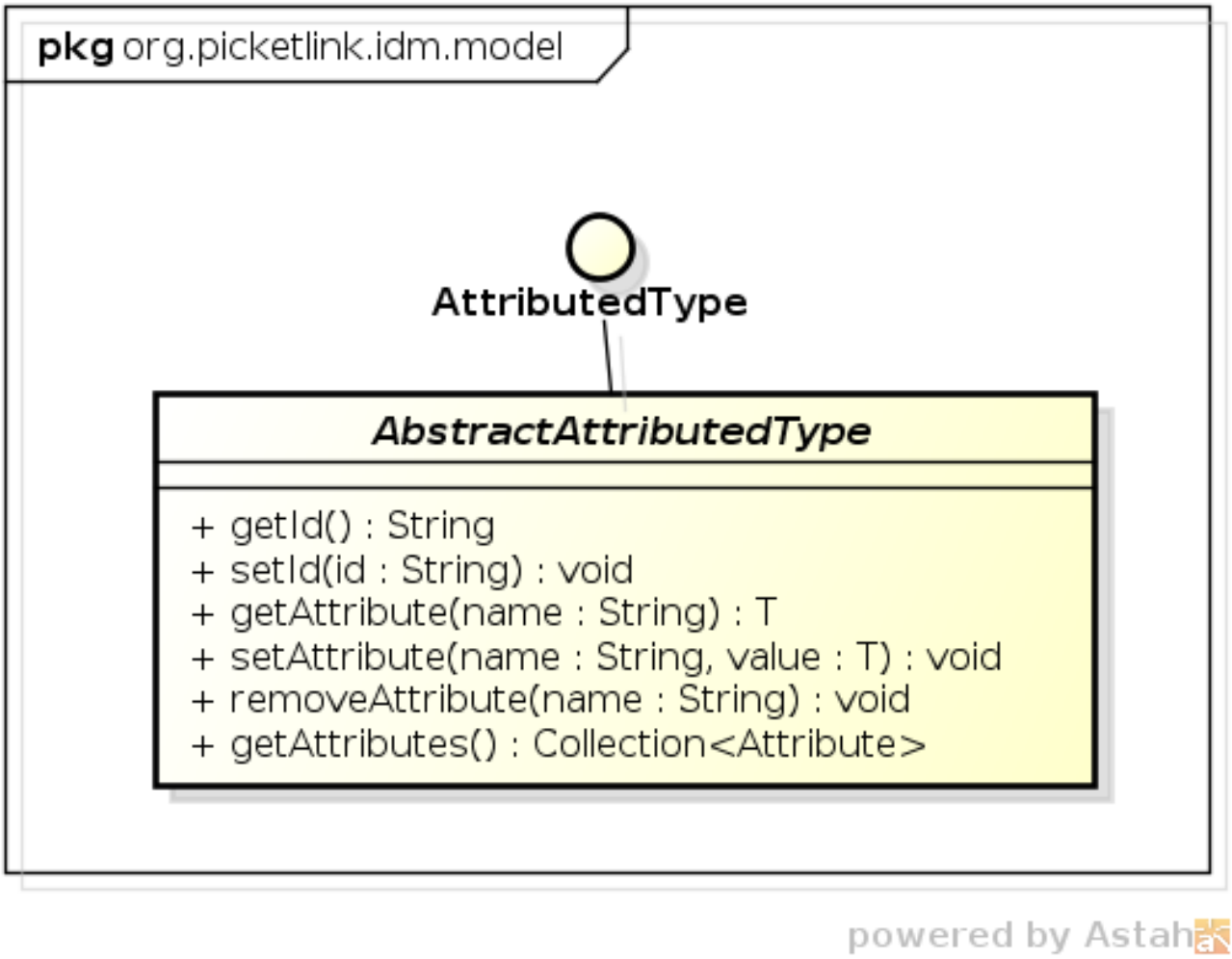
Note

To search for existing relationships between identity objects, use the Relationship Query API described later in this chapter.

Besides the above methods, `IdentityManager` also provides a number of convenience methods for managing many of the built-in relationship types. See the next section for more details.

5.3.1. Built In Relationship Types

PicketLink provides a number of built-in relationship types, designed to address the most common requirements of a typical application. The following sections describe the built-in relationships and how they are intended to be used. Every built-in relationship type extends the `AbstractAttributedType` abstract class, which provides the basic methods for setting a unique identifier value and managing a set of attribute values:

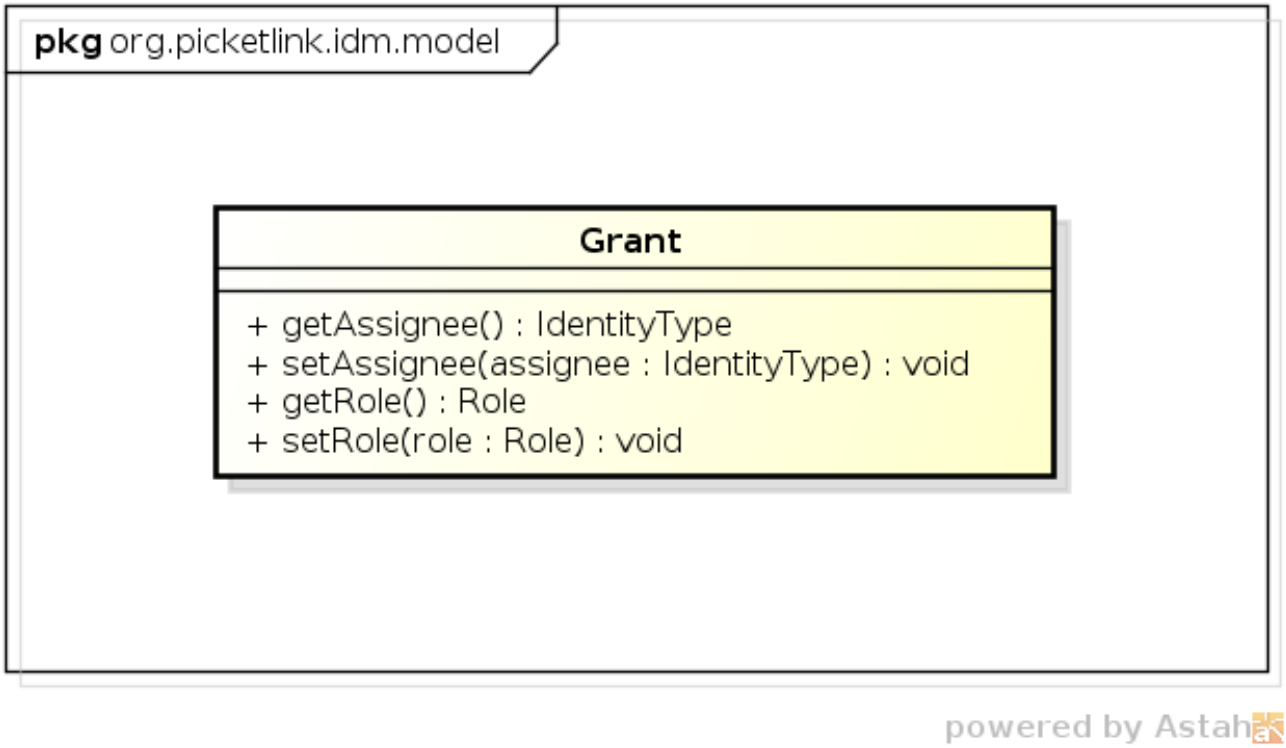


What this means in practical terms, is that every single relationship is assigned and can be identified by, a unique identifier value. Also, arbitrary attribute values may be set for all relationship types, which is useful if you require additional metadata or any other type of information to be stored with a relationship.

5.3.1.1. Application Roles

Application roles are represented by the `Grant` relationship, which is used to assign application-wide privileges to a `User` or `Agent`.

Built
In
Relationship



The `IdentityManager` interface provides methods for directly granting a role. Here's a simple example:

```
User bob = BasicModel.getUser(identityManager, "bob");
Role superuser = BasicModel.getRole(identityManager, "superuser");
BasicModel.grantRole(relationshipManager, bob, superuser);
```

The above code is equivalent to the following:

```
User bob = BasicModel.getUser(identityManager, "bob");
Role superuser = BasicModel.getRole(identityManager, "superuser");
Grant grant = new Grant(bob, superuser);
identityManager.add(grant);
```

A granted role can also be revoked using the `revokeRole()` method:

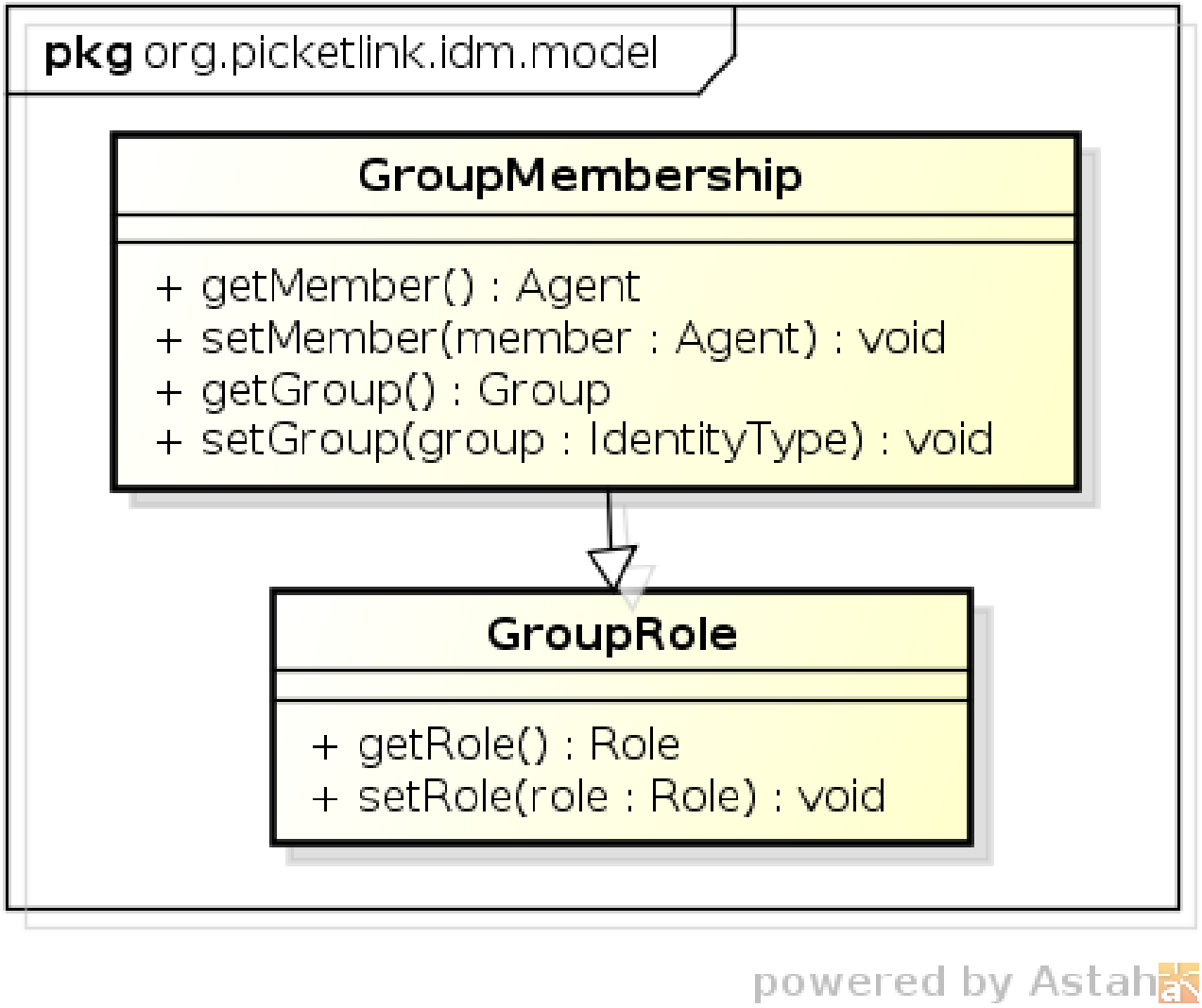
```
User bob = BasicModel.getUser(identityManager, "bob");
Role superuser = BasicModel.getRole(identityManager, "superuser");
BasicModel.revokeRole(relationshipManager, bob, superuser);
```

To check whether an identity has a specific role granted to them, we can use the `hasRole()` method:

```
User bob = BasicModel.getUser(identityManager, "bob");
Role superuser = BasicModel.getRole(identityManager, "superuser");
boolean isBobASuperUser = BasicModel.hasRole(relationshipManager, bob, superuser);
```

5.3.1.2. Groups and Group Roles

The `GroupMembership` and `GroupRole` relationships are used to represent a user's membership within a `Group`, and a user's role for a group, respectively.



Note

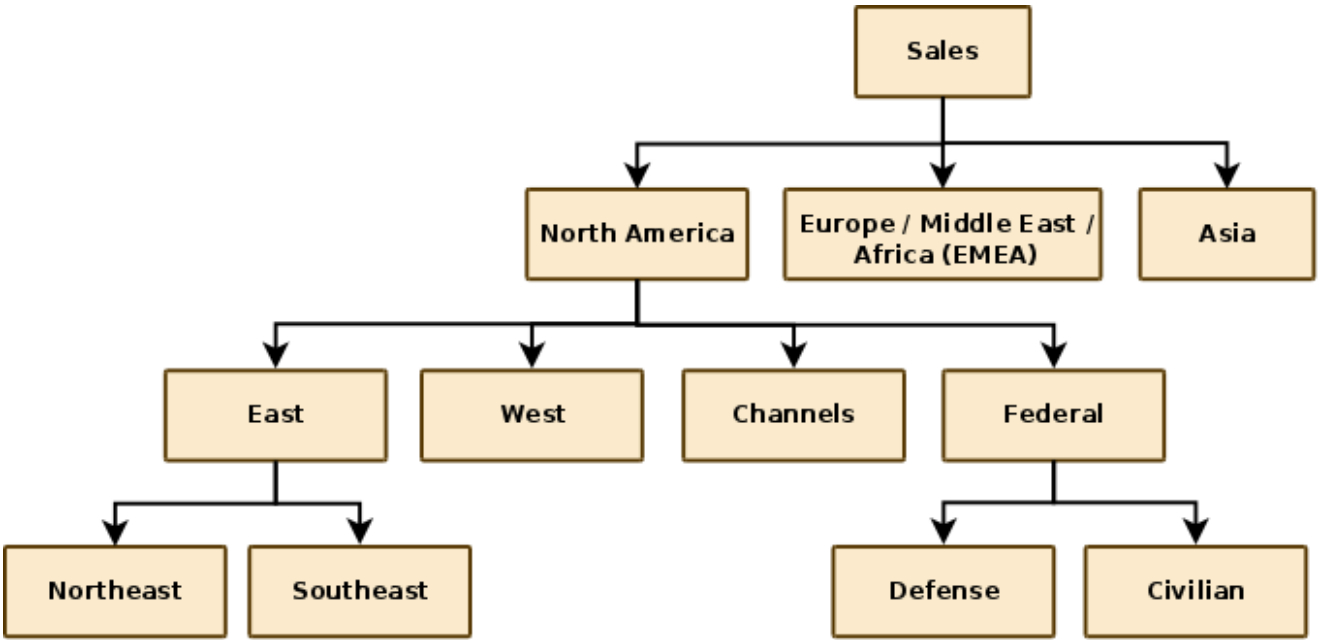
While the `GroupRole` relationship type extends `GroupMembership`, it does *not* mean that a member of a `GroupRole` automatically receives `GroupMembership`

membership also - these are two distinct relationship types with different semantics.

A `Group` is typically used to form logical collections of users. Within an organisation, groups are often used to mirror the organisation's structure. For example, a corporate structure might consist of a sales department, administration, management, etc. This structure can be modelled in PicketLink by creating corresponding groups such as *sales*, *administration*, and so forth. Users (who would represent the employees in a corporate structure) may then be assigned group memberships corresponding to their place within the company's organisational structure. For example, an employee who works in the sales department may be assigned to the *sales* group. Specific application privileges can then be blanket assigned to the *sales* group, and anyone who is a member of the group is free to access the application's features that require those privileges.

The `GroupRole` relationship type should be used when it is intended for an identity to perform a specific role for a group, but not be an actual member of the group itself. For example, an administrator of a group of doctors may not be a doctor themselves, but have an administrative role to perform for that group. If the intent is for an individual identity to both be a member of a group *and* have an assigned role in that group also, then the identity should have both `GroupRole` and `GroupMembership` relationships for that group.

Let's start by looking at a simple example - we'll begin by making the assumption that our organization is structured in the following way:



The following code demonstrates how we would create the hypothetical *Sales* group which is displayed at the head of the above organisational chart:

```
Group sales = new Group("Sales");
identityManager.add(sales);
```

We can then proceed to create its subgroups:

```
identityManager.add(new Group("North America", sales);
identityManager.add(new Group("EMEA", sales);
identityManager.add(new Group("Asia", sales);
// and so forth
```

The second parameter of the `Group()` constructor is used to specify the group's parent group. This allows us to create a hierarchical group structure, which can be used to mirror either a simple or complex personnel structure of an organisation. Let's now take a look at how we assign users to these groups.

The following code demonstrates how to assign an *administrator* group role for the *Northeast* sales group to user *jsmith*. The *administrator* group role may be used to grant certain users the privilege to modify permissions and roles for that group:

```
Role admin = BasicModel.getRole(identityManager, "administrator");
User user = BasicModel.getUser(identityManager, "jsmith");
Group group = BasicModel.getGroup(identityManager, "Northeast");
BasicModel.grantGroupRole(relationshipManager, user, admin, group);
```

A group role can be revoked using the `revokeGroupRole()` method:

```
BasicModel.revokeGroupRole(relationshipManager, user, admin, group);
```

To test whether a user has a particular group role, you can use the `hasGroupRole()` method:

```
boolean isUserAGroupAdmin = BasicModel.hasGroupRole(relationshipManager, user, admin, group);
```

Next, let's look at some examples of how to work with simple group memberships. The following code demonstrates how we assign sales staff *rbrown* to the *Northeast* sales group:

```
User user = BasicModel.getUser(identityManager, "rbrown");
Group group = BasicModel.getGroup(identityManager, "Northeast");
BasicModel.addToGroup(relationshipManager, user, group);
```

A `User` may also be a member of more than one `Group`; there are no built-in limitations on the number of groups that a `User` may be a member of.

We can use the `removeFromGroup()` method to remove the same user from the group:

```
BasicModel.removeFromGroup(relationshipManager, user, group);
```

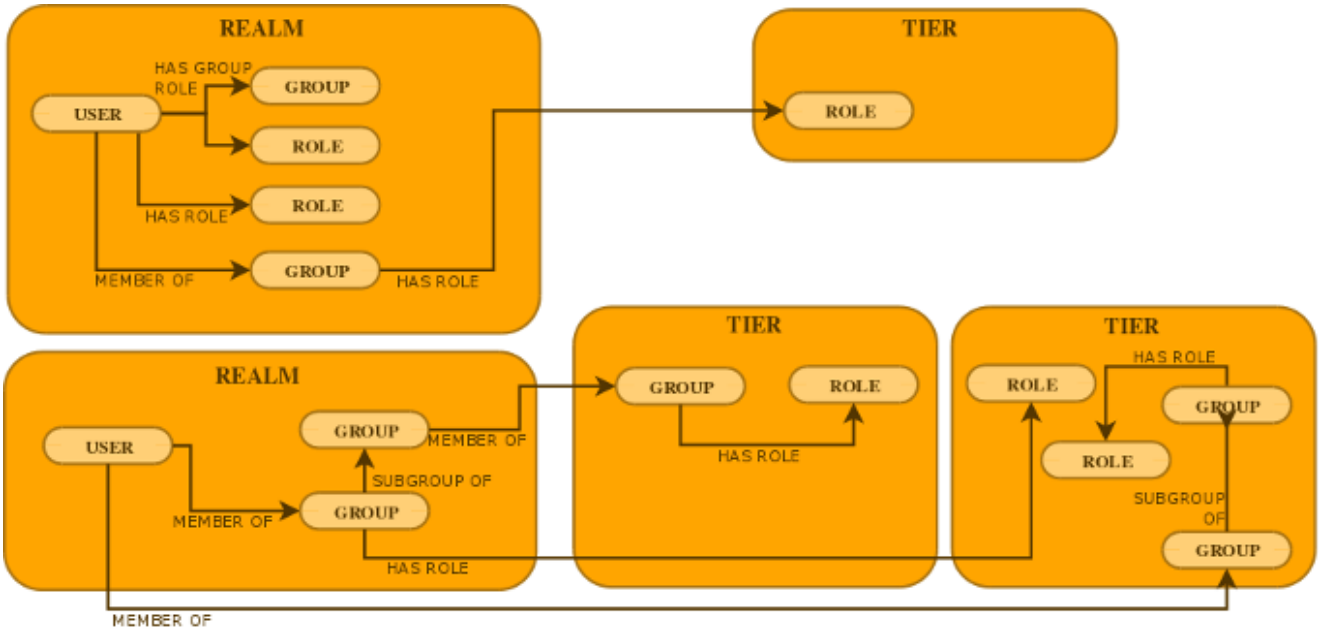
To check whether a user is the member of a group we can use the `isMember()` method:

```
boolean isUserAMember = BasicModel.isMember(relationshipManager, user, group);
```

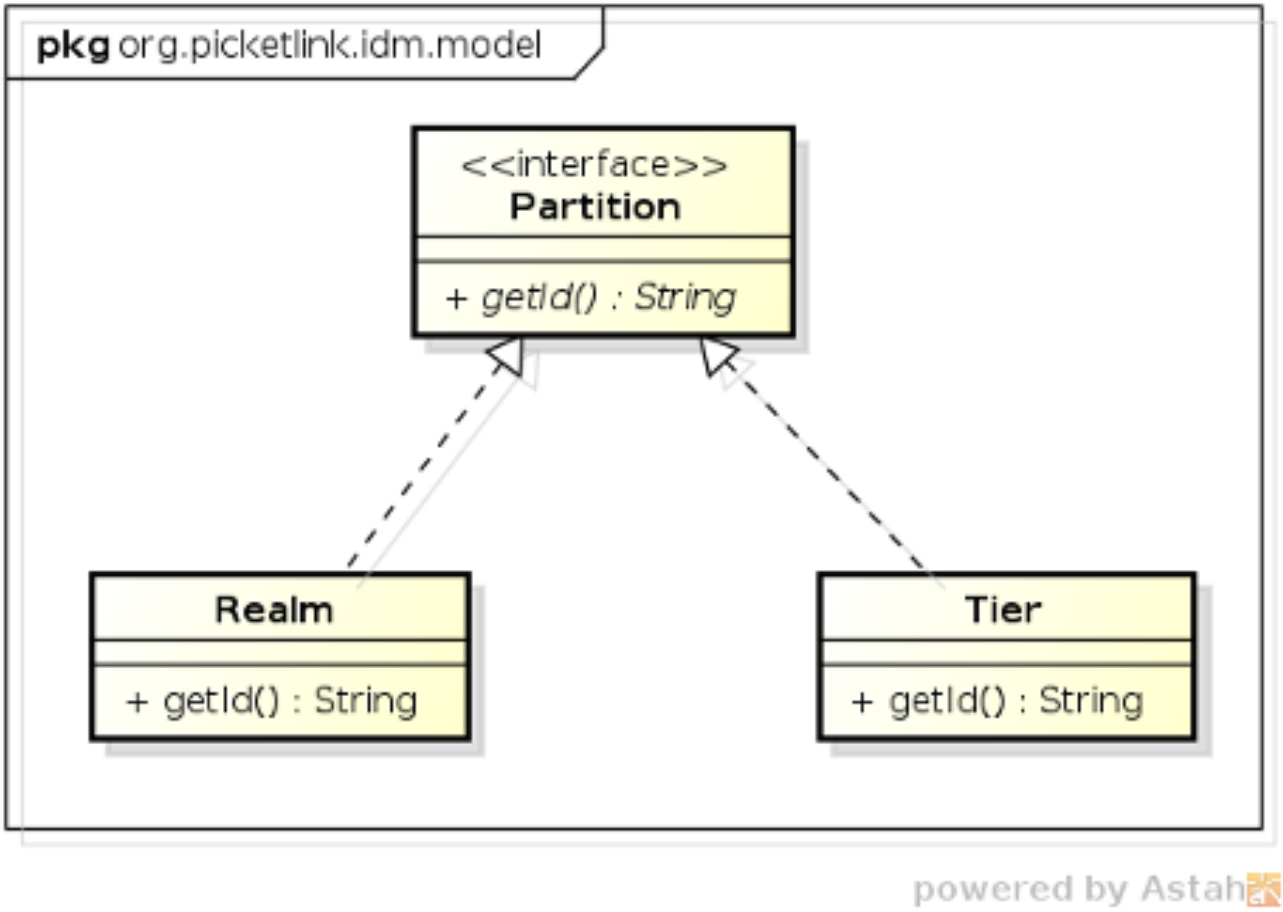
Relationships can also be created via the `add()` method. The following code is equivalent to assigning a group role via the `grantGroupRole()` method shown above:

```
Role admin = BasicModel.getRole(identityManager, "administrator");
User user = BasicModel.getUser(identityManager, "jsmith");
Group group = BasicModel.getGroup(identityManager, "Northeast");
GroupRole groupRole = new GroupRole(user, group, admin);
identityManager.add(groupRole);
```

5.4. Realms and Tiers



In terms of API, both the `Realm` and `Tier` classes implement the `Partition` interface, as shown in the following class diagram:



A *Realm* is used to define a discrete set of users, groups and roles. A typical use case for realms is the segregation of corporate user accounts within a multi-tenant application, although it is not limited this use case only. As all identity management operations must be performed within the context of an *active partition*, PicketLink defines the concept of a *default realm* which becomes the active partition if no other partition has been specified.

A *Tier* is a more restrictive type of partition than a realm, as it only allows groups and roles to be defined (but not users). A Tier may be used to define a set of application-specific groups and roles, which may then be assigned to groups within the same Tier, or to users and groups within a separate Realm.

Chapter 6. Identity Management - Attribute Management

6.1. Overview

PicketLink IDM classifies attributes in two main categories: *formal* and *ad-hoc* attributes.

Usually all attributes you need for a specific type is mapped to its properties. We call those attributes *formal attributes.*. Formal attributes are properties managed by PicketLink and defined directly in your types. By managed, we mean that PicketLink knows how to get the value to store from a specific property and also retrieve its value from the store and populate back to an instance of your type.

But sometimes you may need to define attributes that are not mapped to a specific property of your type. Specially if those are dynamic attributes, whose existence depends on a specific context or rule. We call those attributes *ad-hoc attributes*. Ad-hoc attributes are not strong-typed as formal attributes, they are just a key/value pair. Where the key is the attribute's name and the latter its value.

6.2. Formal attributes

Usually all attributes are formal attributes, in the sense that properties of a type are managed by PicketLink. In order to get them managed by PicketLink you need to annotate each property of your type with Section 3.4.1, “The `@AttributeProperty` Annotation”

If you take the `User` type as an example, you'll see that all its properties are defined as follows:

```
public class User extends Agent {

  @AttributeProperty
  private String firstName;

  @AttributeProperty
  private String lastName;

  @AttributeProperty
  private String email;

}
```

Formal attributes are strongly-typed as they are directly defined as properties in your type.

Those attributes are also queriable. Which means you can use the Query API to search for types with a specific property and value. If a property is queriable, we recommend to always create a `QueryParameter` constant as follows:

```
public class User extends Agent {

  /**
   * A query parameter used to set the firstName value.
   */
  public static final QueryParameter FIRST_NAME = QUERY_ATTRIBUTE.byName("firstName");

  /**
   * A query parameter used to set the lastName value.
   */
  public static final QueryParameter LAST_NAME = QUERY_ATTRIBUTE.byName("lastName");

  /**
   * A query parameter used to set the email value.
   */
  public static final QueryParameter EMAIL = QUERY_ATTRIBUTE.byName("email");

  @AttributeProperty
  private String firstName;

  @AttributeProperty
  private String lastName;

  @AttributeProperty
  private String email;

}
```

Once the query parameters are defined and mapped to your properties, you can search for types based on its properties as follows:

```
IdentityQuery<User> query = identityManager.<User> createIdentityQuery(User.class);

  query.setParameter(User.FIRST_NAME, "John");

  // find only by the first name
  List<User> result = query.getResultList();

  for (User user : result) {
    // do something
  }
```

6.3. Ad-hoc attributes

The best way to understand ad-hoc attributes is look how they're defined. The example below uses three attributes to define some security questions for an user.

```
User user = new User("john");

  user.setAttribute(new Attribute<Integer>("QuestionTotal", 2);
```

```
  // attribute for question #1
  user.setAttribute(new Attribute<String>("Question1", "What is favorite toy?"));
  user.setAttribute(new Attribute<String>("Question1Answer", "Gum"));

  // attribute for question #2
  user.setAttribute(new Attribute<String>("Question2", "What is favorite word?"));
  user.setAttribute(new Attribute<String>("Question2Answer", "Hi"));

identityManager.add(user);
```

Ad-hoc attributes are not strong-typed as formal attributes, they are just a key/value pair. Where the key is the attribute's name and the latter its value. The key point here is that they offer great flexibility to your identity model as you can define any attribute you want without changing any type.

> **Note**
>
> Note that ad-hoc attributes are not typed. You should use them wisely, otherwise they may become unmanageable.

In theory, all PicketLink types such as `IdentityType`, `Relationship` and `Partition` support ad-hoc attributes. The reason is that all those types are a subtypes of `AttributedType`. But in practice, even if a type support ad-hoc attributes the underlying store may not. Which means you can not use those attributes. The LDAP Identity Store, for example, does not support ad-hoc attributes at all if used *alone*. Take a look at Chapter 9, *Identity Management - Working with LDAP* for more details about how to support ad-hoc attributes when using the LDAP store.

Those attributes are also queriable. The example below shows how you can use the Query API to search for types using ad-hoc attributes:

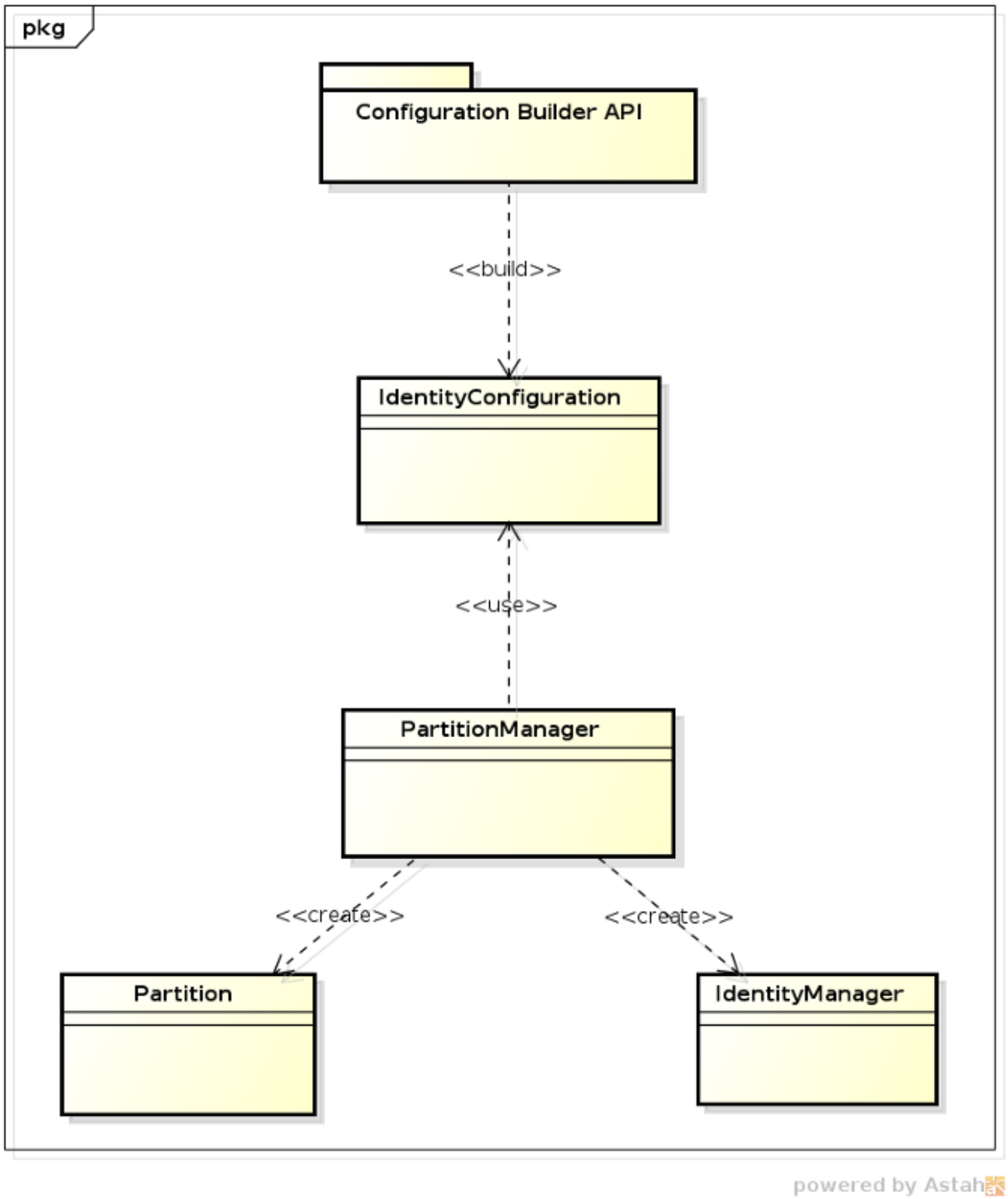
```
IdentityQuery<User> query = identityManager.<User> createIdentityQuery(User.class);

  query.setParameter(IdentityType.QUERY_ATTRIBUTE.byName("SomeAttribute"), "SomeAttributeValue");

  List<User> result = query.getResultList();

  for (User user : result) {
    // do something
  }
```

Chapter 7. Identity Management - Configuration

7.1. Configuration

7.1.1. Architectural Overview

Configuration in PicketLink is in essence quite simple; an `IdentityConfiguration` object must first be created to hold the PicketLink configuration options. Once all configuration options have been set, you just create a `PartitionManager` instance passing the previously created configuration. The `PartitionManager` can then be used to create `Partition` and `IdentityManager` instances.



The `IdentityConfiguration` is usually created using a Configuration Builder API, which provides a rich and fluent API for every single aspect of PicketLink configuration.

7.1.2. Default Configuration

If you'd like to get up and running with IDM quickly, the good news is that PicketLink will provide a default configuration that stores your identity data on the file system if no other configuration is available. This means that if you have the PicketLink libraries in your project, you can simply inject the `PartitionManager`, `IdentityManager` or `RelationshipManager` beans into your own application and start using them immediately:

```
@Inject PartitionManager partitionManager;
@Inject IdentityManager identityManager;
@Inject RelationshipManager relationshipManager;
```

> **Note**
>
> The default configuration is very useful for developing and testing purposes, as you don't need a database or a LDAP server to start managing your identity data.

7.1.3. Providing a Custom Configuration

In certain cases the default configuration may not be enough to your application. You can easily provide your own configuration by observing a specific `IdentityConfigurationEvent`:

```
@ApplicationScoped
public static class PicketLinkConfiguration {

  public void observeIdentityConfigurationEvent(@Observes IdentityConfigurationEvent event) {
    IdentityConfigurationBuilder builder = event.getConfig();

    // use the builder to provide your own configuration
  }
}
```

You can also provide your own configuration by producing one or more `IdentityConfiguration` instances using a `@Producer` annotated method:

```
@ApplicationScoped
public static class PicketLinkConfiguration {

  @Produces
  public IdentityConfiguration produceJPAConfiguration() {
      IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

      builder
          .named("jpa.config")
              .stores()
                  .jpa()
```

```
                    // configure the JPA store

        return builder.build();
    }

    @Produces
    public IdentityConfiguration produceLDAPConfiguration() {
        IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

        builder
            .named("ldap.config")
                .stores()
                    .ldap()
                        // configure the LDAP store

        return builder.build();
    }
}
```

The example above produces two distinct configurations: one using a JPA store and another using the LDAP store. During the startup PicketLink will resolve both configurations and initialize the IDM subsystem with them. You can also provide a single configuration.

For last, you can also build your own `PartitionManager` instance if you want more control.

```
@ApplicationScoped
public static class PicketLinkConfiguration {

    @PicketLink
    @Produces
    public PartitionManager producePartitionManager() {
        IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

        builder
            .named("produced.partition.manager.config")
                .stores()
                    .jpa()
                        .mappedEntity(
                                AccountTypeEntity.class,
                                RoleTypeEntity.class,
                                GroupTypeEntity.class,
                                IdentityTypeEntity.class,
                                RelationshipTypeEntity.class,
                                RelationshipIdentityTypeEntity.class,
                                PartitionTypeEntity.class,
                                PasswordCredentialTypeEntity.class,
                                DigestCredentialTypeEntity.class,
                                X509CredentialTypeEntity.class,
                                OTPCredentialTypeEntity.class,
                                AttributeTypeEntity.class
                        )
                        .addContextInitializer(this.contextInitializer)
                        .supportAllFeatures();

        return new DefaultPartitionManager(builder.build());
    }
```

```
}
```

The example above allows you to produce your own `PartitionManager` instance. Note that the producer method is annotated with the `PicketLink` annotation. Another important thing when producing your own `PartitionManager` is that you must manually create the partitions before start producing `IdentityManager` instances (eg.: the default partition).

7.1.4. Initializing the `PartitionManager`

You may need to initialize the `PartitionManager` with some data before your application starts to produce partition manager instances. PicketLink provides a specific event called `PartitionManagerCreateEvent`, which can used to provide any initialization logic right after a `PartitionManager` instance is created and before it is consumed by any injection point in your application.

```
public class MyPartitionManagerInitializer {

    public void init(@Observes PartitionManagerCreateEvent event) {
        // retrieve the recently created partition manager instance
        PartitionManager partitionManager = event.getPartitionManager();

        // retrieve all the configuration used to build the instance
        Collection configurations = partitionManager.getConfigurations();
    }
}
```

One important thing to keep in mind when providing a observer for `PartitionManagerCreateEvent` is that if any partition is created during the initialization, PicketLink won't try to create the default partition.

> **Note**
>
> Apache TomEE users should always provide an observer for `PartitionManagerCreateEvent` in order to initialize the partition manager properly. Specially if using the JPA store and if no active transaction exists when the `PartitionManager` is being created.

7.1.5. Programmatic Configuration Overview

The Identity Management configuration can be defined programmatically using the Configuration Builder API. The aim of this API is to make it easier to chain coding of configuration options in order to speed up the coding itself and make the configuration more *readable*.

Let's assume that you want to quick start with PicketLink Identity Management features using a File-based Identity Store. First, a fresh instance of `IdentityConfiguration` is created using

the `IdentityConfigurationBuilder` helper object, from where we choose which identity store we want to use (in this case a file-based store) and any other configuration option, if necessary. Finally, we use the configuration to create a `PartitionManager`, from where we can create `Partition` and `IdentityManager` instances:

```
IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

builder
  .named("default")
    .stores()
      .file()
        .supportAllFeatures();

DefaultPartitionManager partitionManager = new DefaultPartitionManager(builder.buildAll());

Realm defaultRealm = new Realm(Realm.DEFAULT_REALM);

// let's add the partition using the default configuration.
partitionManager.add(defaultRealm);

// if no partition is specified to createIdentityManager, defaults to the default Realm.
IdentityManager identityManager = partitionManager.createIdentityManager();

User john = new User("john");

// let's add john to the default partition
identityManager.add(user);
```

7.1.6. Providing Multiple Configurations

A `PartitionManager` can be built considering multiple configurations. This is a very powerful feature given that you can manage your identity data between different partitions each one using a specific configuration.

As discussed before, each configuration has a name. The name can be used to identify a configuration set as well to tell PicketLink the configuration that should be used to manage a specific `Partition`.

Let's take a more close look how you can use multiple configurations:

```
IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

builder
    .named("ldap.config")
        .stores()
            .ldap()
              // specific configuration options for the LDAP store
              .supportAllFeatures();
    .named("jpa.config")
        .stores()
            .jpa()
              // specific configuration options for the JPA store
```

```
                .supportAllFeatures();

DefaultPartitionManager partitionManager = new DefaultPartitionManager(builder.buildAll());

Realm internalPartition = new Realm("internal");

// the 'internal' partition will use the 'ldap.config' configuration
partitionManager.add(internalPartition, "ldap.config");

// we create an IdentityManager for the LDAP managed partition
IdentityManager internalIdentityManager = partitionManager.createIdentityManager(internalPartition);

User john = new User("john");

// john will be added to the LDAP
internalIdentityManager.add(john);

Realm externalPartition = new Realm("external");

// the 'external' partition will use the 'jpa.config' configuration
partitionManager.add(externalPartition, "jpa.config");

User mary = new User("mary");

// we create an IdentityManager for the JPA managed partition
IdentityManager externalIdentityManager = partitionManager.createIdentityManager(externalPartition);

// mary will be added to the database
externalIdentityManager.add(mary);
```

The example above is just one of the different things you can do with PicketLink. The code above defines two partitions: one for internal users and another one for external users. Each partition is associated with one of the provided configurations where the internal partition will use LDAP to store users whether the external partition will use JPA.

When you create a `IdentityManager` for one of those partitions, all identity management operations will be done considering the configuration associated with the current partition. In other words, considering the example above, the user 'john' will be stored in the LDAP and 'mary' in a Database.

7.1.7. Providing Multiple Stores for a Configuration

It is also possible to use multiple `IdentityStore` configurations in a single named configuration. This can be very useful when your identity data is distributed in different stores or even if a specific store have any kind of limitation that can be provided by another one.

For instance, the LDAP store have some limitations and does not support all features provided by PicketLink. One of those unsupported features is the ability to handle ad-hoc attributes. When using LDAP you're tied with a schema that usually is very hard to change in order to support all your needs.

In this cases, PicketLink allows you to combine in a single configuration the LDAP and the JPA store, for example. Where you can use LDAP for users, roles and groups and use the JPA store for relationships.

```
IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

builder
  .named("default")
    .stores()
      .jpa()
        // configuration options for the jpa store
        .supportGlobalRelationship(Relationship.class)
        .supportAttributes(true)
      .ldap()
        // configuration options for the ldap store
        .supportType(IdentityType.class)
```

The example above defines a single configuration with two stores: LDAP and JPA. For the LDAP store configuration we define that only `IdentityType` types should be supported. In other words, we're only storing users, roles and groups. For the JPA store configuration we define that only `Relationship` types should be supported. In other words, we're only storing relationships such as `Grant`, `GroupMembership`, etc.

You may also notice that the JPA store is configured to support attributes too. What means that we can now use ad-hoc attributes for all the supported types.

7.1.8. Configuring Credential Handlers

Each `IdentityStore` may support a different set of credential handlers. This documentations describes the built-in credential handlers provided by PicketLink, but sometimes you may want to provide your own implementations.

When you write your custom credential handler you need to tell PicketLink the identity store that will support it. This is done by the following code:

```
IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

builder
  .named("default")
    .stores()
      .jpa()
        // other JPA configuration
        .addCredentialHandler(UserPasswordCredentialHandler.class)
        .supportAllFeatures();
```

The example above shows how to configure a credential handler for a JPA-based store using the `addCredentialHandler` method.

7.1.8.1. Passing parameters to Credential Handlers

Some credential handlers support a set of configuration options to configure their behavior. These options can be specified as follows:

```
IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

builder
  .named("default")
    .stores()
      .jpa()
        // other JPA configuration
        .setCredentialHandlerProperty(PasswordCredentialHandler.PASSWORD_ENCODER, new BCryptPasswordEncoder(4))
        .supportAllFeatures();
```

The example above shows how to set a property for the `PasswordCredentialHandler` using the `setCredentialHandlerProperty` method.

7.1.9. Identity Context Configuration

The `IdentityContext` plays an important role in the PicketLink IDM achitecture. It is strongly used during the execution of operations. It carries very sensitive and contextual information for a specific operation and provides access for some of the IDM underlying services such as caching, event handling, id generator for `AttributedType` instances, among others.

Operations are always executed by a specific `IdentityStore` in order to persist or store identity data using a specific repository (eg.: LDAP, databases, filesystem, etc). When executing a operation the identity store must be able to:

- Access the current `Partition`. Eg.: `Realm` or `Tier`.
- Access the *Event Handling API* in order to fire events such as when an user is created, updated, etc.
- Access the *Caching API* in order to cache identity data and increase performance.
- Access to external resources, provided before the operation is executed and initialized by a `ContextInitializer`.

7.1.9.1. Initializing the IdentityContext

Sometimes you may need to provide additional configuration or even references for external resources before the operation is executed by an identity store. An example is how you tell to the `JPAIdentityStore` which `EntityManager` instance should be used. When executing an operation, the `JPAIdentityStore` must be able to access the current `EntityManager` to persist or retrieve data from the database. You need someway to populate the `IdentityContext` with such information. When you're configuring an identity store, there is a configuration option that allows you to provide a `ContextInitializer` implementation.

```
public interface ContextInitializer {
    void initContextForStore(IdentityContext context, IdentityStore<?> store);
}
```

The method `initContextForStore` will be invoked for every single operation and before its execution by the identity store. It can be implemented to provide all the necessary logic to initialize and populate the `IdentityContext` for a specific `IdentityStore`.

The configuration is also very simple:

```
IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

builder
  .named("default")
    .stores()
      .file()
        .supportAllFeatures();
        .addContextInitializer(new MyContextInitializer());
```

You can provide multiple initializers.

> **Note**
>
> Remember that initializers are executed for every single operation. Also, the same instance is used between operations which means your implementation should be “stateless”. You should be careful about the implementation in order to not impact performance, concurrency or introduce unexpected behaviors.

7.1.10. IDM configuration from XML file

Actually it's possible to configure IDM with XML configuration. This possibility is good especially in case when you want Picketlink IDM to be part of bigger system and your users won't have a possibility to change source code and so they can't configure it programmatically with the Builder API. So they will just need to change the configuration in XML file instead of doing some changes directly in source code.

7.1.10.1. Unified XML configuration

Whole Picketlink project provides unified format of configuration file, so that you can configure federation and IDM in same file.

```
<PicketLink xmlns="urn:picketlink:identity-federation:config:2.1">

  <PicketLinkIDP xmlns="urn:picketlink:identity-federation:config:1.0"
                 ServerEnvironment="tomcat" BindingType="POST" SupportsSignatures="true">
```

```
    <!-- SAML2 IDP configuration is here -->
  </PicketLinkIDP>

  <Handlers xmlns="urn:picketlink:identity-federation:handler:config:2.1">
     <!-- Configuration of SAML2 handlers is here -->
  </Handlers>

  <PicketLinkSTS xmlns="urn:picketlink:identity-federation:config:1.0"
                 STSName="Test STS" TokenTimeout="7200" EncryptToken="true">
    <!-- Configuration of Picketlink STS is here -->
  </PicketlinkSTS>

  <PicketLinkIDM>
    <!-- IDM configuration is here -->
  </PicketLinkIDM>

</PicketLink>
```

Note that if you don't want to use Picketlink Federation, you can omit it's configuration and use just IDM.

7.1.10.2. XML configuration format

XML configuration is leveraging Builder API and Java reflection during it's parsing, so names of XML elements are actually same like names of particular Builder methods.

For example, let's assume that you want to use `FileIdentityStore` and your programmatic configuration looks like this:

```
IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

builder
    .named(SIMPLE_FILE_STORE_CONFIG)
        .stores()
            .file()
            .preserveState(false)
            .supportGlobalRelationship(Relationship.class)
            .supportAllFeatures();
```

Same XML configuration to configure IDM with `FileIdentityStore` will look like this:

```
<PicketLink xmlns="urn:picketlink:identity-federation:config:2.1">

  <PicketLinkIDM>

    <named value="SIMPLE_FILE_STORE_CONFIG">
      <stores>
        <file>
          <preserveState value="false" />
          <supportGlobalRelationship value="org.picketlink.idm.model.Relationship" />
          <supportAllFeatures />
        </file>
```

```
      </stores>
    </named>

  </PicketLinkIDM>

</PicketLink>
```

You can take a look at testsuite [https://github.com/picketlink/picketlink/tree/master/modules/idm/tests/src/test/resources/config] to see more examples.

7.1.10.3. Bootstrap IDM from XML file

So to initialize Picketlink IDM from XML you can use the code like this:

```
// Replace with your own configuration file
String configFilePath = "config/embedded-file-config.xml";

ClassLoader tcl = Thread.currentThread().getContextClassLoader();
InputStream configStream = tcl.getResourceAsStream(configFilePath);
XMLConfigurationProvider xmlConfigurationProvider = new XMLConfigurationProvider();
IdentityConfigurationBuilder idmConfigBuilder =
  xmlConfigurationProvider.readIDMConfiguration(configStream);
```

Now you can bootstrap IDM from `idmConfigBuilder` in same way, like it's done in Programmatic Configuration. Note that you can initialize builder from XML file and then you can do some additional programmatic configuration. For example, you may need to programmatically add `JPAContextInitializer` in case that you are using JPA, because you will need access to JPA `EntityManager`.

Chapter 8. Identity Management - Working with JPA

8.1. JPAIdentityStoreConfiguration

The JPA identity store uses a relational database to store identity state. The configuration for this identity store provides control over which entity beans are used to store identity data, and how their fields should be used to store various identity-related state. The entity beans that store the identity data must be configured using the annotations found in the `org.picketlink.jpa.annotations` package. All identity configuration annotations listed in the tables below are from this package.

8.1.1. Default Database Schema

If you do not wish to provide your own JPA entities for storing IDM-related state, you may use the default schema provided by PicketLink in the `picketlink-idm-simple-schema` module. This module contains a collection of entity beans suitable for use with `JPAIdentityStore`. To use this module, add the following dependency to your Maven project's `pom.xml` file:

```
<dependency>
    <groupId>org.picketlink</groupId>
    <artifactId>picketlink-idm-simple-schema</artifactId>
    <version>${picketlink.version}</version>
</dependency>
```

In addition to including the above dependency, the default schema entity beans must be configured in your application's `persistence.xml` file. Add the following entries within the `persistence-unit` section:

```
<class>org.picketlink.idm.jpa.model.sample.simple.AttributedTypeEntity</class>
<class>org.picketlink.idm.jpa.model.sample.simple.AccountTypeEntity</class>
<class>org.picketlink.idm.jpa.model.sample.simple.RoleTypeEntity</class>
<class>org.picketlink.idm.jpa.model.sample.simple.GroupTypeEntity</class>
<class>org.picketlink.idm.jpa.model.sample.simple.IdentityTypeEntity</class>
<class>org.picketlink.idm.jpa.model.sample.simple.RelationshipTypeEntity</class>
<class>org.picketlink.idm.jpa.model.sample.simple.RelationshipIdentityTypeEntity</class>
<class>org.picketlink.idm.jpa.model.sample.simple.PartitionTypeEntity</class>
<class>org.picketlink.idm.jpa.model.sample.simple.PasswordCredentialTypeEntity</class>
<class>org.picketlink.idm.jpa.model.sample.simple.DigestCredentialTypeEntity</class>
<class>org.picketlink.idm.jpa.model.sample.simple.X509CredentialTypeEntity</class>
<class>org.picketlink.idm.jpa.model.sample.simple.OTPCredentialTypeEntity</class>
<class>org.picketlink.idm.jpa.model.sample.simple.AttributeTypeEntity</class>
```

yes

8.1.2. Configuring an EntityManager

Before the JPA identity store can be used, it must be provided with an `EntityManager` so that it can connect to a database. In Java EE this can be done by providing a producer method within your application that specifies the `@org.picketlink.annotations.PicketLink` qualifier, for example like so:

```
@Produces
@PicketLink
@PersistenceContext(unitName = "picketlink")
private EntityManager picketLinkEntityManager;
```

8.1.3. Mapping `IdentityType` Types

The following table summarizes all annotations that can be used to map entities to `IdentityType` types:

Table 8.1. `IdentityType` Annotations

Annotation	Description	Property Type	Required
`@IdentityManaged`	This annotation is a type-level annotation and must be used to specify the `IdentityType` types that should be mapped by the annotated entity.	-	True
`@AttributeValue`	This annotation can be used to map a entity property to a `IdentityType` property. The `name` property of this annotation can be used in case the property names are different.	Any Type	False
`@Identifier`	The unique identifier value for the identity.	String	True
`@IdentityClass`	The type for the identity. When a `IdentityType` is stored the FQN of	String	True

Mapping `IdentityType` Types

Annotation	Description	Property Type	Required
	its type is stored in a property annotated with this annotation.		
`@OwnerReference`	The reference to a `Partition` mapped entity. This annotation is used to identify the property that holds a reference to the partition where a `IdentityType` belongs. Usually this annotation is used in conjunction with a @ManyToOne property referencing the entity used to store partitions.	The same type used to map a `Partition`	True

The following code shows an example of an entity class configured to store `User` types:

Example 8.1. Example

```
@IdentityManaged (User.class)
@Entity
public class IdentityTypeEntity implements Serializable {

  @Id
  @Identifier
  private String id;

  @IdentityClass
  private String typeName;

  @AttributeValue
  private String loginName;

  @AttributeValue
  private Date createdDate;

  @AttributeValue
  private Date expirationDate;

  @AttributeValue
  private boolean enabled;

  @OwnerReference
  @ManyToOne
  private PartitionTypeEntity partition;
```

```
    // getters and setters
}
```

8.1.4. Mapping Partition Types

The following table summarizes all annotations that can be used to map entities to IdentityType types:

Table 8.2. Partition Annotations

Annotation	Description	Property Type	Required
@IdentityManaged	This annotation is a type-level annotation and must be used to specify the Partition types that should be mapped by the annotated entity.	-	True
@AttributeValue	This annotation can be used to map a entity property to a Partition property. The name property of this annotation can be used in case the property names are different.	Any Type	False
@Identifier	The unique identifier value for the partition.	String	True
@PartitionClass	The type for the partition. When a Partition is stored the FQN of its type is stored in a property annotated with this annotation.	String	True
@ConfigurationName	This annotation must be used to indicate the field to store the configuration name for a partition.	String	True

The following code shows an example of an entity class configured to store Realm types:

Example 8.2. Example

```
@IdentityManaged (Realm.class)
@Entity
public class PartitionTypeEntity implements Serializable {

  @Id
  @Identifier
  private String id;

  @AttributeValue
  private String name;

  @PartitionClass
  private String typeName;

  @ConfigurationName
  private String configurationName;

    // getters and setters
}
```

8.1.5. Mapping `Relationship` Types

The following table summarizes all annotations that can be used to map entities to `Relationship` types:

Table 8.3. `Relationship` Annotations

Annotation	Description	Property Type	Required
`@IdentityManaged`	This annotation is a type-level annotation and must be used to specify the `Relationship` types that should be mapped by the annotated entity.	-	True
`@AttributeValue`	This annotation can be used to map a entity property to a `Relationship` property. The `name` property of this annotation can be used in case the property names are different.	Any Type	False

Mapping `Relationship` Types

Annotation	Description	Property Type	Required
`@Identifier`	The unique identifier value for the relationship.	String	True
`@RelationshipClass`	The type for the relationship. When a `Relationship` is stored the FQN of its type is stored in a property annotated with this annotation.	String	True
`@RelationshipDescriptor`	This annotation must be used to indicate the field to store the name of the relationship role of a member.	String	True
`@RelationshipMember`	The reference to a `IdentityType` mapped entity. This annotation is used to identify the property that holds a reference to the identity type that belongs to this relationship with a specific descriptor. Usually this annotation is used in conjunction with a @ManyToOne property referencing the entity used to store identity types.	The same type used to map a `IdentityType`	True
`@OwnerReference`	The reference to a `Relationship` mapped entity. This annotation is used to identify the property that holds a reference to the root entity for relationships, usually the entity	The same type used to map an entity with the `@RelationshipClass` annotation.	True

Mapping `Relationship` Types

Annotation	Description	Property Type	Required
	annotated with the `@RelationshipClass` annotation.		

The following code shows an example of an entity class configured to store `Relationship` types:

Example 8.3. Example

```
@IdentityManaged (Relationship.class)
@Entity
public class RelationshipTypeEntity implements Serializable {

  @Id
  @Identifier
  private String id;

  @RelationshipClass
  private String typeName;

    // getters and setters
}
```

When mapping a relationship you also need to provide a specific entity to store its members:

Example 8.4. Example

```
@Entity
public class RelationshipIdentityTypeEntity implements Serializable {

  @Id
  @GeneratedValue
  private Long id;

  @RelationshipDescriptor
  private String descriptor;

  @RelationshipMember
  @ManyToOne
  private IdentityTypeEntity identityType;

  @OwnerReference
  @ManyToOne
  private RelationshipTypeEntity owner;

    // getters and setters
}
```

8.1.6. Mapping Attributes for `AttributedType` Types

The following table summarizes all annotations that can be used to map attributes to `AttributedType` types:

Table 8.4. `Partition` Annotations

Annotation	Description	Property Type	Required
`@AttributeName`	The name of the attribute. A property with this annotation is used to store the name of the attribute.	String	True
`@AttributeValue`	The value of the attribute. A property with this annotation is used to store the value of the attribute. Values are Base64 encoded.	String	True
`@AttributeClass`	The type for the attribute. When a attribute is stored the FQN of its type is stored in a property annotated with this annotation.	String	True
`@OwnerReference`	The reference to a `IdentityType`, or a `Partition` or a `Relationship` mapped entity. This annotation is used to identify the property that holds a reference to the owner of the attributes.	The same type used to map `IdentityType`, or a `Partition` or a `Relationship` type.	True

The following code shows an example of an entity class configured to store attributes for `IdentityType` types:

Example 8.5. Example

```
@Entity
public class IdentityTypeAttributeEntity implements Serializable {

  @OwnerReference
  @ManyToOne
  private IdentityTypeEntity owner;

  @AttributeClass
  private String typeName;

  @AttributeName
  private String name;

  @AttributeValue
  private String value;

    // getters and setters
}
```

8.1.7. Mapping a `CredentialStorage` type

The following table summarizes all annotations that can be used to map attributes to `AttributedType` types:

Table 8.5. `Partition` Annotations

Annotation	Description	Property Type	Required
`@CredentialClass`	The type for the credential. When a credential is stored the FQN of its corresponding `CredentialStorage` type is stored in a property annotated with this annotation.	String	True
`@CredentialProperty`	This annotation can be used to map a entity property to a `CredentialStorage` property. The `name` property of this annotation can be used in case the property names are different.	String	True
`@EffectiveDate`	The effective date for a credential. A	String	True

Mapping a `CredentialStorage` type

Annotation	Description	Property Type	Required
	property annotated with this annotation will be mapped to the `effectiveDate` of a `CredentialStorage` type.		
`@ExpiryDate`	The expiry date for a credential. A property annotated with this annotation will be mapped to the `expiryDate` of a `CredentialStorage` type.	String	True
`@OwnerReference`	The reference to a `IdentityType` mapped entity. This annotation is used to identify the property that holds a reference to the owner of the credential.	The same type used to map a `IdentityType` type.	True

The following code shows an example of an entity class configured to store password-based credentials for `IdentityType` types:

Example 8.6. Example

```
@ManagedCredential (EncodedPasswordStorage.class)
@Entity
public class PasswordCredentialTypeEntity implements Serializable {

  @Id
  @GeneratedValue
  private Long id;

  @OwnerReference
  @ManyToOne
  private IdentityTypeEntity owner;

  @CredentialClass
  private String typeName;

  @EffectiveDate
  private Date effectiveDate;

  @ExpiryDate
```

```
  private Date expiryDate;

  @CredentialProperty (name = "encodedHash")
  private String passwordEncodedHash;

  @CredentialProperty (name = "salt")
  private String passwordSalt;

    // getters and setters
}
```

8.1.8. Configuring the Mapped Entities

Once your entities are properly mapped, you're ready to configure the JPA store with them. To do that you only need to:

```
IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

builder
  .stores()
    .jpa()
      .mappedEntity(IdentityTypeEntity.class, PartitionTypeEntity.class, ...);
```

8.1.9. Providing a `EntityManager`

Sometimes you may need to configure how the `EntityManager` is provided to the `JPAIdentityStore`, like when your application is using CDI and you must run the operations in the scope of the current transaction by using a injected `EntityManager` instance.

In cases like that, you need to initialize the `IdentityContext` by providing a `ContextInitializer` implementation, as discussed in Identity Context Configuration. You can always provide your own implementation for this interface to obtain the `EntityManager` from your application's environment.

```
IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

builder
  .stores()
    .jpa()
      .addContextInitializer(new ContextInitializer() {
        @Override
        public void initContextForStore(IdentityContext context, IdentityStore<?> store) {
          if (store instanceof JPAIdentityStore) {
              EntityManager entityManager = // get the EntityManager
           context.setParameter(JPAIdentityStore.INVOCATION_CTX_ENTITY_MANAGER, entityManager);
          }
        }
      });
```

Providing a `EntityManager`

In most cases you don't need to provide your own `ContextInitializer` but use an implementation provided by PicketLink:

```
@Inject
private EEJPAContextInitializer contextInitializer;

public void observeIdentityConfigurationEvent(@Observes IdentityConfigurationEvent event) {
  IdentityConfigurationBuilder builder = event.getConfig();

  builder
    .stores()
      .jpa()
        // more JPA store config
        .addContextInitializer(this.contextInitializer);
}
```

Chapter 9. Identity Management - Working with LDAP

9.1. Overview

The LDAP Identity Store allows a LDAP Directory to be used as a source of identity data. Most organizations rely on a LDAP Directory to store users, groups, roles and relationships between those entities. Some of them only store users and groups, others only users and so forth. The point is that each organization has its own structure, how data is organized on the server and policies to govern all that. That said, is very hard to get all different use cases satisfied given all those nuances.

To try to overcome that, the LDAP Identity Store provides a simple and easy mapping between the entries in your LDAP tree and the PicketLink types (`IdentityType`, `Relationship` and so forth), plus some additional configuration options that give you more control how the store should integrate with your server.

The store can be used in read-only or read-write mode. Depending on your permissions on the server, you should consider one of these alternatives, otherwise you can get errors when, for example, trying to add, update or remove entries from the server.

The list below summarizes some of the most important capabilities provided by this store:

- Mapping `IdentityType` types to their corresponding LDAP entries and attributes.
- Mapping `Relationship` types to their corresponding LDAP entries and attributes.
- Mapping of parent/child relationships between the LDAP entries mapped to the same type.
- Authentication of users based on username/password credentials.
- Use of LDAP UUID attributes as the identifier for identity types. For each identity type in PicketLink we need to provide a single/unique identifier. The LDAP store uses the `entryUUID` and `objectGUID` (depending on your server implementation, of course) to identify each type.

But the LDAP Directory has also some limitations (schema limitations, restrictive usage policies) and because of that the LDAP Identity Store does not supports all the feature set provided by PicketLink. The table below lists what is not supported by the LDAP Identity Store:

- Chapter 6, *Identity Management - Attribute Management*
- Complex relationship mappings such as `GroupRole`.
- Relationships can not be updated directly using the `IdentityManager`.
- Limited support for credential types. Only username/password is available.

9.2. Configuration

The LDAP Identity Store can be configured as follows:

```
IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

builder
    .named("ldap.config")
        .stores()
            .ldap()
                // connection configuration
                .baseDN("dc=jboss,dc=org")
                .bindDN("uid=admin,ou=system")
                .bindCredential("passwd")
                .url("ldap://localhost:389")

                // mapping configuration
                .mapping(Agent.class)
                    .baseDN("ou=Agent,dc=jboss,dc=org")
                    .objectClasses("account")
                    .attribute("loginName", "uid", true)
                    .readOnlyAttribute("createdDate", "createTimeStamp")
                .mapping(User.class)
                    .baseDN("ou=User,dc=jboss,dc=org")
                    .objectClasses("inetOrgPerson", "organizationalPerson")
                    .attribute("loginName", "uid", true)
                    .attribute("firstName", "cn")
                    .attribute("lastName", "sn")
                    .attribute("email", EMAIL)
                    .readOnlyAttribute("createdDate", "createTimeStamp")
                .mapping(Role.class)
                    .baseDN("ou=Roles,dc=jboss,dc=org")
                    .objectClasses("role")
                    .attribute("name", "cn", true)
                    .readOnlyAttribute("createdDate", "createTimeStamp")
                .mapping(Group.class)
                  .hierarchySearchDepth(4)
                  .objectClasses("group")
                  .attribute("name", "cn", true)
                  .readOnlyAttribute("createdDate", "createTimeStamp")
                  .parentMembershipAttributeName("member")
                .mapping(Grant.class)
                    .forMapping(Role.class)
                    .attribute("assignee", "member")
                .mapping(GroupMembership.class)
                    .forMapping(Group.class)
                    .attribute("member", "member");
```

9.2.1. Connecting to the LDAP Server

The connection to your LDAP server can be configured as follows:

```
.ldap()
  .baseDN("dc=jboss,dc=org")
```

```
  .bindDN("uid=admin,ou=system")
  .bindCredential("passwd")
  .url("ldap://localhost:389")
```

You can also provide additional connection `Properties` that will be used when creating the `LdapContext`.

```
.ldap()
  .connectionProperties(myProperties)
```

The table below describes each configuration option:

Table 9.1. LDAP Connection Configuration Options

Option	Description
`baseDN`	Sets the base DN for a specific mapped type or all types.
`bindDN`	Sets the the DN used to bind against the ldap server. If you want to perform write operations the DN must have permissions on the agent,user,role and group contexts.
`bindCredential`	Sets the password for the bindDN.
`url`	Sets the url that should be used to connect to the server. Eg.: ldap://<<server>>:389 .
`connectionProperties`	Set a `Properties` instance from where additional connection properties will be retrieved from when creating the `LdapContext`.

9.2.2. Mapping Identity Types

The LDAP configuration provides a simple mapping between your identity types and their corresponding LDAP entries. The way you map your types have a huge impact on how the LDAP Identity Store performs its operations.

Usually, a mapping is done as follows:

```
IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

builder
    .named("ldap.config")
        .stores()
            .ldap()
                .mapping(User.class)
                    .baseDN("ou=User,dc=jboss,dc=org")
```

```
                    .objectClasses("inetOrgPerson", "organizationalPerson")
                    .attribute("loginName", "uid", true)
                    .attribute("firstName", "cn")
                    .attribute("lastName", "sn")
                    .attribute("email", "mail")
                    .readOnlyAttribute("createdDate", "createTimeStamp")
```

For each mapping you need to provide the identity type being mapped (in the case above the `User` type) plus all information required to store the type and populate its properties from their corresponding LDAP attributes.

In the example above, we're considering that `User` entries are located at the *baseDN* "ou=User,dc=jboss,dc=org". The baseDN is a very important information, specially if you want to store information from a type instance. Beside that, the baseDN can have a huge impact on performance when querying your LDAP entries for a specific type, as the search will be more restrictive and consider only those entries located at the baseDN and sub entries.

Another important configuration is the *objectClass* list related with a type. The objectClass is very important when storing new entries in your LDAP server. Also, the objectClass helps the LDAP Identity Store to make better queries against your server by restricting which entries should be considered during the search based on the objectClass list you provide.

In order to store and retrieve attributes from the LDAP server, you need to map them to the properties of your type. The attribute mapping is pretty simple, you just provide the name of the property being mapped and its corresponding LDAP attribute name. An important aspect when mapping the attributes is that you should always configure an attribute as the identifier. In the example above, we're telling the LDAP configuration to consider the following attribute as an identifier:

```
.mapping(User.class)
  .attribute("loginName", "uid", true)
```

9.2.3. Mapping Relationship Types

As mentioned before, the relationship support of the LDAP Identity Store is limited. But you can always map the most common relationships such as `Grant` and `GroupMembership`

```
.ldap()
  .mapping(Grant.class)
    .forMapping(Role.class)
      .attribute("assignee", "member"))
```

When mapping a relationship type you need to configure which identity type is the owner of a relationship. For example, when mapping a `Grant` relationship, the LDAP attribute used to map the association between a role and other types is the *member* attribute. This attribute belongs to

role entries on the LDAP server, what makes the `Role` type the owner of this relationship. For last, we need to tell which property on the `Grant` type is related with the associated entries. In the case of the `Grant` relationship, we're configuring the *assignee* property to store the associated type instances.

9.2.4. Mapping a Type Hierarchies

The LDAP configuration supports the mapping of simple hierarchies (parent/child) of a single type. This is specially useful when mapping groups, for example. Where groups can have a parent and also child groups.

```
.ldap()
  .mapping(Group.class)
    .parentMembershipAttributeName("member")
```

In the example above, we're using the *member* attribute from LDAP to store the childs of a parent group.

In some cases, the performance can be impacted when retrieving parent/child hierarchies from the LDAP server. By default, the LDAP Identity Store is configure to resolve only three levels of hierarchies. But you can always override this configuration as follows:

```
.ldap()
  .mapping(Group.class)
    .hierarchySearchDepth(1)
```

In the example above, we're telling the LDAP Identity Store to consider only one level depth. Which means that only the direct parent of a group will be resolved.

9.2.5. Mapping Groups to different contexts

Sometimes may be useful to map a specific group to a specific context or DN.

The following configuration maps the group with path */QA Group* to *ou=QA,ou=Groups,dc=jboss,dc=org*

```
mapping(Group.class)
  .baseDN(embeddedServer.getGroupDnSuffix())
  .objectClasses(GROUP_OF_NAMES)
  .attribute("name", CN, true)
  .readOnlyAttribute("createdDate", CREATE_TIMESTAMP)
  .parentMembershipAttributeName("member")
  .parentMapping("QA Group", "ou=QA,ou=Groups,dc=jboss,dc=org")
```

With this configuration you can have groups with the same name, but with different paths.

Mapping
Groups
to

```
IdentityManager identityManager = getIdentityManager();
Group managers = new SimpleGroup("managers");

identityManager.add(managers); // group's path is /manager

Group qaGroup = identityManager.getGroup("QA Group");
Group managersQA = new SimpleGroup("managers", qaGroup);

// the QA Group is mapped to a different DN.
Group qaManagerGroup = identityManager.add(managersQA); // group's path is /QA Group/managers
```

Chapter 10. PicketLink Subsystem

10.1. Overview

The PicketLink Subsystem extends JBoss Application Server to introduce some new capabilities, providing a infrastructure to deploy and manage PicketLink deployments and services. Currently, only JBoss Enterprise Application Platform 6.1 is supported.

In a nutshell, the most important capabilities are:

- A rich domain model supporting the configuration of PicketLink Federation (specially SAML-based applications) deployments and Identity Management services.
- Minimal configuration for deployments. Part of the configuration is done automatically with some hooks for customizations.
- Minimal dependencies for deployments. All PicketLink dependencies are automatically set from modules.
- Configuration management using JBoss Application Server Management API. It can be managed in different ways: HTTP/JSON, CLI, Native DMR, etc.
- Identity Management Services are exposed in JNDI and are fully integrated with CDI. You can use PicketLink Identity Management without requiring the base module dependencies.
- Applications don't need to change when moving between different environments such as development, testing, staging or production. All the configuration is defined outside the application.
- Users need to learn a single and consolidated schema.

> **Important**
>
> The subsystem is not available yet in JBoss Enterprise Application Platform 6. While it is not updated with the PicketLink modules and subsystem you must follow the instructions on the next sections to get it up and running.

10.2. Installation and Configuration

To get the PicketLink subsystem properly installed you only need to use the PicketLink Installer.

> **Note**
>
> This step may no longer be required once the subsystem is available in a future version of JBoss Enterprise Application Platform.

Once the subsystem is properly installed you need to change your `standalone/domain.xml`, inside your EAP installation, with the following extension and subsystem:

```
<extensions>
  ...

  <!-- Add the PicketLink extension -->
  <extension module="org.picketlink.as.extension" />

</extensions>
<profile>

  <!-- Add the PicketLink Subsystem -->
  <subsystem xmlns="urn:jboss:domain:picketlink:1.0" />

  ...
</profile>
```

10.3. Configuring the PicketLink Dependencies for your Deployment

One your JBoss Application Server is properly configured with all PicketLink libraries and their respective modules, you can add a `META-INF/jboss-deployment-structure.xml` file inside the root directory of your deployment to configure the dependencies as follows:

```
<jboss-deployment-structure>
  <deployment>
      <dependencies>
          <!-- This will enable PicketLink Authentication/Authorization and IDM dependencies
 to your deployment. -->
      <module name="org.picketlink.core" />
        </dependencies>
  </deployment>
</jboss-deployment-structure>
```

```
<jboss-deployment-structure>
  <deployment>
      <dependencies>
        <!-- This will enable only the IDM dependencies to your deployment. -->
      <module name="org.picketlink.idm" />
        </dependencies>
  </deployment>
</jboss-deployment-structure>
```

It is strongly recommended that you use the PicketLink libraries from your JBoss Application Server modules. When using this way, you don't need to add any additional library to your

deployments and you can easily manage the PicketLink libraries without requiring changes to your deployments.

Considering that you no longer need the PicketLink libraries inside your deployment, you must change your Maven dependencies to use the PicketLink dependencies with scope `provided`:

```
<dependency>
  <groupId>org.picketlink</groupId>
  <artifactId>picketlink-api</artifactId>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>org.picketlink</groupId>
  <artifactId>picketlink-idm-api</artifactId>
  <scope>provided</scope>
</dependency>
```

10.4. Domain Model

The subsystem provides a domain model that allows you to configure the PicketLink Federation and Identity Management services using the `standalone/domain.xml` inside your EAP installation. The domain model is very easy to understand if you are already familiar with the PicketLink configuration.

```
<subsystem xmlns="urn:jboss:domain:picketlink:1.0">
  <!-- An example of the PicketLink Federation configuration -->
  <federation alias="federation-with-signatures">
    <saml token-timeout="4000" clock-skew="0"/>
   <key-store url="/jbid_test_keystore.jks" passwd="changeit" sign-key-alias="localhost" sign-
key-passwd="changeit"/>
       <identity-provider  url="http://localhost:8080/idp-sig/"  alias="idp-sig.war"  security-
domain="idp" supportsSignatures="true" strict-post-binding="false">
      <trust>
        <trust-domain name="localhost" cert-alias="localhost"/>
        <trust-domain name="127.0.0.1" cert-alias="localhost"/>
      </trust>
    </identity-provider>
    <service-providers>
            <service-provider  alias="sales-post-sig.war"  security-domain="sp"  url="http://
localhost:8080/sales-post-sig/" post-binding="true" supportsSignatures="true"/>
          <service-provider  alias="sales-redirect-sig.war"  security-domain="sp"  url="http://
localhost:8080/sales-redirect-sig/" post-binding="false" supportsSignatures="true" strict-post-
binding="false"/>
    </service-providers>
  </federation>

  <!-- An example of the PicketLink Identity Management configuration -->
                                <identity-management                jndi-name="picketlink/
JPAPartitionManager" alias="jpa.partition.manager">
    <identity-configuration name="jpa.store.config">
      <jpa-store data-source="jboss/datasources/ExampleDS">
```

```
        <supportedTypes supportsAll="true"/>
      </jpa-store>
    </identity-configuration>
  </identity-management>
</subsystem>
```

> **Note**
>
> The domain model XML schema can be obtained from https://github.com/picketlink/picketlink-as-subsystem/blob/master/src/main/resources/schema/jboss-picketlink_1_0.xsd.

10.5. Identity Management

The subsystem provides a domain model that allows you to configure PicketLink Identity Management Services using the `standalone/domain.xml`. Basically, what the subsystem does is parse the configuration, automatically build a org.picketlink.idm.PartitionManager and expose it via JNDI for further access.

With the subsystem you can :

- Externalize and centralize the IDM configuration for deployments.
- Define multiple configuration for identity management services.
- Expose the `PartitionManager` via JNDI for further access.
- If using CDI, inject the `PartitionManager` instances using the `Resource` annotation.
- If using CDI, use the PicketLink IDM alone without requiring the base module dependencies. In this case you can provide your own configuration without using the subsystem's domain model.

The IDM domain model is an abstraction for all PicketLink IDM configuration, providing a single schema from which all configurations can be defined. If you're already familiar with the Configuration API, you'll find the domain pretty simple and intuitive.

```
<subsystem xmlns="urn:jboss:domain:picketlink:1.0">
                                <identity-management                jndi-name="picketlink/
FileBasedPartitionManager" alias="file.based.partition.manager">
    <identity-configuration name="file.config">
            <file-store  working-dir="/tmp/pl-idm-complete"  always-create-files="true"  async-
write="true"
                  async-write-thread-pool="10">
        <supportedTypes supportsAll="true"/>
      </file-store>
    </identity-configuration>
  </identity-management>
```

```
                                <identity-management                jndi-name="picketlink/
JPADSBasedPartitionManager" alias="jpa.ds.based.partition.manager">
    <identity-configuration name="jpa.config">
      <jpa-store data-source="jboss/datasources/ExampleDS">
        <supportedTypes supportsAll="true"/>
      </jpa-store>
    </identity-configuration>
  </identity-management>
</subsystem>
```

> **Note**
>
> If you are looking for more examples about how to use the domain model, take a look at https://github.com/picketlink/picketlink-as-subsystem/blob/master/src/test/resources/picketlink-subsystem.xml.

Most of the configuration are known if you are familiar with the PicketLink IDM configuration. But the domain model provides some additional configuration in order to allow deployments to access the configured identity management services. Basically, each configuration must have a:

- **jndi-url**, that defines where the `PartitionManager` should be published in the JNDI tree for further access.
- **alias**, an alias for the configuration to allow other subsystems to inject the Identity Management Services using the MSC injection infrastructure.

The rest of the configuration is very similar with how you use the Configuration API to programmaticaly build the IDM configuration. For a complete description of the domain model elements, please take a look at the XML Schema.

10.5.1. `JPAIdentityStore`

In order to provide a better and easy integration with the container, the `JPAIdentityStore` configuration provides some additional configuration to let you configure how the `EntityManagerFactory` is built or used by the `JPAIdentityStore`.

10.5.1.1. Using a DataSource JNDI Url

When you specify a DataSource JNDI url, the subsystem will automatically build a `EntityManagerFactory` using a default configuration. This is the fast way to get a JPA Identity Store configuration.

The DataSource JNDI url can be specified using the **data-source** attribute as follows:

```
<subsystem xmlns="urn:jboss:domain:picketlink:1.0">
                                <identity-management                jndi-name="picketlink/
JPADSBasedPartitionManager" alias="jpa.ds.based.partition.manager">
```

```
    <identity-configuration name="jpa.config">
      <jpa-store data-source="jboss/datasources/ExampleDS">
        <supportedTypes supportsAll="true"/>
      </jpa-store>
    </identity-configuration>
  </identity-management>
</subsystem>
```

This configuration option is very handy if you want to use the basic IDM model provided by PicketLink.

10.5.1.2. Using a `EntityManagerFactory` JNDI Url

Sometimes you may need more control over the JPA Persistence Unit configuration. In this case you can use the **entity-manager-factory** attribute to specify where your previously built `EntityManagerFactory` is located.

```
<subsystem xmlns="urn:jboss:domain:picketlink:1.0">
                                <identity-management                jndi-name="picketlink/
JPAEMFBasedPartitionManager" alias="jpa.emf.based.partition.manager">
    <identity-configuration name="jpa.config">
      <jpa-store entity-manager-factory="jboss/PicketLinkEMF">
        <supportedTypes>
          <supportedType class="org.picketlink.idm.model.Partition"/>
          <supportedType class="org.picketlink.idm.model.IdentityType"/>
          <supportedType class="org.picketlink.idm.model.Relationship"/>
        </supportedTypes>
      </jpa-store>
    </identity-configuration>
  </identity-management>
</subsystem>
```

This configuration option is very useful if you want to support custom types.

10.5.1.3. Using a JBoss Module

The JPA Identity Store configuration allows you to specify a JBoss Module from where the JPA Persistence Unit and mapped entities will be loaded from.

This configuration can be done using two attributes:

- **entity-module**, the module name where the JPA Persistence Unit and all mapped entities are located.
- **entity-module-unit-name**, the name of the JPA Persistence Unit name. If you don't provide a name the subsystem will use **identity**.

```
<subsystem xmlns="urn:jboss:domain:picketlink:1.0">
                                    <identity-management          jndi-name="picketlink/
JPACustomEntityBasedPartitionManager" alias="jpa.custom.entity.based.partition.manager">
```

```
      <identity-configuration name="jpa.config">
              <jpa-store  entity-module="org.picketlink.test"  entity-module-unit-name="custom-
pu" module="org.picketlink.test">
          <supportedTypes>
            <supportedType class="org.picketlink.idm.model.Partition"/>
            <supportedType class="org.picketlink.idm.model.IdentityType"/>
            <supportedType class="org.picketlink.idm.model.Relationship"/>
          </supportedTypes>
          <credential-handlers>
                                                                                <credential-
handler class="test.org.picketlink.as.subsystem.module.idm.SaleAgentPasswordCredentialHandler"/
>
          </credential-handlers>
        </jpa-store>
      </identity-configuration>
    </identity-management>
  </subsystem>
```

10.5.2. Usage Examples

If you want to have your deployment properly configured with the PicketLink Identity Management Services, you should add a `META-INF/jboss-deployment-structure.xml` file to your deployment as follows:

```
<jboss-deployment-structure>
  <deployment>
    <dependencies>
      <module name="org.picketlink.idm" />

        <!-- We only need this dependency because we want to use the built-in schema -->
        <module name="org.picketlink.idm.schema" />
    </dependencies>
  </deployment>
  </jboss-deployment-structure>
```

> **Note**
>
> When you're configuring the PicketLink dependencies using the `META-INF/jboss-deployment-structure.xml` file you don't need to ship the libraries inside your deployment. All the necessary dependencies are automatically resolved and configured.

10.5.2.1. Injecting a `PartitionManager` using `Resource` annotation

```
@Resource(mappedName="picketlink/JPADSBasedPartitionManager")
  private PartitionManager jpaDSBasedPartitionManager;
```

10.5.2.2. Producing your own configuration

```
@ApplicationScoped
  public static class MyIdentityConfigurationProducer {

  @Produces
  public IdentityConfiguration produceIdentityConfiguration() {

  return new IdentityConfigurationBuilder().named("default").stores().file().supportAllFeatures().build();
  }

  }
```

10.6. Federation

All the configuration is external from applications where there is no need to add or change configuration files inside the application being deployed. The subsystem is responsible for during deployment time properly configure the applications being deployed, according with the configurations defined in the domain model:

- The configurations in `picketlink.xml` are automatically created. No need to have this file inside your deployment.
- The PicketLink Authenticators (Apache Tomcat Valves) for Identity Providers and Service Providers are automatically registered. No need to have a jboss-web.xml file inside your deployment.
- The PicketLink dependencies are automatically configured. No need to have a `META-INF/jboss-deployment-structure.xml` inside your deployment defining the `org.picketlink` module as a dependency.
- The Security Domain is automatically configured using the configurations defined in the domain model. No need to have a `WEB-INF/jboss-web.xml` file inside your deployment.

The table bellow summarizes the main differences between the traditional configuration and the subsystem configuration for PicketLink applications:

Configuration	Old Configuration	Subsystem Configuration
WEB-INF/picketlink.xml	Required	Not required. If present it will be considered instead of the configurations defined in the domain model.
WEB-INF/jboss-web.xml	Required	Not required. The PicketLink Authenticators (Tomcat Valves) and the Security Domain is read from the domain model.

Configuration	Old Configuration	Subsystem Configuration
META-INF/jboss-deployment-structure.xml	Required	Not required. When the PicketLink Extension/ Subsystem is enabled, the dependency to the org.picketlink module is automatically configured.

10.6.1. The Federation concept (Circle of Trust)

When using the PicketLink subsystem to configure and deploy your identity providers and service providers, all of them are grouped in a Federation.

A Federation can be understood as a Circle of Trust (CoT) from which applications share common configurations (certificates, saml specific configurations, etc) and where each participating domain is trusted to accurately document the processes used to identify a user, the type of authentication system used, and any policies associated with the resulting authentication credentials.

Each federation has one Identity Provider and many Service Providers. You do not need to specify for each SP the IDP that it trusts, because this is defined by the federation.

10.6.2. Federation Domain Model

The domain model is an abstraction for all PicketLink Federation configuration, providing a single schema from which all configurations can be defined for Identity Providers or Service Providers, for example.

The example bellow shows how the domain model can used to configure an Identity Provider and a Service Provider.

```
<subsystem xmlns="urn:jboss:domain:picketlink:1.0">
  <federation alias="federation-without-signatures">

    <saml token-timeout="4000" clock-skew="0" />

                                <identity-provider        alias="idp.war"        security-
domain="idp" supportsSignatures="false" url="http://localhost:8080/idp/">
        <trust>
            <trust-domain name="localhost" />
            <trust-domain name="mycompany.com2" />
            <trust-domain name="mycompany.com3" />
            <trust-domain name="mycompany.com4" />
        </trust>
        <handlers>
            <handler class="com.mycompany.CustomHandler">
                <handler-parameter name="param1" value="paramValue1"/>
                <handler-parameter name="param2" value="paramValue2"/>
                <handler-parameter name="param3" value="paramValue3"/>
            </handler>
        </handlers>
```

```
    </identity-provider>

    <service-providers>
      <service-provider alias="sales.war" post-binding="true" security-domain="sp" url="http://
localhost:8080/sales/" supportsSignatures="false">
            <handlers>
                <handler class="com.mycompany.CustomHandler">
                    <handler-parameter name="param1" value="paramValue1"/>
                    <handler-parameter name="param2" value="paramValue2"/>
                    <handler-parameter name="param3" value="paramValue3"/>
                </handler>
            </handlers>
        </service-provider>
                    <service-provider   alias="employee.war"   post-binding="true"   security-
domain="sp" url="http://localhost:8080/employee/" supportsSignatures="false" />
    </service-providers>
  </federation>
</subsystem>
```

Note

If you are looking for more examples about how to use the domain model, take a look at https://github.com/picketlink/picketlink-as-subsystem/blob/master/src/test/resources/picketlink-subsystem.xml.

10.6.3. Usage Examples

This section will guide you through the basic steps to get an Identity Provider and a Service Provider working using the subsystem configuration.

Download the PicketLink Federation Quickstarts from https://repository.jboss.org/nexus/content/groups/public/org/picketlink/quickstarts/picketlink-quickstarts/2.1.8.Final/picketlink-quickstarts-2.1.8.Final-webapps-jboss-as7.zip.

Extract the file and copy the idp.war and sales-post.war to${JBOSS.HOME.dir}/standalone/deployments.

Open both files (idp.war and sales-post.war) and remove the following configuration files:

- WEB-INF/picketlink.xml
- META-INF/jboss-deployment-structure.xml
- WEB-INF/jboss-web.xml

Important

Don't forget to configure the security domains for both applications.

Open the standalone.xml and add the following configuration for the PicketLink subsystem:

```
<subsystem xmlns="urn:jboss:domain:picketlink:1.0">
  <federation alias="example-federation">
      <!-- Identity Provider configuration -->
                                      <identity-provider      alias="idp.war"      security-
domain="idp" supportsSignatures="false" url="http://localhost:8080/idp/">
          <trust>
              <trust-domain name="localhost" />
          </trust>
      </identity-provider>

      <!-- Service Provider configuration -->
      <service-providers>
                    <service-provider  alias="sales-post.war"  post-binding="false"  security-
domain="sp" url="http://localhost:8080/sales-post/" supportsSignatures="false" />
      </service-providers>
  </federation>
</subsystem>
```

To make sure that everything is ok, please start JBoss AS and try to access the sales application. You should be redirected to the idp application.

10.6.4. Metrics and Statistics

Metrics and statistics can be collected from applications deployed using the PicketLink subsystem. This means you can get some useful information about how your Identity Providers and Service providers are working.

- How many SAML assertions were issued by your identity provider ?
- How many times your identity provider respond to service providers ?
- How many SAML assertions were expired ?
- How many authentications are done by your identity provider ?
- How many errors happened ? Trusted Domain errors, signing errors, etc.

To query those metrics and statistics you can use JBoss CLI as follows:

```
[standalone@localhost:9999   federation=example-federation]   ./identity-provider=idp.war:read-
resource(include-runtime=true)
{
    "outcome" => "success",
    "result" => {
        "alias" => "idp.war",
        "created-assertions-count" => "1",
        "error-response-to-sp-count" => "0",
        "error-sign-validation-count" => "0",
        "error-trusted-domain-count" => "0",
```

```
        "expired-assertions-count" => "0",
        "external" => false,
        "handler" => undefined,
        "login-complete-count" => "0",
        "login-init-count" => "0",
        "response-to-sp-count" => "3",
        "security-domain" => "idp",
        "strict-post-binding" => false,
        "supportsSignatures" => false,
        "url" => "http://localhost:8080/idp",
        "trust-domain" =>{"localhost" => undefined}
    }
}
```

10.7. Management Capabilities

One of the benefits about using the PicketLink subsystem to deploy your applications is that they can be managed in different ways:

- **PicketLink Console**

  The console provides a UI, based on the AS7 Administration Console, to help manage your PicketLink deployments. Basically, all the configuration defined in the domain model can be managed using the console.

- **JBoss AS7 CLI Interface (Native Interface)**

  The CLI interface provides a command line tool from where you can query and change all the configuration defined for your applications.

- **JBoss AS7 HTTP Interface**

  JBoss AS7 allows you to manage your running installations using the HTTP protocol with a JSON encoded protocol and a de-typed RPC style API.

> **Note**
>
> Currently the PicketLink Console supports only the federation domain model.

Chapter 11. Federation

11.1. Overview

In this chapter, we look at PicketLink `single sign on (SSO)` and `trust` features. We describe `SAML` SSO in detail.

11.2. SAML SSO

SAML is an `OASIS Standards Consortium` standard for single sign on. PicketLink suppors `SAML v2.0` and `SAML v1.1`.

PicketLink contains support for the following profiles of SAML specification.

- SAML Web Browser SSO Profile.
- SAML Global Logout Profile.

11.3. SAML Web Browser Profile

PicketLink supports the following standard bindings:

- SAML HTTP Redirect Binding
- SAML HTTP POST Binding

11.4. PicketLink SAML Specification Support

PicketLink aims to provide support for both SAML v1.1 and v2.0 specifications. The emphasis is on SAMLv2.0 as v1.1 is deprecated.

11.5. SAML v2.0

11.5.1. Which Profiles are supported ?

- SAML2 Web Browser Profile
- SAML2 Metadata Profile
- SAML2 Logout Profile

11.5.2. Which Bindings are supported ?

The SAML v2 specification defines the concept of SAML protocol bindings (or just bindings). These bindings defines how SAML request-response messages are exchanged onto standard messaging or communication protocols. Currently, PicketLink support the following bindings:

- SAML HTTP Redirect Binding
- SAML HTTP POST Binding

11.5.3. PicketLink Identity Provider (PIDP)

11.5.3.1. Introduction

The Identity Provider is the authoritative entity responsible for authenticating an end user and asserting an identity for that user in a trusted fashion to trusted partners.

> **Tip**
>
> Please look at the PicketLink Quickstarts [https://docs.jboss.org/author/pages/viewpage.action?pageId=23986289] for the PicketLink Identity Provider web application. The quickstarts are useful resources where you can get configuration files.

11.5.3.2. How to create your own PicketLink Identity Provider

The best way to create your own Identity Provider implementation is using one of the examples provided by the PicketLink Quickstarts.

You should also take a look at the following documentations:

- Section 11.5.3.4, “Identity Provider Configuration”
- Section 11.5.3.3, “Identity Provider Authenticators”
- Section 11.5.3.5, “Identity Stores”

11.5.3.3. Identity Provider Authenticators

11.5.3.3.1. Introduction

The PicketLink Identity Provider Authenticator is a component responsible for the authentication of users and for issue and validate SAML assertions.

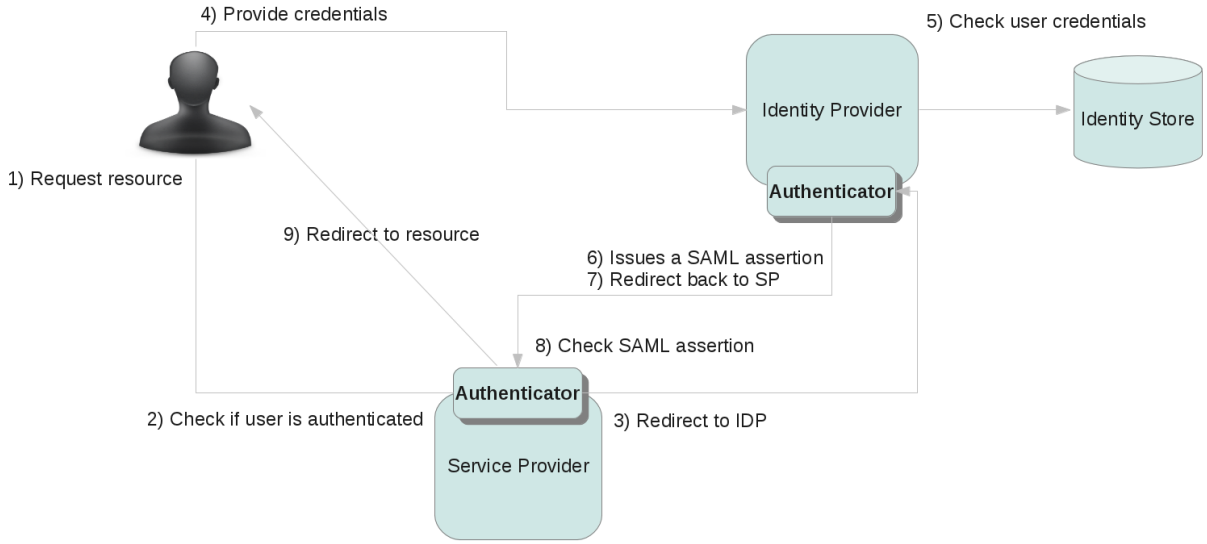


Figure 11.1. TODO InformalFigure image title empty

11.5.3.3.2. Configuring an Authenthicator for a Identity Provider

The PicketLink Authenticator is basically a Tomcat Valve [http://tomcat.apache.org/tomcat-6.0-doc/config/valve.html] (org.apache.catalina.authenticator.FormAuthenticator). The only thing you need to do is change the valves configuration for your application.

This configuration changes for each supported binding.

11.5.3.3.2.1. JBoss Application Server v7

In JBoss Application Server v7 the valves configuration are located inside the **WEB-INF/jboss-web.xml** file. Bellow is a example of how this file looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-web>
 <security-domain>idp</security-domain>
 <context-root>idp</context-root>
 <valve>
    <class-name>org.picketlink.identity.federation.bindings.tomcat.idp.IDPWebBrowserSSOValve</
class-name>
 </valve>
</jboss-web>
```

The valve configuration is done using the **<valve>** element.

11.5.3.3.2.2. JBoss Application Server v5 or v6

In JBoss Application Server v5 or v6, the valves configuration are located inside the **WEB-INF/context.xml** file. Bellow is a example of how this file looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<Context>
<Valve
 className="org.picketlink.identity.federation.bindings.tomcat.idp.IDPWebBrowserSSOValve" />
</Context>
```

The valve configuration is done using the **<Valve>** element.

11.5.3.3.2.3. Apache Tomcat 6

In Apache Tomcat 6 the valves configuration are located inside the **META-INF/context.xml** file. Bellow is a example of how this file looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<Context>
<Valve
 className="org.picketlink.identity.federation.bindings.tomcat.idp.IDPWebBrowserSSOValve" />
</Context>
```

The valve configuration is done using the **<Valve>** element.

11.5.3.3.3. Built-in Authenticators

PicketLink provides default implementations for Service Provider Authenticators. The list bellow shows all the available implementations:

Name	***Description***
org.picketlink.identity.federation.bindings.tomcat.idp.IDPWebBrowserSSOValve	Default implementation for an Identity Provider Authenticator.

11.5.3.3.4. IDPWebBrowserSSOValve

IDPWebBrowserSSOValve from PicketLink provides the core IDP functionality on JBoss Application Server or Apache Tomcat.

11.5.3.3.4.1. Configuration

11.5.3.3.4.1.1. JBoss Application Server v6 and v5.x

Configure in WEB-INF/context.xml

11.5.3.3.4.1.2.

11.5.3.3.4.1.3. Apache Tomcat 5.5 and 6

Configure in META-INF/context.xml

11.5.3.3.4.1.4.

11.5.3.3.4.1.5. Example:

Example 11.1. context.xml

```
<Context>
<Valve
 className="org.picketlink.identity.federation.bindings.tomcat.idp.IDPWebBrowserSSOValve"
           signOutgoingMessages="false"
           ignoreIncomingSignatures="true"/>
</Context>
```

11.5.3.3.4.2.

11.5.3.3.4.3. Attributes

#	Name	Type	Objective	Since version
1	attributeList	String	a comma separated list of attribute keys IDP interested in	2.0
2	configProvider	String	an *optional* implementation of the SAMLConfigurationProvider interface. Provide the fully qualified name.	2.0
3	ignoreIncomingSignatures	boolean	if the IDP should ignore the signatures on the incoming messages Default: false	2.0 **Deprecated since 2.1.2.**
4	ignoreAttributesGeneration	boolean	if the IDP should not generate attribute statements in response to Service Providers	2.0
5	signOutgoingMessages	boolean	Should the IDP sign the outgoing messages? Default: true	2.0 **Deprecated since 2.1.2.**
6	roleGenerator	String	optional fqn of a role generator Default: org.picketlink.identity.	2.0 **Deprecated**

#	Name	Type	Objective	Since version
			federation.bindings. tomcat.TomcatRoleGenerator	**since 2.1.2.**
7	samlHandlerChainClass	String	fqn of a custom SAMLHandlerChain implementation	2.0 **Deprecated since 2.1.2.**
8	identityParticipantStack	String	fqn of a custom IdentityParticipantStack	2.0 **Deprecated since 2.1.2.**

11.5.3.4. Identity Provider Configuration

11.5.3.4.1. Configuring a Identity Provider

To configure an application as a PicketLink Identity Provider you need to follow this steps:

1. Configure the web.xml.
2. Configure an **Authenticator** .
3. Configure a **Security Domain** for your application.
4. Configure **PicketLink JBoss Module** [https://docs.jboss.org/author/display/PLINK/JBoss+Modules] as a dependency.
5. Create and configure a file named **WEB-INF/picketlink.xml** .

11.5.3.4.2. Configuring the web.xml

Before configuring your application as an Identity Provider you need to add some configurations to your web.xml.

Let's start by defining a **security-constraint** element to restrict access to resources from unauthenticated users:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Manager command</web-resource-name>
        <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>manager</role-name>
    </auth-constraint>
</security-constraint>
```

```
<security-role>
    <description>
      The role that is required to log in to IDP Application
    </description>
    <role-name>manager</role-name>
</security-role>
```

As you can see above, we define that only users with a role named **manager** are allowed to access the protected resources. Make sure to give your users the same role you defined here, otherwise they will get a 403 HTTP status code.

The next step is define your *FORM* login configuration using the **login-config** element:

```
<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>PicketLink IDP Application</realm-name>
    <form-login-config>
        <form-login-page>/jsp/login.jsp</form-login-page>
        <form-error-page>/jsp/login-error.jsp</form-error-page>
    </form-login-config>
</login-config>
```

Make sure you have inside your application the pages defined in the elements **form-login-page** and **form-error-page** .

> **Important**
>
> Please, make sure you have a welcome file page in your application. You can define it in your web.xml or simply create an **index.jsp** at the root directory of your application.

11.5.3.4.3. The picketlink.xml configuration file

All the configuration for an especific Identity Provider goes at the WEB-INF/picketlink.xml file. This file is responsible to define the behaviour of the Authenticator. During the identity provider startup, the authenticator parses this file and configures itself.

Bellow is how the picketlink.xml file should looks like:

```
<PicketLink xmlns="urn:picketlink:identity-federation:config:2.1">


    <PicketLinkIDP xmlns="urn:picketlink:identity-federation:config:2.1">

        <IdentityURL>http://localhost:8080/idp/ </IdentityURL>
```

```
        <Trust>
            <Domains>locahost,mycompany.com</Domains>
        </Trust>

      <KeyProvider ClassName="org.picketlink.identity.federation.core.impl.KeyStoreKeyManager">

            <Auth Key="KeyStoreURL" Value="/jbid_test_keystore.jks" />
            <Auth Key="KeyStorePass" Value="store123" />
            <Auth Key="SigningKeyPass" Value="test123" />
            <Auth Key="SigningKeyAlias" Value="servercert" />

            <ValidatingAlias Key="localhost" Value="servercert" />
            <ValidatingAlias Key="127.0.0.1" Value="servercert" />

        </KeyProvider>

    </PicketLinkIDP>

      <PicketLinkSTS xmlns="urn:picketlink:identity-federation:config:1.0" TokenTimeout="1000"
 ClockSkew="1000">
        <TokenProviders>
                                                                        <TokenProvider
 ProviderClass="org.picketlink.identity.federation.core.saml.v2.providers.SAML20AssertionTokenProvider"
               TokenType="urn:oasis:names:tc:SAML:2.0:assertion" TokenElement="Assertion"
               TokenElementNS="urn:oasis:names:tc:SAML:2.0:assertion" />
        </TokenProviders>
    </PicketLinkSTS>

    <Handlers xmlns="urn:picketlink:identity-federation:handler:config:2.1">

                                                                          <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2IssuerTrustHandler" />
                                                                          <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2LogOutHandler" />
                                                                          <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2AuthenticationHandler" />
                                                                          <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.RolesGenerationHandler" />

     </Handlers>

</PicketLink>
```

> **Important**
>
> The schema for the picketlink.xml file is available here: https://github.com/picketlink/federation/blob/master/picketlink-core/src/main/resources/schema/config/picketlink_v2.1.xsd .

11.5.3.4.3.1. PicketLinkIDP Element

This element defines the basic configuration for the identity provider. The table bellow provides more information about the attributes supported by this element:

Name	Description	Value
AssertionValidity	Defines the timeout for the SAML assertion validity, in miliseconds.	Defaults to **300000** . *Deprecated. Use the PicketLinkSTS element, instead.*
RoleGenerator	Defines the name of the **org.picketlink. identity.federation. core.interfaces. RoleGenerator** subclass to be used to obtain user roles.	Defaults to **org.picketlink.identity. federation.core. impl.EmptyRoleGenerator** .
AttributeManager	Defines the name of the **org.picketlink. identity.federation. core.interfaces. AttributeManager** subclass to be used to obtain the SAML assertion attributes.	Defautls to **org.picketlink. identity.federation. core.impl. EmptyAttributeManager** .
StrictPostBinding	SAML Web Browser SSO Profile has a requirement that the IDP does not respond back in Redirect Binding. Set this to false if you want to force the IDP to respond to SPs using the Redirect Binding.	Values: **true|false** . Defaults to true, the IDP always respond via POST Binding.
SupportsSignatures	Indicates if digital signature/ verification of SAML assertions are enabled. If this attribute is marked to true the Service Providers must support signatures too, otherwise the SAML messages will be considered as invalid.	Values: **true|false.** Defaults to false.
Encrypt	Indicates if SAML Assertions should be encrypted. If this attribute is marked to true the Service Providers must support signatures too, otherwise the SAML messages will be considered as invalid.	Values: **true|false.** Defaults to false

Name	Description	Value
IdentityParticipantStack	Defines the name of the **org.picketlink.identity.federation.web.core.IdentityParticipantStack** subclass to be used to register and deregister participants in the identity federation.	Defaults to **org.picketlink.identity.federation.web.core.IdentityServer.STACK.**

11.5.3.4.3.1.1. IdentityURL Element

This element value refers to the URL of the Identity Provider.

Eg.: http://localhost:8080/idp/

11.5.3.4.3.1.2. Trust/Domains Elements

The Trust and Domains elements defines the hosts trusted by this Identity Provider. You just need to inform a list of comma separated domain names.

11.5.3.4.3.1.3. SAML Digital Signature Configuration (KeyProvider Element)

To enable digital signatures for the SAML assertions you need to configure:

1. Set the **SupportsSignature** attribute to true;
2. Add the Section 11.5.7.11, “SAML2SignatureGenerationHandler” and the Section 11.5.7.12, “SAML2SignatureValidationHandler” in the handlers chain (Handler Element).
3. Configure a **KeyProvider** * *element.

11.5.3.4.3.1.4. SAML Encryption Configuration

To enable encryption for SAML assertions you need to configure:

1. Set the **Encrypt** attribute to true;
2. Add the **Section 11.5.7.8, “SAML2EncryptionHandler”** and the Section 11.5.7.12, “SAML2SignatureValidationHandler” in the handlers chain (Handler Element).
3. Configure a **KeyProvider** * *element.

11.5.3.4.3.2. SAML Handlers Configuration (Handlers Element)

PicketLink provides some built-in Handlers to help the Identity Provider Authenticator processing the SAML requests and responses.

The handlers are configured through the **Handlers** element.

11.5.3.4.4.3.3. SecurityToken Service Configuration (PicketLinkSTS Element)

> **Important**
>
> When configuring the IDP, you do not need to specify the PicketLinkSTS element in the configuration. If it is ommited PicketLink will load the default configurations from a file named core-sts inside the picketlink-core-VERSION.jar.
>
> Override this configuration only if you need to. Eg.: change the token timeout or specify a custom Security Token Provider for SAML assertions.

See the documentation at Section 11.5.3.6, “Security Token Service Configuration” .

11.5.3.5. Identity Stores

11.5.3.5.1. Introduction

The Identity Provider needs a Identity Store to retrieve users information. These informations will be used during the authentication and authorization process. Identity Stores can be any type of repository: a database, LDAP, properties file, etc.

The PicketLink Identity Provider uses JAAS to connect to an Identity Store. This configuration is usually made at the container side using any LoginModule implementation.

If you are using the JBoss Application Server you can use one of the existing LoginModules or you can create your custom implementation:

- https://community.jboss.org/wiki/JBossAS7SecurityDomainModel

11.5.3.5.2. Configuring a Security Domain for a Identity Store

In order to authenticate users, the Identity Provider needs to be configured with the properly security domain configuration. The security domain is responsible for authenticating the user in a specific Identity Store.

This is done by defining a **<security-domain>** element in jboss-web.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-web>
    <security-domain>idp</security-domain>
    <valve>
     <class-name>org.picketlink.identity.federation.bindings.tomcat.idp.IDPWebBrowserSSOValve</
class-name>
    </valve>
```

```
</jboss-web>
```

In order to use the security domain above, you need to configure it in your server. For JBoss AS7 you just need to add the following configuration to standalone.xml:

```
<subsystem xmlns="urn:jboss:domain:security:1.1">
    <security-domains>
        <security-domain name="idp" cache-type="default">
            <authentication>
                <login-module code="UsersRoles" flag="required">
                    <module-option name="usersProperties" value="users.properties"/>
                    <module-option name="rolesProperties" value="roles.properties"/>
                </login-module>
            </authentication>
    </security-domain>

    ...

</subsystem>
```

The example above uses a JAAS LoginModule that uses two properties files to authenticate users and retrieve their roles. These properties files needs to be located at WEB-INF/classes folder.

11.5.3.6. Security Token Service Configuration

11.5.3.6.1. SecurityToken Service Configuration (PicketLinkSTS Element)

To issue/renew/cancel/validate SAML tokens, the IDP relies on the PicketLink STS API and configuration. This configurations define how the tokens should be used by the IDP.

This *PicketLinkSTS* element defines the basic configuration for the Security Token Service. The table bellow provides more information about the attributes supported by this element:

Name	Description	Value
STSName	Name for this STS configuration.	Name for this Security Token Service.
TokenTimeout	Defines the token timeout in miliseconds.	Defaults to **3600** miliseconds.
ClockSkew	Defines the clock skew, or timing skew, for the token timeout.	Defaults to **2000** miliseconds.
SignToken	Indicates if the tokens should be signed.	Values: **true|false** . Defaults to **false** .
EncryptToken	Indicates if the tokens should be encrypted.	Values: **true|false** . Defaults to **false** .

Name	Description(PSP)	Value
CanonicalizationMethod	Sets the canonicalization method.	Defaults to http://www.w3.org/2001/10/xml-exc-c14n#WithComments

11.5.3.6.1.1. Security Token Providers (TokenProviders/TokenProvider elements)

The PicketLink STS defines the concept of *Security Token Providers.* This tokens providers are implementations of the interface **org.picketlink.identity.federation.core.interfaces.SecurityTokenProvider** .

The purpose of providers is to plug any implementation for a specific token type. PicketLink provides default implementations for the following token type:

- **SAML v2.0** : *org.picketlink.identity.federation.core.saml.v2.providers.SAML20AssertionTokenProvider*
- **WS-Trust** _ : org.picketlink.identity.federation.core.wstrust.plugins.saml.SAML20TokenProvider_

Each provider is linked to a specific *TokenType* and *TokenElementNS,* both attributes of the *TokenProvider* element.

You can always provide your own implementation for a specific TokenType or customize the behaviour for one of the built-in providers.

11.5.4. PicketLink Service Provider (PSP)

11.5.4.1. Introduction

The PicketLink Service Provider relies on the PicketLink Identity Provider to assert information about a user via an electronic user credential, leaving the service provider to manage access control and dissemination based on a trusted set of user credential assertions.

> **Tip**
>
> Please have a look at the PicketLink Quickstarts [https://docs.jboss.org/author/pages/viewpage.action?pageId=23986289] to obtain service provider applications. The quickstarts are useful resources where you can get configuration files.

11.5.4.2. How to create your own PicketLink Service Provider

The best way to create your own Service Provider implementation is using one of the examples provided by the PicketLink Quickstarts.

You should also take a look at the following documentations:

- Section 11.5.4.3, “Service Provider Configuration”
- Section 11.5.4.4, “Service Provider Authenticators”
- Configuring a SAML Security Domain

11.5.4.3. Service Provider Configuration

11.5.4.3.1. Configuring a Service Provider

To configure an application as a PicketLink Service Provider you need to follow this steps:

1. Configuring the web.xml.
2. Configure an **Authenticator** .
3. Configure a **Security Domain** for your application.
4. Configure **PicketLink JBoss Module** [https://docs.jboss.org/author/display/PLINK/JBoss+Modules] as a dependency.
5. Create and configure a file named **WEB-INF/picketlink.xml** .

11.5.4.3.2. Configuring the web.xml

Before configuring your application as an Service Provider you need to add some configurations to your web.xml.

Let's start by defining a **security-constraint** element to restrict access to resources from unauthenticated users:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Manager command</web-resource-name>
        <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>manager</role-name>
    </auth-constraint>
</security-constraint>

<security-role>
    <description>
      The role that is required to log in to the Manager Application
    </description>
    <role-name>manager</role-name>
```

```
</security-role>
```

As you can see above, we define that only users with a role named **manager** are allowed to access the protected resources. Make sure to give your users the same role you defined here, otherwise they will get a 403 HTTP status code.

During the logout process, PicketLink will try to redirect the user to a **logout.jsp** page located at the root directory of your application. Please, make sure to create it.

> **Important**
>
> Please, make sure you have a welcome file page in your application. You can define it in your web.xml or simply create an **index.jsp** at the root directory of your application.

11.5.4.3.3. The picketlink.xml configuration file

All the configuration for an especific Service Providers goes at the WEB-INF/picketlink.xml file. This file is responsible to define the behaviour of the Authenticator. During the service provider startup, the authenticator parses this file and configures itself.

Bellow is how the picketlink.xml file should looks like:

```
<PicketLink xmlns="urn:picketlink:identity-federation:config:2.1">


    <PicketLinkSP xmlns="urn:picketlink:identity-federation:config:2.1"
        BindingType="REDIRECT"
        RelayState="someURL"
        ErrorPage="/someerror.jsp"
        LogOutPage="/customLogout.jsp"
        IDPUsesPostBinding="true"
        SupportsSignatures="true">

        <IdentityURL>http://localhost:8080/idp/ </IdentityURL>
        <ServiceURL>http://localhost:8080/employee/ </ServiceURL>

      <KeyProvider ClassName="org.picketlink.identity.federation.core.impl.KeyStoreKeyManager">

            <Auth Key="KeyStoreURL" Value="/jbid_test_keystore.jks" />
            <Auth Key="KeyStorePass" Value="store123" />
            <Auth Key="SigningKeyPass" Value="test123" />
            <Auth Key="SigningKeyAlias" Value="servercert" />

            <ValidatingAlias Key="localhost" Value="servercert" />
            <ValidatingAlias Key="127.0.0.1" Value="servercert" />

        </KeyProvider>

    </PicketLinkSP>
```

```
    <Handlers xmlns="urn:picketlink:identity-federation:handler:config:2.1">

                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2IssuerTrustHandler" />
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2LogOutHandler" />
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2AuthenticationHandler" />
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.RolesGenerationHandler" />

     </Handlers>

</PicketLink>
```

> **Important**
>
> The schema for the picketlink.xml file is available here: https://github.com/picketlink/federation/blob/master/picketlink-core/src/main/resources/schema/config/picketlink_v2.1.xsd .

11.5.4.3.3.1. PicketLinkSP Element

This element defines the basic configuration for the service provider. The table bellow provides more information about the attributes supported by this element:

Name	Description	Value
BindingType	Defines which SAML binding should be used: SAML HTTP POST or Redirect bindings.	POST|REDIRECT. Defaults to POST if no specified.
ErrorPage	Defines a custom error page to be displayed when some error occurs during the request processing.	Defaults to /error.jsp.
LogOutPage	Defines a custom logout page to be displayed after the logout.	Defaults to /logout.jsp.
IDPUsesPostBinding	Indicates if the Identity Provider configured for this Service Provider is always using POST for SAML responses.	true|false. Defaults to true if no specified.
SupportsSignature	Indicates if digital signature/ verification of SAML	true|false. Defaults to false if no specified.

Name	Description(PSP)	Value
	assertions are enabled. If this attribute is marked to true the Identity Provider configured for this Service Provider must support signatures too, otherwise the SAML messages will be considered as invalid.	

11.5.4.3.3.1.1. IdentityURL Element

This element value refers to the URL of the Identity Provider used by this Service Provider.

Eg.: http://localhost:8080/idp/

11.5.4.3.3.1.2. ServiceURL Element

This element value refers to the URL of the Service Provider.

Eg.: http://localhost:8080/sales/

11.5.4.3.3.2. SAML Digital Signature Configuration (KeyProvider Element)

To enable digital signatures for the SAML assertions you need to configure:

1. Set the **SupportsSignature** attribute to true;
2. Add the Section 11.5.7.11, “SAML2SignatureGenerationHandler” and the Section 11.5.7.12, “SAML2SignatureValidationHandler” in the handlers chain (Handler Element).
3. Configure a **KeyProvider** * *element.

11.5.4.3.3.3. SAML Handlers Configuration (Handlers Element)

PicketLink provides some built-in Handlers to help the Service Provider Authenticator processing the SAML requests and responses.

The handlers are configured through the **Handlers** element.

11.5.4.4. Service Provider Authenticators

11.5.4.4.1. Introduction

PicketLink Service Providers Authenticators are important components responsible for the authentication of users using the SAML Assertion previously issued by an Identity Provider.

PicketLink Service Provider (PicketLink SP)

They are responsible for intercepting each request made to an application, checking if a SAML assertion is present in the request, validating its signature and executing SAML specific validations and creating a security context for the user in the requested application.

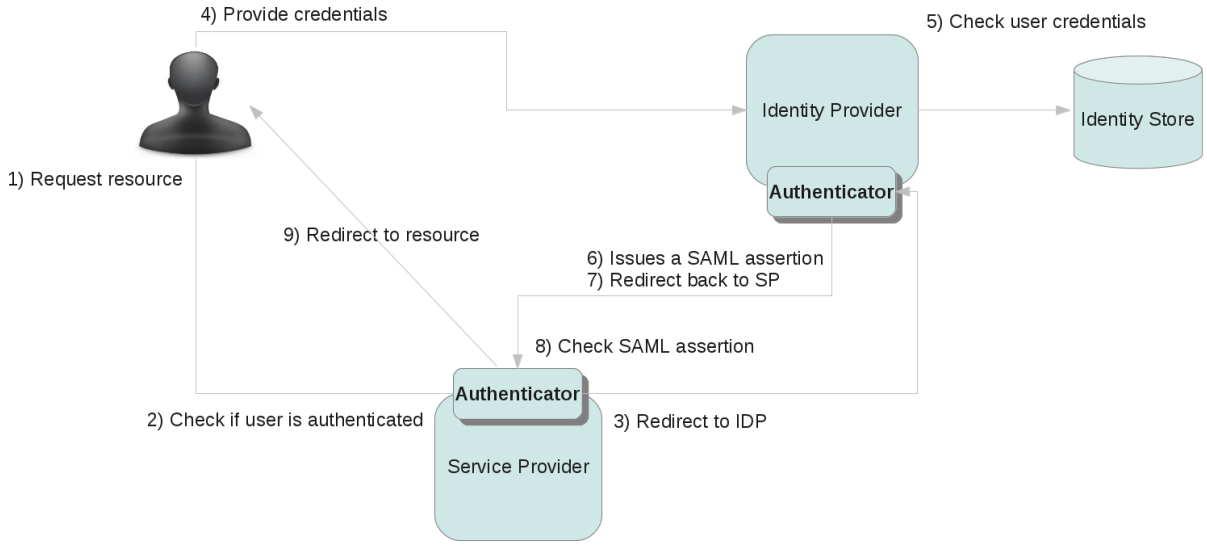


Figure 11.2. TODO InformalFigure image title empty

11.5.4.4.2. Configuring an Authenthicator for a Service Provider

The PicketLink Authenticator is basically a Tomcat Valve [http://tomcat.apache.org/tomcat-6.0-doc/config/valve.html] (org.apache.catalina.authenticator.FormAuthenticator). The only thing you need to do is change the valves configuration for your application.

This configuration changes for each supported binding.

11.5.4.4.2.1. JBoss Application Server v7

In JBoss Application Server v7 the valves configuration are located inside the **WEB-INF/jboss-web.xml** file. Bellow is a example of how this file looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-web>
 <security-domain>sp</security-domain>
 <context-root>employee</context-root>
 <valve>
```

```
                                                                                <class-
 name>org.picketlink.identity.federation.bindings.tomcat.sp.ServiceProviderAuthenticator</
 class-name>
  </valve>
 </jboss-web>
```

The valve configuration is done using the **<valve>** element.

11.5.4.4.2.2. JBoss Application Server v5 or v6

In JBoss Application Server v5 or v6, the valves configuration are located inside the **WEB-INF/context.xml** file. Bellow is a example of how this file looks like:

```
 <?xml version="1.0" encoding="UTF-8"?>
 <Context>
                                                                                <Valve
className="org.picketlink.identity.federation.bindings.tomcat.sp.ServiceProviderAuthenticator"
 >
 </Context>
```

The valve configuration is done using the **<Valve>** element.

11.5.4.4.2.3. Apache Tomcat 6

In Apache Tomcat 6 the valves configuration are located inside the **META-INF/context.xml** file. Bellow is a example of how this file looks like:

```
 <?xml version="1.0" encoding="UTF-8"?>
 <Context>
                                                                                <Valve
className="org.picketlink.identity.federation.bindings.tomcat.sp.ServiceProviderAuthenticator"
 >
 </Context>
```

The valve configuration is done using the **<Valve>** element.

11.5.4.4.3. Built-in Authenticators

PicketLink provides default implementations for Service Provider Authenticators. The list bellow shows all the available implementations:

Name	***Description***
org.picketlink.identity. federation.bindings. tomcat.sp. ServiceProviderAuthenticator	Preferred service provider authenticator. Supports both SAML HTTP Redirect and POST bindings.
org.picketlink. identity.federation. bindings.tomcat. sp.SPPostFormAuthenticator	**Deprecated** . Supports only HTTP POST Binding without signature of SAML assertions.

Name	PicketLink Service Provider (PSP) Description
org.picketlink.identity.federation.bindings.tomcat.sp.SPPostSignatureFormAuthenticator	**Deprecated** . Supports only HTTP POST Binding with signature of SAML assertions.
org.picketlink.identity.federation.bindings.tomcat.sp.SPRedirectFormAuthenticator	**Deprecated** . Supports only HTTP Redirect Binding without signature of SAML assertions.
org.picketlink.identity.federation.bindings.tomcat.sp.SPRedirectSignatureFormAuthenticator	**Deprecated** . Supports only HTTP Redirect Binding with signature of SAML assertions.

> **Warning**
>
> Prefer using the ??? ServiceProviderAuthenticator authenticator if you are using PicketLink v.2.1 or above. The others authenticators are **DEPRECATED** .

11.5.4.4.4. ServiceProviderAuthenticator

As of PicketLink v2.1, the ServiceProviderAuthenticator is the preferred Service Provider configuration to the deprecated Section 11.5.4.4.8, “SPPostFormAuthenticator” , Section 11.5.4.4.6, “SPRedirectFormAuthenticator” , Section 11.5.4.4.7, “SPPostSignatureFormAuthenticator” and Section 11.5.4.4.5, “SPRedirectSignatureFormAuthenticator” .

11.5.4.4.4.1. Configuration

https://docs.jboss.org/author/display/PLINK/Service+Provider+Configuration

11.5.4.4.4.2.

11.5.4.4.5. SPRedirectSignatureFormAuthenticator

> **Warning**
>
> As of PicketLink v2.1, the Section 11.5.4.4.4, “ServiceProviderAuthenticator” is the preferred Service Provider configuration to the **deprecated** Section 11.5.4.4.8, “SPPostFormAuthenticator” , Section 11.5.4.4.6, “SPRedirectFormAuthenticator” , Section 11.5.4.4.7, “SPPostSignatureFormAuthenticator” and Section 11.5.4.4.5, “SPRedirectSignatureFormAuthenticator” .

SPRedirectSignatureFormAuthenticator is used to provide signature/encryption services to a Service Provider (SP) application for HTTP/Redirect binding of SAMLv2 specification. This authenticator

is an extension of the Section 11.5.4.4.6, “SPRedirectFormAuthenticator” .

11.5.4.4.5.1. Binding

HTTP/Redirect Binding (along with signature/encryption support)

11.5.4.4.5.2. Configuration

11.5.4.4.5.2.1. JBoss Application Server v5.x/6

Configure in WEB-INF/context.xml

11.5.4.4.5.2.2. Apache Tomcat v5.5/6.x

Configure in META-INF/context.xml

11.5.4.4.5.2.3.

11.5.4.4.5.2.4. Example:

Example 11.2. context.xml

```
<Context>
                                                                                <Valve
 className="org.picketlink.identity.federation.bindings.tomcat.sp.SPRedirectSignatureFormAuthenticator"
  />
</Context>
```

11.5.4.4.5.2.5. Attributes

#	Name	Type	Objective	Since
1	configFile	String	optional - fully qualified location of the config file Default: /WEB-INF/picketlink-idfed.xml	2.0
2	samlHandlerChainClass	String	optional - fqn of a custom SAMLHandlerChain implementation	2.0
3	serviceURL	String	optional - the service provider URL	2.0
4	saveRestoreRequest	boolean	should the authenticator save the original request and restore it after authentication Default: true	2.0
5	configProvider	String	optional - a fqn of the SAMLConfigurationProvider implementation	2.0
6	issuerID	String	optional - customize the issuer id	2.0

#	Name	Type	Objective	Since
7	idpAddress	String	optional - If the request.getRemoteAddr is not exactly the IDP address that you have keyed in your deployment descriptor for keystore alias, you can configure it explicitly	2.0

11.5.4.4.6. SPRedirectFormAuthenticator

> **Warning**
>
> As of PicketLink v2.1, the Section 11.5.4.4.4, “ServiceProviderAuthenticator” is the preferred Service Provider configuration to the **deprecated** Section 11.5.4.4.8, “SPPostFormAuthenticator” , Section 11.5.4.4.6, “SPRedirectFormAuthenticator” , Section 11.5.4.4.7, “SPPostSignatureFormAuthenticator” and Section 11.5.4.4.5, “SPRedirectSignatureFormAuthenticator” .

SPRedirectFormAuthenticator provides the SAMLv2 HTTP/Redirect binding support for service provider (SP) applications.

11.5.4.4.6.1. Binding

SAMLv2 HTTP/Redirect Binding

11.5.4.4.6.2. Configuration

11.5.4.4.6.2.1. JBoss Application Server v5.x/6

Configure in WEB-INF/context.xml

11.5.4.4.6.2.2. Apache Tomcat v5.5/6.x

Configure in META-INF/context.xml

11.5.4.4.6.2.3.

11.5.4.4.6.2.4. Example:

Example 11.3. context.xml

```
<Context>
                                                                                <Valve
 className="org.picketlink.identity.federation.bindings.tomcat.sp.SPRedirectFormAuthenticator"
  />
</Context>
```

PicketLink Service Provider (PSP)

11.5.4.4.6.2.5. Attributes

#	Name	Type	Objective	Since
1	configFile	String	optional - fully qualified location of the config file Default: /WEB-INF/picketlink-idfed.xml	2.0
2	samlHandlerChainClass	String	optional - fqn of a custom SAMLHandlerChain implementation	2.0
3	serviceURL	String	optional - the service provider URL	2.0
4	saveRestoreRequest	boolean	should the authenticator save the original request and restore it after authentication Default: true	2.0
5	configProvider	String	optional - a fqn of the SAMLConfigurationProvider implementation	2.0
6	issuerID	String	optional - customize the issuer id	2.0

11.5.4.4.7. SPPostSignatureFormAuthenticator

> **Warning**
>
> As of PicketLink v2.1, the Section 11.5.4.4.4, “ServiceProviderAuthenticator” is the preferred Service Provider configuration to the **deprecated** Section 11.5.4.4.8, “SPPostFormAuthenticator” , Section 11.5.4.4.6, “SPRedirectFormAuthenticator” , Section 11.5.4.4.7, “SPPostSignatureFormAuthenticator” and Section 11.5.4.4.5, “SPRedirectSignatureFormAuthenticator” .

SPPostSignatureFormAuthenticator is used to provide signature/encryption services to a Service Provider (SP) application for HTTP/POST binding of SAMLv2 specification. This authenticator

is an extension of the Section 11.5.4.4.8, “SPPostFormAuthenticator” .

11.5.4.4.7.1. Binding

HTTP/POST Binding (along with signature/encryption support)

11.5.4.4.7.2. Configuration

11.5.4.4.7.2.1. JBoss Application Server v5.x/6

Configure in WEB-INF/context.xml

11.5.4.4.7.2.2. Apache Tomcat v5.5/6.x

Configure in META-INF/context.xml

11.5.4.4.7.2.3.

11.5.4.4.7.2.4. Example:

Example 11.4. context.xml

```
<Context>
                                                                                <Valve
 className="org.picketlink.identity.federation.bindings.tomcat.sp.SPPostSignatureFormAuthenticator"
  />
</Context>
```

11.5.4.4.7.2.5. Attributes

#	Name	Type	Objective	Since
1	configFile	String	optional - fully qualified location of the config file Default: /WEB-INF/picketlink-idfed.xml	2.0
2	samlHandlerChainClass	String	optional - fqn of a custom SAMLHandlerChain implementation	2.0
3	serviceURL	String	optional - the service provider URL	2.0
4	saveRestoreRequest	boolean	should the authenticator save the original request and restore it after authentication Default: true	2.0
5	configProvider	String	optional - a fqn of the SAMLConfigurationProvider implementation	2.0
6	issuerID	String	optional - customize the issuer id	2.0
7	idpAddress	String	optional - If the request.getRemoteAddr is not exactly the IDP address that you have keyed in your deployment descriptor for keystore alias, you can configure it explicitly	2.0

11.5.4.4.8. SPPostFormAuthenticator

> **Warning**
>
> As of PicketLink v2.1, the Section 11.5.4.4.4, “ServiceProviderAuthenticator” is the preferred Service Provider configuration to the **deprecated** Section 11.5.4.4.8, “SPPostFormAuthenticator” , Section 11.5.4.4.6, “SPRedirectFormAuthenticator” ,

Section 11.5.4.4.7, “SPPostSignatureFormAuthenticator” and Section 11.5.4.4.5, “SPRedirectSignatureFormAuthenticator” .

SPPostFormAuthenticator is the main authenticator used to configure a service provider (SP) application for SAMLv2.0

11.5.4.4.8.1. Binding

SAMLv2 HTTP/Post Binding

11.5.4.4.8.2. Configuration

11.5.4.4.8.2.1. JBoss Application Server v5.x/6

Configure in WEB-INF/context.xml

11.5.4.4.8.2.2. Apache Tomcat v5.5/6.x

Configure in META-INF/context.xml

11.5.4.4.8.2.3.

11.5.4.4.8.2.4. Example:

Example 11.5. context.xml

```
<Context>
                                                                                <Valve
 className="org.picketlink.identity.federation.bindings.tomcat.sp.SPPostFormAuthenticator"
  />
</Context>
```

11.5.4.4.8.2.5. Attributes

#	Name	Type	Objective	Since
1	configFile	String	optional - fully qualified location of the config file Default: /WEB-INF/picketlink-idfed.xml	2.0
2	samlHandlerChainClass	String	optional - fqn of a custom SAMLHandlerChain implementation	2.0
3	serviceURL	String	optional - the service provider URL	2.0
4	saveRestoreRequest	boolean	should the authenticator save the original request and restore it after authentication Default: true	2.0
5	configProvider	String	optional - a fqn of the SAMLConfigurationProvider implementation	2.0

#	Name	Type	Objective	Since
6	issuerID	String	optional - customize the issuer id	2.0

11.5.4.5. Service Provider Security Domain

11.5.4.5.1. Configuring a security domain

In order to handle the SAML assertions returned by the Identity Provider, the Service Provider needs to be configured with the properly security domain configuration. This is done by defining a **<security-domain>** element in jboss-web.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-web>
    <security-domain>sp</security-domain>
    <valve>
                                                                                <class-
name>org.picketlink.identity.federation.bindings.tomcat.sp.ServiceProviderAuthenticator</
class-name>
    </valve>
</jboss-web>
```

In order to use the security domain above, you need to configure it in your server. For JBoss AS7 you just need to add the following configuration to standalone.xml:

```
<subsystem xmlns="urn:jboss:domain:security:1.1">
    <security-domains>
        <security-domain name="sp" cache-type="default">
            <authentication>

  <login-module   code="org.picketlink.identity.federation.bindings.jboss.auth.SAML2LoginModule"
 flag="required"/>
            </authentication>
    </security-domain>

    ...

</subsystem>
```

11.5.5. SAML Authenticators (Tomcat,JBossAS)

11.5.5.1. Introduction

The PicketLink Identity Provider Authenticator is a component responsible for the authentication of users and for issue and validate SAML assertions.

Basically, there two different different authenticator implementations type:

- Identity Provider Authenticators
- Service Provider Authenticators

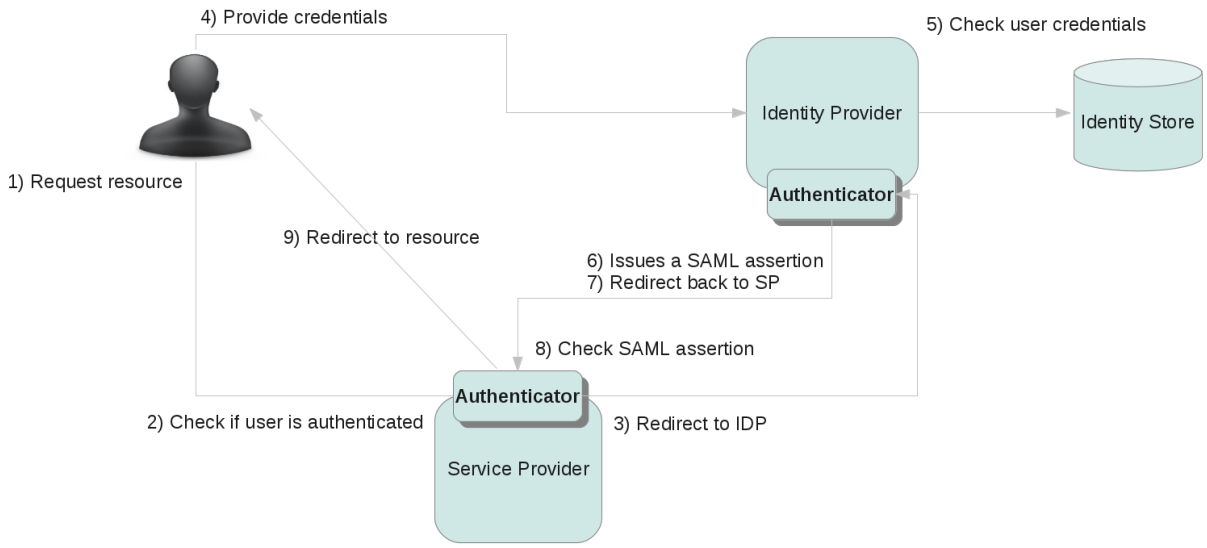


Figure 11.3. TODO InformalFigure image title empty

11.5.5.2. Tomcat Authenticators for use in Apache Tomcat and JBoss Application Server

PicketLink includes a number of Authenticators for providing SAML support on Apache Tomcat and JBoss Application Server.

11.5.5.2.1. Authenticators/Valves for Identity Provider (IDP)

1. Section 11.5.3.3.4, “IDPWebBrowserSSOValve”

11.5.5.2.2. Authenticators/Valves for Service Provider (SP)

1. Section 11.5.4.4.4, “ServiceProviderAuthenticator”

11.5.5.2.2.1. Deprecated (as of PicketLink v2.1)

1. Section 11.5.4.4.8, “SPPostFormAuthenticator”

2. Section 11.5.4.4.6, “SPRedirectFormAuthenticator”
3. Section 11.5.4.4.7, “SPPostSignatureFormAuthenticator”
4. Section 11.5.4.4.5, “SPRedirectSignatureFormAuthenticator”

11.5.5.3.

11.5.5.4. Useful Information

- Tomcat Character Encoding (UTF-8 etc) [http://wiki.apache.org/tomcat/FAQ/CharacterEncoding]

11.5.6. Digital Signatures in SAML Assertions

11.5.6.1. Configuring the KeyProvider

To support digital signatures of SAML assertions you should define a KeyProvider element inside a PicketLinkIDP or PicketLinkSP.

> **Important**
>
> When using digital signatures you need to configure and enable it in both Identity Provider and Service Providers. Otherwise the SAML assertions would probably be considered as invalid.

```
<KeyProvider ClassName="org.picketlink.identity.federation.core.impl.KeyStoreKeyManager">
    <Auth Key="KeyStoreURL" Value="/jbid_test_keystore.jks" />
    <Auth Key="KeyStorePass" Value="store123" />
    <Auth Key="SigningKeyPass" Value="test123" />
    <Auth Key="SigningKeyAlias" Value="servercert" />

    <ValidatingAlias Key="idp.example.com" Value="servercert" />
    <ValidatingAlias Key="localhost" Value="servercert" />
</KeyProvider>
```

In order to configure the KeyProvider, you need to specify some configurations about the Java KeyStore that should be used to sign SAML assertions:

Auth Key	Description
KeyStoreURL	Where the value of the **Value** attribute points to the location of a Java KeyStore with the properly installed certificates.
KeyStorePass	Where the value of the **Value** attribute refers to the password of the referenced Java KeyStore.

Auth Key	Description
SigningKeyAlias	Where the value of the **Value** attribute refers to the password of the installed certificate to be used to sign the SAML assertions.
SigningKeyPass	Where the value of the **Value** attribute refers to the alias of the certificate to be used to sign the SAML assertions.

The Service Provider also needs to know how to verify the signatures for the SAML assertions. This is done by the **ValidationAlias** elements.

```
<ValidatingAlias Key="idp.example.com" Value="servercert" />
```

> **Tip**
>
> Note that we declare the validating certificate for each domain using the *ValidatingAlias* .
>
> At the IDP side you need an entry for each server/domain name defined as a trusted domain (Trust/Domains elements).
>
> At the SP side you need an entry for the the server/domain name where the IDP is deployed.

11.5.6.2. SImple Example Scenario

11.5.6.2.1. How SAML assertions are signed ?

When digital signatures are enable, the authenticator will look at the **SigningKeyAlias** for the alias that should me used to look for a private key configured in the Java KeyStore. This private key will be used to sign the SAML assertion.

11.5.6.2.2. How signatures are validated ?

When digital signatures are enabled, the authenticator will look at the ValidatingAlias table for a entry that matches the value of the **Key** attribute with the host name of the Issuer of the SAML assertion. For example, consider the following SAML Assertion issued by an Identity Provider located at http://idp.example.com:

```
<saml2:Assertion xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion"
    ID="ID_ab0392ef-b557-4453-95a8-a7e168da8ac5" IssueInstant="2010-09-30T19:13:37.869Z"
    Version="2.0">
    <saml2:Issuer>http://idp.example.com </saml2:Issuer>
    <saml2:Subject>
```

```
        <saml2:NameID NameQualifier="urn:picketlink:identity-federation">jduke</saml2:NameID>
        <saml2:SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer" />
    </saml2:Subject>
    <saml2:Conditions NotBefore="2010-09-30T19:13:37.869Z"
         NotOnOrAfter="2010-09-30T21:13:37.869Z" />
    <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:SignedInfo>
                    <ds:CanonicalizationMethod  Algorithm="http://www.w3.org/2001/10/xml-exc-
c14n#WithComments" />
            <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmlds#rsa-sha1" />
            <ds:Reference URI="#ID_ab0392ef-b557-4453-95a8-a7e168da8ac5">
                <ds:Transforms>
               <ds:Transform Algorithm="http://www.w3.org/2000/09/xmlds#enveloped-signature" />
                    <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
                </ds:Transforms>
                <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmlds#sha1" />
                <ds:DigestValue>0Y9QM5c5qCShz5UWmbFzBmbuTus=</ds:DigestValue>
            </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>
            se/flQ2htUQ0IUYieVkXNn9cfjnfgv6H99nFarsTNTpRI9xuSlw5OTai/2PYdZI2Va9+QzzBf99m
            VFyigfFdfrqug6aKFhF0lsujzlFfPfmXBbDRiTFX+4SkBeV71uuy7rOUI/jRiitEA0QrKqs0e/pV
            \+C8PoaariisK96Mtt7A=
        </ds:SignatureValue>
        <ds:KeyInfo>
            <ds:KeyValue>
                <ds:RSAKeyValue>
                    <ds:Modulus>
                   suGIyhVTbFvDwZdx8Av62zmP+aGOlsBN8WUE3eEEcDtOIZgO78SImMQGwB2C0eIVMhiLRzVPqoW1
                   dCPAveTm653zHOmubaps1fY0lLJDSZbTbhjeYhoQmmaBro/tDpVw5lKJwspqVnMuRK19ju2dxpKw
                        lYGGtrP5VQv00dfNPbs=
                    </ds:Modulus>
                    <ds:Exponent>AQAB</ds:Exponent>
                </ds:RSAKeyValue>
            </ds:KeyValue>
        </ds:KeyInfo>
    </ds:Signature>
</saml2:Assertion>
```

During the signature validation for this SAML assertion, the authenticator (in this case a Service Provider Authenticator) will try to find a **ValidationAlias** element with the value **idp.example.com** for its **Key** attribute. This alias references a certificate in your Java KeyStore that will be used to check the signature validity.

Usually, Java KeyStores would contain a key pair (public and private keys) to be used for signing and validating messages for an specific server and the trusted public keys to be used to validate messages received from others servers.

11.5.7. SAML2 Handlers

11.5.7.1. Introduction

When using PicketLink SAML Support, both IDP and SP need to be configured with *Handlers* . This handlers help the IDP and SP Authenticators to process SAML requests and responses.

The handlers are basically an implementation of the Chain of Responsibility pattern (Gof). Each handler provides a specific logic about how to process SAML requests and responses.

11.5.7.2. Configuring Handlers

The handlers are configures inside the picketlink.xml file. Here is how it looks like:

```
<Handlers xmlns="urn:picketlink:identity-federation:handler:config:2.1">
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2IssuerTrustHandler" />
   <Handler class="org.picketlink.identity.federation.web.handlers.saml2.SAML2LogOutHandler" />
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2AuthenticationHandler" />
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.RolesGenerationHandler" />
</Handlers>
```

11.5.7.2.1. Handlers Element

This element defines a list of Handler elements.

Name	Description	Value
ChainClass	Defines the name of a class that implements the *org.picketlink. identity.federation. core.saml. v2.interfaces. SAML2HandlerChain* interface.	Defaults to **org.picketlink. identity.federation. core.saml. v2.impl. DefaultSAML2HandlerChain.**

11.5.7.2.2. Handler Element

This element defines a specific Handler.

Name	Description
class	Defines the name of a class that implements the org.picketlink. identity.federation. core.saml. v2.interfaces. SAML2Handler interface.

11.5.7.3. Custom Handlers

PicketLink provide ways to you create your own handlers. Just create a class that implements the *org.picketlink.identity.federation.core.saml.v2.interfaces.SAML2Handler interface.*

Before create your own implementations, please take a look at the built-in handlers. They can help you a lot.

11.5.7.4. Built-in Handlers

PicketLink as part of the SAMLv2 support has a number of handlers that need to be configured.

The Handlers are:

1. Section 11.5.7.7, “SAML2AuthenticationHandler”
2. Section 11.5.7.6, “SAML2AttributeHandler”
3. Section 11.5.7.5, “RolesGenerationHandler”
4. Section 11.5.7.9, “SAML2IssuerTrustHandler”
5. SAML2LogOutHandler

11.5.7.5. RolesGenerationHandler

11.5.7.5.1. Objective

Handler dealing with attributes for SAML2

11.5.7.5.2. Fully Qualified Name

org.picketlink.identity.federation.web.handlers.saml2.RolesGenerationHandler

11.5.7.5.3. Configuration

Should be configured in WEB-INF/picketlink-handlers.xml

11.5.7.5.3.1. Example:

Example 11.6. WEB-INF/picketlink-handlers.xml

```
<Handlers xmlns="urn:picketlink:identity-federation:handler:config:1.0">
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2IssuerTrustHandler"/>
  <Handler class="org.picketlink.identity.federation.web.handlers.saml2.SAML2LogOutHandler"/>
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2AuthenticationHandler"/>
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.RolesGenerationHandler"/>
</Handlers>
```

11.5.7.5.4. Configuration Parameters

#	Name	Type	Objective	Default Value	SP/IDP	Since Version
1	ATTRIBUTE_MANAGER	string	fqn of attribute	org.picketlink.identity.federation.	IDP	2.0

#	Name	Type	Objective	Default Value	SP/IDP	Since Version
			manager class	core.impl. EmptyAttributeManager		

11.5.7.5.4.1. Example:

Example 11.7. WEB-INF/picketlink-handlers.xml

```
<Handler class="org.picketlink.
                    identity.federation.
                    web.handlers.
                    saml2.RolesGenerationHandler">
<Option Key="ATTRIBUTE_MANAGER" Value="org.some.fun.class"/>
</Handler>
```

11.5.7.6. SAML2AttributeHandler

11.5.7.6.1. Objective

Handler dealing with attributes for SAML2. On the SP side, it converts IDP-returned Attributes and stores them under the user's HttpSession. On the IDP side, converts the given HttpSession attributes into SAML Response Attributes. SP-side code can retrieve the Attributes from a Map stored under the session key GeneralConstants.SESSION_ATTRIBUTE_MAP.

11.5.7.6.2. Fully Qualified Name

org.picketlink.identity.federation.web.handlers.saml2.SAML2AttributeHandler

11.5.7.6.3. Configuration

Should be configured in WEB-INF/picketlink-handlers.xml

11.5.7.6.3.1. Example:

Example 11.8. WEB-INF/picketlink-handlers.xml

```
<Handlers xmlns="urn:picketlink:identity-federation:handler:config:1.0">
                                                                        <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2IssuerTrustHandler"/>
  <Handler class="org.picketlink.identity.federation.web.handlers.saml2.SAML2LogOutHandler"/>
                                                                        <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2AuthenticationHandler"/>
                                                                        <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.RolesGenerationHandler"/>
</Handlers>
```

11.5.7.6.4. Configuration Parameters

#	Name	Type	Objective	Default Value	SP/IDP	Since Version
1	ATTRIBUTE_MANAGER	string	fqn of attribute manager class	org.picketlink.identity.federation.core.impl.EmptyAttributeManager	IDP	2.0
2	ATTRIBUTE_KEYS	String	a comma separated list of string values representing attributes to be sent		IDP	2.0
3	ATTRIBUTE_CHOOSE_FRIENDLY_NAME	boolean	set to true if you require attributes to be keyed by friendly name rather than default name.		SP	2.0

11.5.7.6.4.1. Example:

Example 11.9. WEB-INF/picketlink-handlers.xml

```
<Handler class="org.picketlink.
                    identity.federation.
                    web.handlers.
                    saml2.SAML2AttributeHandler">
<Option Key="ATTRIBUTE_CHOOSE_FRIENDLY_NAME" Value="true"/>
</Handler>
```

11.5.7.6.4.2.

11.5.7.6.4.3. Example:

11.5.7.6.4.4.

Map<String, List<Object>> sessionMap = (Map<String, List<Object>>) session .getAttribute(GeneralConstants.SESSION_ATTRIBUTE_MAP); assertNotNull(sessionMap);

List<Object> values = sessionMap.get("testKey"); assertEquals("hello", values.get(0));

11.5.7.6.4.5.

11.5.7.6.4.6. Additional References

- PicketLink IDP using LDAP Attributes [https://community.jboss.org/wiki/PicketLinkIDPUsingLDAPAttributes]

11.5.7.7. SAML2AuthenticationHandler

11.5.7.7.1. Objective

Handler handles the SAML request at the IDP and the SAML response at the SP.

11.5.7.7.2. Fully Qualified Name

org.picketlink.identity.federation.web.handlers.saml2.SAML2AuthenticationHandler

11.5.7.7.3. Configuration

Should be configured in WEB-INF/picketlink-handlers.xml

11.5.7.7.3.1. Example:

Example 11.10. WEB-INF/picketlink-handlers.xml

```
<Handlers xmlns="urn:picketlink:identity-federation:handler:config:1.0">
                                                                        <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2IssuerTrustHandler"/>
  <Handler class="org.picketlink.identity.federation.web.handlers.saml2.SAML2LogOutHandler"/>
                                                                        <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2AuthenticationHandler"/>
                                                                        <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.RolesGenerationHandler"/>
</Handlers>
```

11.5.7.7.4. Configuration Parameters

#	Name	Type	Objective	SP/IDP	Since Version
1	CLOCK_ SKEW_ MILIS	string	a long value in miliseconds to add a clock skew to assertion expiration validation at the Service provider	SP	2.0

#	Name	Type	Objective	SP/IDP	Since Version
2	DISABLE_ AUTHN_ STATEMENT	boolean	Setting a value will disable the generation of an AuthnStatement	IDP	2.0
3	DISABLE_ SENDING_ ROLES	boolean	Setting any value will disable the generation and return of roles to SP	IDP	2.0
4	DISABLE_ ROLE_ PICKING	boolean	Setting to true will disable picking IDP attribute statements	SP	2.0
5	ROLE_ KEY	String	a csv list of strings that represent the roles coming from IDP	SP	2.0
6	ASSERTION_ CONSUMER_ URL	String	the url to be used for assertionConsumerURL	SP	2.0
7	NAMEID_ FORMAT	String	Setting to a value will provide the nameid format to be sent to IDP	SP	2.0
8	ASSERTION_ SESSION_ ATTRIBUTE_ NAME	String	Specifies the name of the session attribute where the assertion will be stored. The assertion	SP	2.1.7

#	Name	Type	Objective	SP/IDP	Since Version
			is stored as a DOM Document. This option is useful when you need to obtain the user's assertion to propagate or validate it against the STS.		
9	AUTHN_CONTEXT_CLASSES	String	Specifies a single or a comma separated list of SAML Authentication Classes to be used when creating an AuthnRequest. The value can be a full qualified name (FQN) or an alias. For each standard class name there is an alias, as defined by the `org.picketlink.common.constants.SAMLAuthenticationConte`	SP	2.5.0
9	REQUESTED_AUTHN_CONTEXT_COMPARISON	String	Specifies the Comparison attribute of the RequestedAuthnContext.	SP	2.5.0

#	Name	Type	Objective	SP/IDP	Since Version
			This option should be used in conjunction with the AUTHN_CONTEXT_CLASSES option. Only the values defined by the specification are supported.		

11.5.7.7.4.1. Example:

Example 11.11. WEB-INF/picketlink-handlers.xml

```
<Handler class="org.picketlink.identity.
                    federation.web.
                    handlers.saml2.SAML2AuthenticationHandler">
<Option Key="DISABLE_ROLE_PICKING" Value="true"/>
</Handler>
```

11.5.7.7.4.2. NAMEID_FORMAT:

The **transient** and **persistent nameid-formats** are used to obfuscate the actual identity in order to make linking activities extremely difficult between different SPs being served by the same IDP. A transient policy only lasts for the duration of the login session, where a persistent policy will reuse the obfuscated identity across multiple login sessions.

The Value can either be one of the following "official" values or a vendor-specific value supported by the IDP. Any string value is passed through to the NameIDPolicy's Format attribute as-is in an AuthnRequest.

urn:oasis:names:tc:SAML:2.0:nameid-format: **transient** urn:oasis:names:tc:SAML:2.0:nameid-format: **persistent** urn:oasis:names:tc:SAML:1.1:nameid-format: **unspecified** urn:oasis:names:tc:SAML:1.1:nameid-format: **emailAddress** urn:oasis:names:tc:SAML:1.1:nameid-format: **X509SubjectName** urn:oasis:names:tc:SAML:1.1:nameid-format: **WindowsDomainQualifiedName** urn:oasis:names:tc:SAML:2.0:nameid-format: **kerberos** urn:oasis:names:tc:SAML:2.0:nameid-format: **entity**

11.5.7.8. SAML2EncryptionHandler

11.5.7.8.1. Objective

Handles SAML Assertions Encryption and Signature Generation. This handler uses the configuration provided in the KeyProvider to encrypt and sign SAML Assertions.

11.5.7.8.2. Fully Qualified Name

org.picketlink.identity.federation.web.handlers.saml2.SAML2EncryptionHandler

11.5.7.8.3. Restrictions

- This handler should be used only when configuring Identity Providers.
- For Service Providers, the decryption of SAML Assertion is already done by the authenticators.
- When using this handler, make sure that your service providers are also configured with the Section 11.5.7.11, “SAML2SignatureGenerationHandler” and the Section 11.5.7.12, “SAML2SignatureValidationHandler” handlers.
- *Do not use this handler with the __ Section 11.5.7.11, “SAML2SignatureGenerationHandler”* _ configured in the same chain. Otherwise SAML messages will be signed several times._

11.5.7.8.4. Configuration

Should be configured in WEB-INF/picketlink.xml:

11.5.7.8.4.1. Example:

11.5.7.8.4.2.

```
<Handlers xmlns="urn:picketlink:identity-federation:handler:config:2.1">
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2IssuerTrustHandler" />
   <Handler class="org.picketlink.identity.federation.web.handlers.saml2.SAML2LogOutHandler" />
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2AuthenticationHandler" />
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.RolesGenerationHandler" />
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2EncryptionHandler" />
                                                                                <Handler

>
</Handlers>
```

11.5.7.8.5. Configuration Parameters

#	Name	Type	Objective	Default Value	SP/IDP	Since Version

11.5.7.8.5.1.

11.5.7.9. SAML2IssuerTrustHandler

11.5.7.9.1. Objective

Handles Issuer trust.Trust decisions are based on the url of the issuer of the saml request/ response sent to the handler chain.

11.5.7.9.2. Fully Qualified Name

org.picketlink.identity.federation.web.handlers.saml2.SAML2IssuerTrustHandler

11.5.7.9.3. Configuration

Should be configured in WEB-INF/picketlink-handlers.xml

11.5.7.9.3.1. Example:

Example 11.12. WEB-INF/picketlink-handlers.xml

```
<Handlers xmlns="urn:picketlink:identity-federation:handler:config:1.0">
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2IssuerTrustHandler"/>
  <Handler class="org.picketlink.identity.federation.web.handlers.saml2.SAML2LogOutHandler"/>
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2AuthenticationHandler"/>
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.RolesGenerationHandler"/>
</Handlers>
```

11.5.7.9.4. Configuration Parameters

#	Name	Type	Objective	Default Value	SP/IDP	Since Version

11.5.7.9.4.1.

11.5.7.10. SAML2LogOutHandler.java

11.5.7.10.1. Objective

Handler for SAML2 Logout Profile.

11.5.7.10.2. Fully Qualified Name

org.picketlink.identity.federation.web.handlers.saml2.SAML2LogOutHandler.java

11.5.7.10.3. Configuration

Should be configured in WEB-INF/picketlink-handlers.xml

11.5.7.10.3.1. Example:

Example 11.13. WEB-INF/picketlink-handlers.xml

```
<Handlers xmlns="urn:picketlink:identity-federation:handler:config:1.0">
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2IssuerTrustHandler"/>
  <Handler class="org.picketlink.identity.federation.web.handlers.saml2.SAML2LogOutHandler"/>
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2AuthenticationHandler"/>
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.RolesGenerationHandler"/>
</Handlers>
```

11.5.7.10.4. Configuration Parameters

#	Name	Type	Objective	Default Value	SP/IDP	Since Version

11.5.7.10.4.1.

11.5.7.11. SAML2SignatureGenerationHandler

11.5.7.11.1. Objective

Handles SAML Signature Generation. This handler uses the configuration provided in the KeyProvider to sign SAML messages.

11.5.7.11.2. Fully Qualified Name

org.picketlink.identity.federation.web.handlers.saml2.SAML2SignatureGenerationHandler

11.5.7.11.3. Configuration

Should be configured in WEB-INF/picketlink.xml.

11.5.7.11.3.1. Example:

11.5.7.11.3.2.

```
<Handlers xmlns="urn:picketlink:identity-federation:handler:config:2.1">
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2IssuerTrustHandler" />
   <Handler class="org.picketlink.identity.federation.web.handlers.saml2.SAML2LogOutHandler" />
```

```
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2AuthenticationHandler" />
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.RolesGenerationHandler" />
                                                                                <Handler
class="org.picketlink.identity.federation.web.handlers.saml2.SAML2SignatureGenerationHandler"/
>
                                                                                <Handler


>
</Handlers>
```

11.5.7.11.4. Configuration Parameters

#	Name	Type	Objective	Default Value	SP/IDP	Since Version

11.5.7.11.4.1.

11.5.7.12. SAML2SignatureValidationHandler

11.5.7.12.1. Objective

Handles SAML Signature Validation. This handler uses the configuration provided in the KeyProvider to process signature validation.

11.5.7.12.2. Fully Qualified Name

org.picketlink.identity.federation.web.handlers.saml2.SAML2SignatureValidationHandler

11.5.7.12.3. Configuration

Should be configured in WEB-INF/picketlink.xml.

11.5.7.12.3.1. Example:

11.5.7.12.3.2.

```
<Handlers xmlns="urn:picketlink:identity-federation:handler:config:2.1">
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2IssuerTrustHandler" />
   <Handler class="org.picketlink.identity.federation.web.handlers.saml2.SAML2LogOutHandler" />
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.SAML2AuthenticationHandler" />
                                                                                <Handler
 class="org.picketlink.identity.federation.web.handlers.saml2.RolesGenerationHandler" />
                                                                                <Handler
class="org.picketlink.identity.federation.web.handlers.saml2.SAML2SignatureGenerationHandler"/
>
                                                                                <Handler


>
```

```
</Handlers>
```

11.5.7.12.4. Configuration Parameters

#	Name	Type	Objective	Default Value	SP/IDP	Since Version

11.5.7.12.4.1.

11.5.8. Single Logout

Table of Contents

Even though the SAML v2.0 specification has support for Global Logout, you have to use it very very wisely. Just remember that you need to keep the participants to a low number (say upto 5 participants with one IDP).

Global Logout : The user initiates GLO at one service provider which will log out the user at the IDP and all the service providers.

Local Logout : The user logs out of one service provider only. The session at the IDP and other service providers is intact.

11.5.8.1. Configuring the GLO

The service provider url should be appended with "?GLO=true"

Basically, in the service provider page, have a url that has the query parameter.

Assume, your service provider is http://localhost:8080/sales/, [http://localhost:8080/sales/,] then the url for the global log out would be http://localhost:8080/sales/?GLO=true

11.5.8.2. Configuring the LLO

The service provider url should be appended with "?LLO=true"

Basically, in the service provider page, have a url that has the query parameter.

Assume, your service provider is http://localhost:8080/sales/, [http://localhost:8080/sales/,] then the url for the local log out would be http://localhost:8080/sales/?LLO=true

When using LLO, you must be aware of some security implications. The user is only disconnect from the service provider from which he logged out, which means that the user's session in the identity provider and others service providers are still active. In other words, the user's SSO session is still active and he is still able to log in in any other service provider. We strongly recommend to always use the Single Logout Profile (GLO).

Important

In the case of LLO, the service provider invalidates the session and forwards to a default logout page (logout.jsp) .Custom logout page can be configured in picketlink.xml page. Please refer to Service Provider Configuration.

11.5.9. SAML2 Configuration Providers

Table of Contents

It is possible to use different Configuration Providers at the IDP and SP.

The configuration providers will then be the sole configuration leaders (instead of picketlink.xml)

11.5.9.1. Configuration providers at the IDP

11.5.9.1.1. IDPMetadataConfigurationProvider

Fully Qualified Name: org.picketlink.identity.federation.web.config.IDPMetadataConfigurationProvider

How does it work?

You will need to provide the metadata file inside idp-metadata.xml and put it in the IDP web application classpath. Put it in WEB-INF/classes directory.

11.5.9.2. Configuration Providers at the SP

11.5.9.2.1. SPPostMetadataConfigurationProvider

Fully Qualified Name: org.picketlink.identity.federation.web.config.SPPostMetadataConfigurationProvider

Binding Supported: SAML2/HTTP-POST

11.5.9.2.1.1. How does it work?

You will need to provide the metadata file inside sp-metadata.xml and put it in the SP web application classpath. Put it in WEB-INF/classes directory.

Remember, in the case of SP, the metadata file should have a IDPSSODescriptor as well as a SPSSODescriptor.

11.5.9.2.2. SPRedirectMetadataConfigurationProvider

Fully Qualified Name: org.picketlink.identity.federation.web.config.SPRedirectMetadataConfigurationProvider

Binding Supported: SAML2/HTTP-Redirect

11.5.9.2.2.1. How does it work?

You will need to provide the metadata file inside sp-metadata.xml and put it in the SP web application classpath. Put it in WEB-INF/classes directory.

Remember, in the case of SP, the metadata file should have a IDPSSODescriptor as well as a SPSSODescriptor.

11.5.9.2.3. What about Key Information and other configuration that comes via picketlink-idfed.xml?

Both the IDP and SP applications when provided with the saml configuration provider will be given a parsed representation of the WEB-INF/picketlink-idfed.xml, which implies that the IDPType and SPType coming out finally will be a merger of the configuration provider and the settings from picketlink-idfed.xml

11.5.10. Metadata Support

Table of Contents

11.5.10.1. Introduction

It is possible to use different Configuration Providers at the IDP and SP. The configuration providers will then be the sole configuration leaders (instead of picketlink.xml) or provide additional configuration.

PicketLink SAML Metadata Support is provided and configured by the following configuration providers implementations:

Name	Description	Provider Type
org.picketlink.identity.federation.web.config.IDPMetadataConfigurationProvider	For Identity Providers	IDP
org.picketlink.identity.federation.web.config.SPPostMetadataConfigurationProvider	For Service Providers using HTTP-POST Binding	SP
org.picketlink.identity.federation.web.config.SPRedirectMetadataConfigurationProvider	For Service Providers using HTTP-REDIRECT Binding	SP

These providers allows you to define some additional configuration to your IDP or SP using a SAML Metadata XML Schema instance, merging them with the ones provided in your *WEB-INF/picketlink.xml* .

11.5.10.2. Configuration

To configure the SAML Metadata Configuration Providers you need to follow these steps:

- Define the PicketLink Authenticator (SP or IDP valves) and provide the configuration provider class name as an attribute
- Depending if you're configuring an IDP or SP, provide a metadata file and put it on the classpath:
- For Identity Providers : WEB-INF/classes/idp-metadata.xml
- For Service Providers : WEB-INF/classes/sp-metadata.xml

11.5.10.2.1. Configuring the PicketLink Authenticator

To configure one of the provided SAML Metadata configuration providers you just need to configure the PicketLink Authenticator with the **configProvider** parameter/attribute.

For Identity Providers you should have a configuration as follow:

```
<jboss-web>
  <security-domain>idp</security-domain>
  <context-root>idp-metadata</context-root>
  <valve>
     <class-name>org.picketlink.identity.federation.bindings.tomcat.idp.IDPWebBrowserSSOValve</
class-name>
      <param>
       <param-name>configProvider</param-name>
                                                                                <param-
value>org.picketlink.identity.federation.web.config.IDPMetadataConfigurationProvider</param-
value>
      </param>
    </valve>
</jboss-web>
```

For Service Providers you should have a configuration as follow:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-web>
  <security-domain>sp</security-domain>
  <context-root>sales-metadata</context-root>
    <valve>
                                                                                <class-
name>org.picketlink.identity.federation.bindings.tomcat.sp.ServiceProviderAuthenticator</
class-name>
      <param>
```

```
 <param-name>configProvider</param-name>
                                                                                <param-
value>org.picketlink.identity.federation.web.config.SPPostMetadataConfigurationProvider</
param-value>
      </param>
    </valve>
</jboss-web>
```

11.5.10.2.2. What about Key Information and other configuration that comes via picketlink-idfed.xml?

Both the IDP and SP applications when provided with the saml configuration provider will be given a parsed representation of the WEB-INF/picketlink.xml, which implies that the IDPType and SPType coming out finally will be a merger of the configuration provider and the settings from picketlink.xml

11.5.10.3. Examples

The PicketLink Quickstarts [https://docs.jboss.org/author/pages/viewpage.action?pageId=23986289] provide some examples for the SAML Metadata Support. Please check the following provided quickstarts:

- https://github.com/picketlink/picketlink-quickstarts/tree/master/saml/idp-metadata
- https://github.com/picketlink/picketlink-quickstarts/tree/master/saml/sales-metadata

11.5.11. Token Registry

11.5.11.1. Introduction

PicketLink supports the concept of *Token Registry* to store tokens using any store such databases, filesystem or memory.

They are useful for auditing and to track the tokens that were issued or revocated by the Identity Provider or the Security Token Service.

> **Tip**
>
> When running PicketLink in a clustered environment, consider using Token Registries with databases. That way changes to the token table are visible to all nodes.

11.5.11.2. of-box Token Registries

The table bellow shows all implementations provided by PicketLink:

Name	Description	Version
org.picketlink.identity.federation.core.sts.registry.DefaultTokenRegistry	In-memory based registry. *Used by default if no configuration is provided*	2.x.x
org.picketlink.identity.federation.core.sts.registry.FileBasedTokenRegistry	Filesystem based registry	2.x.x
org.picketlink.identity.federation.core.sts.registry.JPABasedTokenRegistry	Database/JPA based registry	2.1.3

11.5.11.3. Configuration

Token Registries are configured through the **PicketLinkSTS** (Security Token Service configuration) element in the **WEB-INF/picketlink.xml** file:

> **Tip**
>
> Read the documentation for more information about the **PicketLinkSTS** element and the **Section 11.5.3.6, "Security Token Service Configuration" .**

```
<PicketLinkSTS    xmlns="urn:picketlink:identity-federation:config:1.0"    TokenTimeout="5000"
 ClockSkew="0">
        <TokenProviders>
  <TokenProvider

 ProviderClass="org.picketlink.identity.federation.core.saml.v2.providers.SAML20AssertionTokenProvider"
   TokenType="urn:oasis:names:tc:SAML:2.0:assertion"
   TokenElement="Assertion" TokenElementNS="urn:oasis:names:tc:SAML:2.0:assertion">
                                              <Property               Key="TokenRegistry"
 Value="org.picketlink.identity.federation.core.sts.registry.JPABasedTokenRegistry" />
  </TokenProvider>
 </TokenProviders>
</PicketLinkSTS>
```

The example above uses a SAML v2 Token Provider configured with the *org.picketlink.identity.federation.core.sts.registry.JPABasedTokenRegistry* implementation. This is done by the **TokenRegistry** property.

11.5.11.3.1. org.picketlink.identity.federation.core.sts.registry.FileBasedTokenRegistry

```
<TokenProvider
 ProviderClass="org.picketlink.identity.federation.core.saml.v2.providers.SAML20AssertionTokenProvider"
 TokenType="urn:oasis:names:tc:SAML:2.0:assertion"
```

```
 TokenElement="Assertion" TokenElementNS="urn:oasis:names:tc:SAML:2.0:assertion">
    <Property Key="TokenRegistry" Value="FILE" />
    <Property Key="TokenRegistryFile" Value="/some/dir/token.registry" />
</TokenProvider>
```

Use the **TokenRegistryFile** to specify a file where the tokens should be persisted.

11.5.11.3.2. org.picketlink.identity.federation.core.sts.registry.JPABasedTokenRegistry

```
<TokenProvider
 ProviderClass="org.picketlink.identity.federation.core.saml.v2.providers.SAML20AssertionTokenProvider"
 TokenType="urn:oasis:names:tc:SAML:2.0:assertion"
 TokenElement="Assertion" TokenElementNS="urn:oasis:names:tc:SAML:2.0:assertion">
                                                    <Property           Key="TokenRegistry"
 Value="org.picketlink.identity.federation.core.sts.registry.JPABasedTokenRegistry" />
</TokenProvider>
```

This implementation requires that you have a valid JPA Persistence Unit named **picketlink-sts.**

11.5.11.4. Custom Token Registry

If none of the built-in implementations are useful for you, PicketLink allows you to create your own implementation. To do that, just create a class that implements the **org.picketlink.identity.federation.core.sts.registry.SecurityTokenRegistry** interface.

> **Tip**
>
> We recommend that you take a look first at one of the provided implementation before building your own.

Bellow is an skeleton for a custom Token Registry implementation:

```
public class CustomSecurityTokenRegistry implements SecurityTokenRegistry {

    @Override
    public void addToken(String tokenID, Object token) throws IOException {
        // TODO: logic to add a token to the registry

    }

    @Override
    public void removeToken(String tokenID) throws IOException {
        // TODO: logic to remove a token to the registry
    }

    @Override
    public Object getToken(String tokenID) {
        // TODO: logic to get a token from the registry
```

```
        return null;
    }

}
```

11.5.12. Standalone vs JBossAS Distribution

PicketLink has SAMLv2 support for both JBossAS and a regular servlet container. The JBoss AS version contains deeper integration with the web container security such that you can make use of api such as request.getUserPrincipal() etc. Plus you can configure your favorite JAAS login module for authentication at the IDP side.

So, choose the JBossAS version of PicketLink [https://docs.jboss.org/author/display/PLINK/Tomcat+Authenticators+%28Tomcat%2CJBossAS%29] . If you do not run on JBoss AS or Apache Tomcat, then choose the standalone version .

11.5.13. Standalone Web Applications(All Servlet Containers)

If your IDP or SP applications are not running on JBoss Application Server or Apache Tomcat, then you can use the standalone mode of PicketLink.

11.5.13.1. Service Provider Configuration

In your web.xml, configure a Section 11.5.13.6, “SPFilter” as shown below as an example:

Example 11.14. web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_2_5.xsd"
   version="2.5">

   <description>Sales Standalone Application</description>

   <filter>
     <description>
           The SP Filter intersects all requests at the SP and sees if there is a need to
 contact the IDP.
     </description>
     <filter-name>SPFilter</filter-name>
     <filter-class>org.picketlink.identity.federation.web.filters.SPFilter</filter-class>
     <init-param>
       <param-name>ROLES</param-name>
       <param-value>sales,manager</param-value>
     </init-param>
   </filter>

   <filter-mapping>
     <filter-name>SPFilter</filter-name>
```

```
     <url-pattern>/*</url-pattern>
   </filter-mapping>
</web-app>
```

After the SAML workflow is completed, the user principal is available in the http session at "picketlink.principal".

Something like,

import org.picketlink.identity.federation.web.constants.GeneralConstants;

Principal userPrincipal = (Principal) session.getAttribute(GeneralConstants.PRINCIPAL_ID);

11.5.13.2. IDP Configuration

For an IDP web application to be SAML enabled on any Servlet Container, you will have to add listeners and servlets as shown in the web.xml below:

Part of the **idp-standalone.war**

Example 11.15. web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_2_5.xsd"
   version="2.5">

  <display-name>Standalone IDP</display-name>
  <description>
    IDP Standalone Application
  </description>

  <!-- Listeners -->
  <listener>
    <listener-class>org.picketlink.identity.federation.web.core.IdentityServer</listener-class>
  </listener>


  <!-- Create the servlet -->
  <servlet>
    <servlet-name>IDPLoginServlet</servlet-name>
      <servlet-class>org.picketlink.identity.federation.web.servlets.IDPLoginServlet</servlet-
class>
  </servlet>
  <servlet>
    <servlet-name>IDPServlet</servlet-name>
    <servlet-class>org.picketlink.identity.federation.web.servlets.IDPServlet</servlet-class>
  </servlet>

  <servlet-mapping>
```

```
    <servlet-name>IDPLoginServlet</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>

  <servlet-mapping>
    <servlet-name>IDPServlet</servlet-name>
    <url-pattern>/IDPServlet</url-pattern>
  </servlet-mapping>

</web-app>
```

A jsp for login would be:

Example 11.16. jsp/login.jsp

```
<html><head><title>Login Page</title></head>
<body>
<font size='5' color='blue'>Please Login</font><hr>

<form action='<%=application.getContextPath()%>/' method='post'>
<table>
 <tr><td>Name:</td>
   <td><input type='text' name='JBID_USERNAME'></td></tr>
 <tr><td>Password:</td>
   <td><input type='password' name='JBID_PASSWORD' size='8'></td>
 </tr>
</table>
<br>
  <input type='submit' value='login'>
</form></body>
 </html>
```

The jsp for error would be:

Example 11.17. jsp/error.jsp

```
<html> <head> <title>Error!</title></head>
<body>

<font size='4' color='red'>
  The username and password you supplied are not valid.
</p>
Click <a href='<%= response.encodeURL("login.jsp") %>'>here</a>
to retry login

</body>
</form>
</html>
```

11.5.13.3. Other References

1. http://community.jboss.org/wiki/PicketLinkSAMLSSOForWebContainers

11.5.13.4. IDPLoginServlet

IDPLoginServlet provides the login capabilities for IDP applications running on any servlet container.

11.5.13.4.1. Initialization Parameters

#	Name	Type	Objective	Default	Since
1	loginClass	String	fqn of an implementation of the ILoginHandler interface. Provides the authentication/ authorization.	org.picketlink.identity.federation.web.handlers.DefaultLoginHandler	2.0

11.5.13.4.2. Configuration

The IDP application needs to contain **/jsp/login.jsp** . The jsp file needs to have a form with two text fields namely: JBID_USERNAME and JBID_PASSWORD to indicate username and password.

On successful authentication, this servlet redirects to the IDPServlet.

11.5.13.5. IDPServlet

IDPServlet supports the SAMLv2 HTTP/POST binding for an IDP running on any servlet container.

11.5.13.5.1. Initialization Parameters

#	Name	Type	Objective	Default	Since
1	CONFIG_PROVIDER	String	optional - fqn of an implementation of the SAMLConfigurationProvider interface.		2.0
2	SIGN_OUTGOING_MESSAGES	boolean	optional - whether the IDP should	true	2.0

#	Name	Type	Objective	Default	Since
			Standalone Web Applications(All Servlet Containers) sign outgoing messages		
3	ROLE_ GENERATOR	String	optional - fqn of a RoleGenerator	org.picketlink. identity.federation. web.roles. DefaultRoleGenerator	2.0
4	ATTRIBUTE_ KEYS	String	optional - comma separated list of keys for attributes that need to be sent		2.0
5	IDENTITY_ PARTICIPANT_ STACK	String	optional - fqn of a custom IdentityParticipantStack implementation		2.0

11.5.13.5.2. Configuration

The Section 11.5.13.4, “IDPLoginServlet” that is configured in the web application authenticates the user. The IDPServlet then sends back the SAML response message with the SAML assertion back to the Service Provider(SP).

11.5.13.6. SPFilter

SPFilter is the filter that service provider applications need to have to provide HTTP/POST binding of the SAMLv2 specification for web applications running on any servlet container.

11.5.13.6.1. Initialization Parameters

#	Name	Type	Objective	Default	Since
1	IGNORE_ SIGNATURES	boolean	optional - should the SP ignore signatures	false	2.0
2	SAML_ HANDLER_ CHAIN_ CLASS	String	optional - fqn of custom SAMLHandlerChain interface		2.0
3	ROLE_ VALIDATOR	String	optional - fqn of a	org.picketlink. identity.federation.	2.0

#	Name	Type	Objective	Default	Since
			IRoleValidator interface	web.roles. DefaultRoleValidator	
4	ROLES	String	optional - comma separated list of roles that the sp will take		2.0
5	LOGOUT_ PAGE	String	optional - a logout page	/logout.jsp	2.0

11.6. SAML v1.1

11.6.1. SAML v1.1

Please refer to the wikipedia page [http://en.wikipedia.org/wiki/SAML_1.1] for more information.

11.6.2. PicketLink SAML v1.1 Support

Please read it at http://community.jboss.org/wiki/PicketLinkSAMLV11Support

11.7. Trust

11.7.1. Security Token Server (STS)

11.7.1.1. Introduction

The WS-Trust specification defines extensions that build on WS-Security to provide a framework for requesting and issuing security tokens. Particularly, WS-Trust defines the concept of a security token service (STS), a service that can issue, cancel, renew and validate security tokens, and specifies the format of security token request and response messages.

> **Tip**
>
> Please look at the PicketLink Quickstarts [https://docs.jboss.org/author/pages/viewpage.action?pageId=23986289] for the PicketLink Identity Provider web application. The quickstarts are useful resources where you can get configuration files.

11.7.1.2. References

PicketLink STS Dashboard [http://community.jboss.org/wiki/PicketLinkSTSDashboard]

11.7.1.3. PicketLink JBoss Web Services Handlers

Page to list all the JBoss Web Services handlers that are part of the PicketLink project.

1. SAML2Handler
2. BinaryTokenHandler
3. WSAuthenticationHandler
4. WSAuthorizationHandler

11.7.1.3.1. BinaryTokenHandler

11.7.1.3.1.1. Fully Qualified Name

org.picketlink.trust.jbossws.handler.BinaryTokenHandler

11.7.1.3.1.2. Objective

A JBoss Web Services Handler that is stack agnostic that can be added on the client side to either pick a http header or cookie, that contains a binary token.

11.7.1.3.1.3. Author

Anil Saldhana

11.7.1.3.1.4. Settings

Configuration:

System Properties:

- binary.http.header: http header name
- binary.http.cookie: http cookie name
- binary.http.encodingType: attribute value of the EncodingType attribute
- binary.http.valueType: attribute value of the ValueType attribute
- binary.http.valueType.namespace: namespace for the ValueType attribute
- binary.http.valueType.prefix: namespace for the ValueType attribute
- binary.http.cleanToken: true or false dependending on whether the binary token has to be cleaned

Setters:

Please see the Security Token Server (STS) see also section.See Also:setHttpHeaderName(String)setHttpCookieName(String)setEncodingType(String)setValueType(String)setValue

11.7.1.3.1.5. Test Case

http://anonsvn.jboss.org/repos/picketlink/integration-tests/trunk/picketlink-trust-tests/src/test/java/org/picketlink/test/trust/tests/STSWSBinaryTokenTestCase.java

11.7.1.3.2. SAML2Handler

11.7.1.3.2.1. Full Name:

org.picketlink.trust.jbossws.handler.SAML2Handler

11.7.1.3.2.2. Authors:

- Marcus Moyses
- Anil Saldhana

11.7.1.3.2.3. Objective:

This is a JBossWS handler (stack agnostic) that supports the SAML token profile of the Oasis Web Services Security (WSS) standard.

It can be configured both on the client side and the server side. The configuration is shown below both the client(outbound) as well as server(inbound).

11.7.1.3.2.3.1. Outbound:

This is the behavior when the handler is configured on the client side.

The client side usage is shown in the following client class. If you need to use an XML file to specify the handler on the client side, then please look in the references section below.

Example 11.18. STSWSClientTestCase.java

```
package org.picketlink.test.trust.tests;

import java.net.URL;
import java.util.List;

import javax.xml.namespace.QName;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.Service;
import javax.xml.ws.handler.Handler;

import org.junit.Test;
import org.picketlink.identity.federation.api.wstrust.WSTrustClient;
import org.picketlink.identity.federation.api.wstrust.WSTrustClient.SecurityInfo;
import org.picketlink.identity.federation.core.wstrust.WSTrustException;
import org.picketlink.identity.federation.core.wstrust.plugins.saml.SAMLUtil;
```

```
import org.picketlink.test.trust.ws.WSTest;
import org.picketlink.trust.jbossws.SAML2Constants;
import org.picketlink.trust.jbossws.handler.SAML2Handler;
import org.w3c.dom.Element;

/**
 * A Simple WS Test for the SAML Profile of WSS
 * @author Marcus Moyses
 * @author Anil Saldhana
 */
public class STSWSClientTestCase
{
   private static String username = "UserA";
   private static String password = "PassA";

   @SuppressWarnings("rawtypes")
   @Test
   public void testWSInteraction() throws Exception {
      WSTrustClient client = new WSTrustClient("PicketLinkSTS", "PicketLinkSTSPort",
            "http://localhost:8080/picketlink-sts/PicketLinkSTS",
            new SecurityInfo(username, password));
    Element assertion = null;
    try {
        System.out.println("Invoking token service to get SAML assertion for " + username);
        assertion = client.issueToken(SAMLUtil.SAML2_TOKEN_TYPE);
        System.out.println("SAML assertion for " + username + " successfully obtained!");
    } catch (WSTrustException wse) {
        System.out.println("Unable to issue assertion: " + wse.getMessage());
        wse.printStackTrace();
        System.exit(1);
    }

    URL wsdl = new URL("http://localhost:8080/picketlink-wstest-tests/WSTestBean?wsdl");
    QName serviceName = new QName("http://ws.trust.test.picketlink.org/", "WSTestBeanService");
    Service service = Service.create(wsdl, serviceName);
          WSTest   port   =   service.getPort(new   QName("http://ws.trust.test.picketlink.org/",
 "WSTestBeanPort"), WSTest.class);
    BindingProvider bp = (BindingProvider)port;
    bp.getRequestContext().put(SAML2Constants.SAML2_ASSERTION_PROPERTY, assertion);
    List<Handler> handlers = bp.getBinding().getHandlerChain();
    handlers.add(new SAML2Handler());
    bp.getBinding().setHandlerChain(handlers);

    port.echo("Test");
   }
}
```

Note: the SAML2Handler is instantiated and added to the handler list that is obtained from the BindingProvider binding.

There are two ways by which the SAML2Handler picks the SAML2 Assertion to send via the SOAP message.

- The Client can push the SAML2 Assertion into the SOAP MessageContext under the key " **org.picketlink.trust.saml.assertion** ". In the example code above, look in the call bindingProvider.getRequestContext().put(xxxxx)

- The SAML2 Assertion is available as part of the JAAS subject on the security context. This can happen if there has been a JAAS interaction with the usage of PicketLink STS login modules.

11.7.1.3.2.3.2. Inbound:

This is the behavior when the handler is configured on the server side.

The server side setting is as follows:

Example 11.19. handlers.xml

```
<?xml version="1.0" encoding="UTF-8"?>


<handler-chains xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ns1="http://org.jboss.ws/jaxws/samples/logicalhandler"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee javaee_web_services_1_2.xsd">



  <handler-chain>
    <handler>
      <handler-name>SAML2Handler</handler-name>
      <handler-class>org.picketlink.trust.jbossws.handler.SAML2Handler</handler-class>
    </handler>
  </handler-chain>


</handler-chains>
```

The SAML2Handler looks for a SAML2 Assertion on the SOAP message. If it is available then it constructs a SamlCredential object with the assertion and then sets it on the SecurityContext for the JAAS layer to authenticate the call.

11.7.1.3.2.4. References

JBossWS User Guide on Handlers [http://community.jboss.org/wiki/JBossWS-UserGuide#Handler_Framework]

JBossWS JAXWS Client Configuration [http://community.jboss.org/wiki/JBossWS-JAX-WSClientConfiguration]

11.7.1.3.3. WSAuthenticationHandler

11.7.1.3.3.1. FQN:

org.picketlink.trust.jbossws.handler.WSAuthenticationHandler

11.7.1.3.3.2. Objective:

Perform authentication for POJO based webservices.

11.7.1.3.3.3. Example Usage:

Assume that you have a POJO.

```
package org.picketlink.test.trust.ws;

import javax.jws.HandlerChain;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

/**
 * POJO that is exposed as WS
 * @author Anil Saldhana
 */
@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
@HandlerChain(file="authorize-handlers.xml")
public class POJOBean
{
   @WebMethod
   public void echo(String echo)
   {
      System.out.println(echo);
   }

   @WebMethod
   public void echoUnchecked(String echo)
   {
      System.out.println(echo);
   }
}
```

Note the use of the @HandlerChain annotation that defines the handler xml.

The handler xml is authorize-handlers.xml.

```
<?xml version="1.0" encoding="UTF-8"?>


<handler-chains xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee javaee_web_services_1_2.xsd">

  <handler-chain>


    <handler>
      <handler-name>WSAuthorizationHandler</handler-name>
     <handler-class>org.picketlink.trust.jbossws.handler.WSAuthorizationHandler</handler-class>
    </handler>

    <handler>
      <handler-name>WSAuthenticationHandler</handler-name>
```

```
          <handler-class>org.picketlink.trust.jbossws.handler.WSAuthenticationHandler</handler-
class>
    </handler>

    <handler>
      <handler-name>SAML2Handler</handler-name>
      <handler-class>org.picketlink.trust.jbossws.handler.SAML2Handler</handler-class>
    </handler>


  </handler-chain>


</handler-chains>
```

Warning

Note : The order of execution of the handlers is SAML2Handler, WSAuthenticationHandler and WSAuthorizationHandler. These need to be defined in reverse order in the xml.

Since we intend to expose a POJO as a webservice, we need to package in a web archive (war).

The web.xml is:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://java.sun.com/xml/ns/javaee/web-
app_2_5.xsd"
 version="2.5">

 <servlet>
  <display-name>POJO Web Service</display-name>
  <servlet-name>POJOBeanService</servlet-name>
  <servlet-class>org.picketlink.test.trust.ws.POJOBean</servlet-class>
  <load-on-startup>1</load-on-startup>
 </servlet>
 <servlet-mapping>
  <servlet-name>POJOBeanService</servlet-name>
  <url-pattern>/POJOBeanService</url-pattern>
 </servlet-mapping>
</web-app>
```

Warning

Please do not define any <security-constraint> in the web.xml

The jboss-web.xml is:

```
<jboss-web>
  <security-domain>sts</security-domain>
</jboss-web>
```

The jboss-wsse.xml is

```
<jboss-ws-security xmlns="http://www.jboss.com/ws-security/config"
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xsi:schemaLocation="http://www.jboss.com/ws-security/config
                   http://www.jboss.com/ws-security/schema/jboss-ws-security_1_0.xsd">

  <port name="POJOBeanPort">
    <operation name="{http://ws.trust.test.picketlink.org/}echoUnchecked">
      <config>
        <authorize>
          <unchecked/>
        </authorize>
      </config>
    </operation>

    <operation name="{http://ws.trust.test.picketlink.org/}echo">
      <config>
        <authorize>
          <role>JBossAdmin</role>
        </authorize>
      </config>
    </operation>
  </port>



</jboss-ws-security>
```

As you can see, there are two operations defined on the POJO web services and each of these operations require different access control. The echoUnchecked() method allows free access to any authenticated user whereas the echo() method requires the caller to have "JBossAdmin" role.

The war should look as:

```
anil@localhost:~/picketlink/picketlink/integration-tests/trunk/picketlink-trust-tests$ jar tvf
 target/pojo-test.war
     0 Mon Apr 11 19:48:32 CDT 2011 META-INF/
   123 Mon Apr 11 19:48:30 CDT 2011 META-INF/MANIFEST.MF
     0 Mon Apr 11 19:48:30 CDT 2011 WEB-INF/
     0 Mon Apr 11 19:48:30 CDT 2011 WEB-INF/classes/
     0 Mon Apr 11 19:48:30 CDT 2011 WEB-INF/classes/org/
     0 Mon Apr 11 19:48:30 CDT 2011 WEB-INF/classes/org/picketlink/
     0 Mon Apr 11 19:48:30 CDT 2011 WEB-INF/classes/org/picketlink/test/
     0 Mon Apr 11 19:48:30 CDT 2011 WEB-INF/classes/org/picketlink/test/trust/
     0 Mon Apr 11 19:48:30 CDT 2011 WEB-INF/classes/org/picketlink/test/trust/ws/
     0 Mon Apr 11 19:48:30 CDT 2011 WEB-INF/lib/
   858 Mon Apr 11 19:48:26 CDT 2011 WEB-INF/classes/authorize-handlers.xml
```

```
  1021 Mon Apr 11 19:48:28 CDT 2011 WEB-INF/classes/org/picketlink/test/trust/ws/POJOBean.class
    65 Mon Apr 11 12:00:32 CDT 2011 WEB-INF/jboss-web.xml
   770 Mon Apr 11 17:44:16 CDT 2011 WEB-INF/jboss-wsse.xml
   598 Mon Apr 11 16:25:46 CDT 2011 WEB-INF/web.xml
     0 Mon Apr 11 19:48:32 CDT 2011 META-INF/maven/
     0 Mon Apr 11 19:48:32 CDT 2011 META-INF/maven/org.picketlink/
     0 Mon Apr 11 19:48:32 CDT 2011 META-INF/maven/org.picketlink/picketlink-integration-trust-
tests/
  7918 Mon Apr 11 18:56:16 CDT 2011 META-INF/maven/org.picketlink/picketlink-integration-trust-
tests/pom.xml
   142 Mon Apr 11 19:48:30 CDT 2011 META-INF/maven/org.picketlink/picketlink-integration-trust-
tests/pom.properties
anil@localhost:~/picketlink/picketlink/integration-tests/trunk/picketlink-trust-tests
```

The Test Case is something like:

```
 package org.picketlink.test.trust.tests;

import java.net.URL;
import java.util.List;

import javax.xml.namespace.QName;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.Service;
import javax.xml.ws.handler.Handler;

import org.junit.Test;
import org.picketlink.identity.federation.api.wstrust.WSTrustClient;
import org.picketlink.identity.federation.api.wstrust.WSTrustClient.SecurityInfo;
import org.picketlink.identity.federation.core.wstrust.WSTrustException;
import org.picketlink.identity.federation.core.wstrust.plugins.saml.SAMLUtil;
import org.picketlink.test.trust.ws.WSTest;
import org.picketlink.trust.jbossws.SAML2Constants;
import org.picketlink.trust.jbossws.handler.SAML2Handler;
import org.w3c.dom.Element;

/**
 * A Simple WS Test for POJO WS Authorization using PicketLink
 * @author Anil Saldhana
 * @since Oct 3, 2010
 */
public class POJOWSAuthorizationTestCase
{
   private static String username = "UserA";
   private static String password = "PassA";

   @SuppressWarnings("rawtypes")
   @Test
   public void testWSInteraction() throws Exception
   {
      // Step 1:  Get a SAML2 Assertion Token from the STS
      WSTrustClient client = new WSTrustClient("PicketLinkSTS", "PicketLinkSTSPort",
            "http://localhost:8080/picketlink-sts/PicketLinkSTS",
            new SecurityInfo(username, password));
      Element assertion = null;
      try {
```

```
         System.out.println("Invoking token service to get SAML assertion for " + username);
         assertion = client.issueToken(SAMLUtil.SAML2_TOKEN_TYPE);
         System.out.println("SAML assertion for " + username + " successfully obtained!");
      } catch (WSTrustException wse) {
         System.out.println("Unable to issue assertion: " + wse.getMessage());
         wse.printStackTrace();
         System.exit(1);
      }

        // Step 2: Stuff the Assertion on the SOAP message context and add the SAML2Handler
 to client side handlers
      URL wsdl = new URL("http://localhost:8080/pojo-test/POJOBeanService?wsdl");
      QName serviceName = new QName("http://ws.trust.test.picketlink.org/", "POJOBeanService");
      Service service = Service.create(wsdl, serviceName);
           WSTest  port  =  service.getPort(new  QName("http://ws.trust.test.picketlink.org/",
 "POJOBeanPort"), WSTest.class);
      BindingProvider bp = (BindingProvider)port;
      bp.getRequestContext().put(SAML2Constants.SAML2_ASSERTION_PROPERTY, assertion);
      List<Handler> handlers = bp.getBinding().getHandlerChain();
      handlers.add(new SAML2Handler());
      bp.getBinding().setHandlerChain(handlers);

      //Step 3: Access the WS. Exceptions will be thrown anyway.
      port.echo("Test");
   }
}
```

11.7.1.3.4. WSAuthorizationHandler

11.7.1.3.4.1. FQN:

org.picketlink.trust.jbossws.handler.WSAuthorizationHandler

11.7.1.3.4.2. Objective:

Provide authorization capabilities to POJO based web services.

11.7.1.3.4.3. Example Usage:

Please refer to the documentation on WSAuthenticationHandler.

> **Important**
>
> The example is in WSAuthenticationHandler [https://docs.jboss.org/author/display/PLINK/WSAuthenticationHandler] section.

11.7.1.4. Protecting EJB Endpoints

11.7.1.4.1. Introduction

PicketLink provides ways to protect your EJB endpoints using a SAML Security Token Service. This means that you can apply some security to your EJBs where only users with a valid SAML assertion can invoke to them.

This scenario is very common if you are looking for:

1. Leverage your Single Sign-On infrastructure to your service layer (EJBs, Web Services, etc)
2. Integrate your SAML Service Providers with your services by trusting the assertion previously issued by the Identity Provider
3. Any situation that requires the propagation of authorization/authentication information from one domain to another

11.7.1.4.1.1. Process Overview

The client must first obtain the SAML assertion from PicketLink STS by sending a WS-Trust request to the token service. This process usually involves authentication of the client. After obtaining the SAML assertion from the STS, the client includes the assertion in the security context of the EJB request before invoking an operation on the bean. Upon receiving the invocation, the EJB container extracts the assertion and validates it by sending a WS-Trust validate message to the STS. If the assertion is considered valid by the STS (and the proof of possession token has been verified if needed), the client is authenticated.

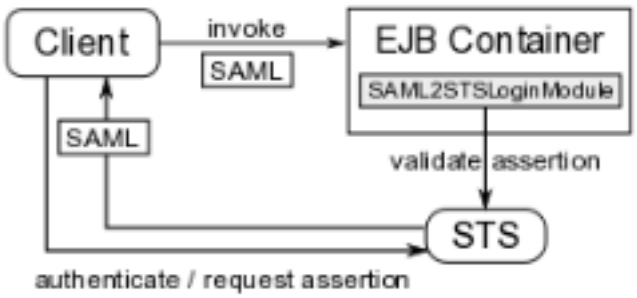


Figure 11.4. TODO Gliffy image title empty

On JBoss, the SAML assertion validation process is handled by the SAML2STSLoginModule. It reads properties from a configurable file (specified by the configFile option) and establishes communication with the STS based on these properties. We will see how a configuration file looks like later on. If the assertion is valid, a Principal is created using the assertion subject name and

if the assertion contains roles, these roles are also extracted and associated with the caller's Subject.

The client must first obtain the SAML assertion from the PicketLink STS or you Identity Provider. This process usually involves authentication of the client. After obtaining the SAML assertion, the client includes the assertion in the security context of the EJB request before invoking an operation on the bean. Upon receiving the invocation, the EJB container extracts the assertion and validates it by sending a WS-Trust validate message to the STS. If the assertion is considered valid by the STS (and the proof of possession token has been verified if needed), the client is authenticated.

On JBoss, the SAML assertion validation process is handled by the Section 11.7.1.5.3, “SAML2STSLoginModule” . It reads properties from a configurable file (specified by the configFile option) and establishes communication with the STS based on these properties. We will see how a configuration file looks like later on. If the assertion is valid, a Principal is created using the assertion subject name and if the assertion contains roles, these roles are also extracted and associated with the caller's Subject.

11.7.1.4.2. Configuration

This section will cover two possible scenarios to protect and access your secured EJB endpoints. The main difference between these two scenarios is where the EJB client is deployed.

- Remote EJB Client using JNDI
- EJB Client is deployed at the same instance than your EJB endpoints

11.7.1.4.2.1. Remote EJB Client using JNDI

> **Important**
>
> Before starting, please take a look at the following documentation Remote EJB invocations via JNDI [https://docs.jboss.org/author/display/AS71/Remote+EJB+invocations+via+JNDI+-+EJB+client+API+or+remote-naming+project] .
>
> The configuration described in this section only works with versions 7.2.0+ and 7.1.3+ of JBoss Application Server.

If your endpoints are accessible from remote clients (in a different VM or server than your endpoints) you need to configure your JBoss Application Server 7 to allow use a SAML Assertion during the InitialContext creation.

Basically, the configuration envolves the following steps:

1. Add a new Security Realm to your standalone.xml
2. Create a Security Domain using the Section 11.7.1.5.3, “SAML2STSLoginModule”

3. Change the Remoting Connector to use the new Security Realm

11.7.1.4.2.1.1. Add a new Security Realm

> **Important**
>
> Security Realms [https://docs.jboss.org/author/display/AS71/Security+Realms] are better described in the JBoss Application Server Documentation.

Edit your standalone.xml and add the following configuration for a new Security Realm:

```
<security-realm name="SAMLRealm">
    <authentication>
        <jaas name="ejb-remoting-sts"/>
    </authentication>
</security-realm>
```

The configuration above defines a Security Realm that delegates the username/password information to a JAAS Security Domain (that we'll create later) in order to authenticate an user.

When using the JAAS configuration for Security Realms, the remoting subsystem enables the PLAIN SASL authentication. This will allow your remote clients send the username/password where the password would be the previously issued SAML Assertion.In our case, the password will be the String representation of the SAML Assertion.

> **Tip**
>
> Make sure you also enable SSL. Otherwise all communication with the server will be done using plain text.

11.7.1.4.2.1.2. Create a Security Domain using the SAML2STSLoginModule

Edit your standalone.xml and add the following configuration for a new Security Domain:

```
<security-domain name="ejb-remoting-sts" cache-type="default">
    <authentication>
                                                                                <login-
module       code="org.picketlink.identity.federation.bindings.jboss.auth.SAML2STSLoginModule"
 flag="required" module="org.picketlink">
                    <module-option  name="configFile"  value="${jboss.server.config.dir}/sts-
config.properties"/>
            <module-option name="password-stacking" value="useFirstPass"/>
        </login-module>
    </authentication>
</security-domain>
```

This configuration above defines a Security Domain that uses the SAML2STSLoginModule to get the String representation of the SAML Assertion and validate it against the Security Token Service.

You may notice that we provided a properties file as module-option. This properties file defines all the configuration needed to invoke the PicketLink STS. It should look like this:

```
serviceName=PicketLinkSTS
portName=PicketLinkSTSPort
endpointAddress=http://localhost:8080/picketlink-sts/PicketLinkSTS
username=admin
#password=admin
password=MASK-0BbleBL2LZk=
salt=18273645
iterationCount=56

#java    -cp    picketlink-fed-core.jar    org.picketlink.identity.federation.core.util.PBEUtils
 18273645 56 admin
#Encoded password: MASK-0BbleBL2LZk=
```

This security domain will be used to authenticate your remote clients during the creation of the JNDI Initial Context.

11.7.1.4.2.1.3. Change the Remoting Connector Security Realm

Edit your standalone.xml and change the security-realm attribute of the remoting connector:

```
<subsystem xmlns="urn:jboss:domain:remoting:1.1">
    <connector name="remoting-connector" socket-binding="remoting" security-realm="SAMLRealm"/>
</subsystem>
```

The connector configuration is already present in your standalone.xml. You only need to change the security-realm attribute to match the one we created before.

11.7.1.4.2.1.4. EJB Remote Client

The code above shows you how a EJB Rremote Client may look like:

```
// add the JDK SASL Provider that allows to use the PLAIN SASL Client
Security.addProvider(new Provider());

Element assertion = getAssertionFromSTS("UserA", "PassA");

// JNDI environment configuration properties
Hashtable<String, Object> env = new Hashtable<String, Object>();

env.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
env.put("java.naming.factory.initial",
 "org.jboss.naming.remote.client.InitialContextFactory");
env.put("java.naming.provider.url", "remote://localhost:4447");
env.put("jboss.naming.client.ejb.context", "true");
```

```
env.put("jboss.naming.client.connect.options.org.xnio.Options.SASL_POLICY_NOPLAINTEXT",
 "false");
env.put("javax.security.sasl.policy.noplaintext", "false");

// provide the user principal and credential. The credential is the previously issued SAML
 assertion
env.put(Context.SECURITY_PRINCIPAL, "admin");
env.put(Context.SECURITY_CREDENTIALS, DocumentUtil.getNodeAsString(assertion));

// create the JNDI Context and perform the authentication using the SAML2STSLoginModule
Context context = new InitialContext(env);

// lookup the EJB
EchoService     object     =     (EchoService)     context.lookup("ejb-test/EchoServiceImpl!
org.picketlink.test.trust.ejb.EchoService");

// If everything is ok the Principal name will be added to the message
Assert.assertEquals("Hi UserA", object.echo("Hi "));
```

11.7.1.4.3. References

- JBoss AS 5 : https://community.jboss.org/wiki/SAMLEJBIntegrationWithPicketLinkSTS

11.7.1.5. STS Login Modules

This page references the PicketLink Login Modules for the Security Token Server.

11.7.1.5.1. References

> **Tip**
>
> PicketLink STS Login Modules [http://community.jboss.org/wiki/PicketLinkSTSLoginModules] has the required details.

11.7.1.5.2. JBWSTokenIssuingLoginModule

11.7.1.5.2.1. Fully Qualified Name

org.picketlink.trust.jbossws.jaas. **JBWSTokenIssuingLoginModule**

11.7.1.5.2.2. Objective

A variant of the PicketLink STSIssuingLoginModule that allows us to:

1. Inject BinaryTokenHandler or SAML2Handler or both as client side handlers to the STS WS call.
2. Inject the JaasSecurityDomainServerSocketFactory DomainSocketFactory as a request property to the BindingProvider set to the key "org.jboss.ws.socketFactory". This is useful

for mutually authenticated SSL with the STS, where in we use a trust store defined by a JaasSecurityDomain instance.

11.7.1.5.2.3. Configuration

Options Include:

- **configFile** : a properties file that gives details on the STS to the login module. This can be optional if you want to specify values directly.
- **handlerChain** : Comma separated list of handlers you need to set for handling outgoing message to STS. Values: binary (to inject BinaryTokenHandler), saml2 (to inject SAML2Handler), map (to inject MapBasedTokenHandler) or class name of your own handler with default constructor.
- **cache.invalidation** : set it to "true" if you want the JBoss auth cache to invalidate caches based on saml token expiry. By default, this value is false.
- **inject.callerprincipa** l: set it to "true" if the login module should add a group principal called "CallerPrincipal" to the subject. This is useful in JBoss AS for programmatic security in web/ejb components.
- **groupPrincipalName** : by default, JBoss AS security uses "Roles" as the group principal name in the subject. You can give a different value.
- **endpointAddress** : endpoint url of STS
- **serviceName** : service Name of STS
- **portName** : port name of STS
- **username** : username of account on STS.
- **password** : password of account on STS
- **wsaIssuer** : if you need to customize the WS-Addressing Issuer address in the WS-Trust call to the STS.
- **wspAppliesTo** : if you need to customize the WS-Policy AppliesTo in the WS-Trust call to the STS.
- **securityDomainForFactory** : if you have a JaasSecurityDomain mbean service in JBoss AS that provides the truststore.
- **map.token.key** : key to find binary token in JAAS sharedState map. Defaults to "ClientID".
- **soapBinding** : allow to change SOAP binding for SAML reuest.

- **requestType** : allows to override SAML request type when sending request to STS. Default: "http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue" Other possible value: "http://docs.oasis-open.org/ws-sx/ws-trust/200512/Validate".

Note: The configFile option is optional. If you provide that, then it should be as below.

Configuration file such as sts-client.properties.

serviceName=PicketLinkSTS

portName=PicketLinkSTSPort

endpointAddress=http://localhost:8080/picketlink-sts/PicketLinkSTS

username=admin

password=admin

wsaIssuer=http://localhost:8080/someissuer

wspAppliesTo=http://localhost:8080/testws

Note:

- the password can be masked according to http://community.jboss.org/wiki/PicketLinkConfigurationMaskpassword which would give us something like, password=MASK-dsfdsfdslkfh
- wsaIssuer can be **optionally** added if you want a value for the WS-Addressing issuer in the WS-Trust call to the STS.
- wspAppliesTo can be **optionally** added if you want a value for WS-Policy AppliesTo in the WS-Trust call to the STS.
- serviceName, portName, endpointAddress are **mandatory** .
- username and password keys are not needed if you are using mutual authenticated ssl (MASSL) with the STS.

11.7.1.5.2.3.1. SSL DomainSocketFactory in use by the client side

Many a times, the login module has to communicate with the STS over a mutually authenticated SSL. In this case, you want to specify the truststore. JBoss AS provides JaasSecurityDomain mbean to specify truststore. For this reason, there is a special JaasSecurityDomainServerSocketFactory that can be used for making the JBWS calls. Specify the "securityDomainForFactory" module option with the security domain name (in the JaasSecurityDomain mbean service).

11.7.1.5.2.4. Example configurations

Either you specify the module options directly or you can use a properties file for the STS related properties.

11.7.1.5.2.4.1. Configuration specified directly

```
<application-policy name="saml-issue-token">
 <authentication>
                                                                                <login-module

 flag="required">

       <module-option name="password-stacking">useFirstPass</module-option>

       <module-option name="endpointAddress">http://somests</module-option>

        <module-option name="serviceName">PicketLinkSTS</module-option>

       <module-option name="portName">PicketLinkPort</module-option>

       <module-option name="username">admin</module-option>

       <module-option name="password">admin</module-option>

       <module-option name="inject.callerprincipal">true</module-option>
       <module-option name="groupPrincipalName">Membership</module-option>
     </login-module>
    </authentication>
</application-policy>
```

11.7.1.5.2.4.2. Configuration with configFileOption

```
<application-policy name="saml-issue-token">
 <authentication>
                                                                                <login-module

 flag="required">
       <module-option name="configFile">/sts-client.properties</module-option>
       <module-option name="password-stacking">useFirstPass</module-option>
<module-option name="cache.invalidation">true</module-option>
<module-option name="inject.callerprincipal">true</module-option>
<module-option name="groupPrincipalName">Membership</module-option>
     </login-module>
    </authentication>
</application-policy>
```

11.7.1.5.2.4.3. Dealing with Roles

If the STS sends roles via Attribute Statements in the SAML assertion, then the user has to use the SAMLRoleLoginModule.

```
<application-policy name="saml">
  <authentication>
         <login-module  code="org.picketlink.trust.jbossws.jaas.JBWSTokenIssuingLoginModule"
 flag="required">
         <module-option name="endpointAddress">SOME_URL</module-option>
         <module-option name="serviceName">SecurityTokenService</module-option>
         <module-option name="portName">RequestSecurityToken</module-option>
         <module-option name="inject.callerprincipal">true</module-option>
         <module-option name="handlerChain">binary</module-option>
    </login-module>
   <login-module code="org.picketlink.trust.jbossws.jaas.SAMLRoleLoginModule"  flag="required"/
>
  </authentication>
</application-policy>
```

If the STS does not send roles, then the user has to configure a different JAAS login module to pick the roles for the username. Something like the UsernamePasswordLoginModule.

```
<application-policy xmlns="urn:jboss:security-beans:1.0" name="binary">
      <authentication>
            <login-module code="org.picketlink.trust.jbossws.jaas.JBWSTokenIssuingLoginModule"
 flag="required">
                  <module-option  name="endpointAddress">http://localhost:8080/picketlink-sts/
PicketLinkSTS</module-option>
            <module-option name="serviceName">PicketLinkSTS</module-option>
            <module-option name="portName">PicketLinkSTSPort</module-option>
            <module-option name="inject.callerprincipal">true</module-option>
            <module-option name="handlerChain">binary</module-option>
            <module-option name="username">admin</module-option>
            <module-option name="password">MASK-0BbleBL2LZk=</module-option>
            <module-option name="salt">18273645</module-option>
            <module-option name="iterationCount">56</module-option>
            <module-option name="useOptionsCredentials">true</module-option>
            <module-option name="overrideDispatch">true</module-option>
             <module-option name="wspAppliesTo">http://services.testcorp.org/provider1</module-
option>
            <module-option name="wsaIssuer">http://something</module-option>
            <module-option name="password-stacking">useFirstPass</module-option>
         </login-module>

       <login-module code="org.jboss.security.auth.spi.UsersRolesLoginModule" flag="required">
            <module-option name="usersProperties">sts-users.properties</module-option>
            <module-option name="rolesProperties">sts-roles.properties</module-option>
            <module-option name="password-stacking">useFirstPass</module-option>
         </login-module>
      </authentication>
   </application-policy>
```

11.7.1.5.3. SAML2STSLoginModule

11.7.1.5.3.1. FQN

org.picketlink.identity.federation.bindings.jboss.auth. **SAML2STSLoginModule**

11.7.1.5.3.2. Author:

Stefan Guilhen

11.7.1.5.3.3. Objective

This LoginModule authenticates clients by validating their SAML assertions with an external security token service (such as PicketLinkSTS). If the supplied assertion contains roles, these roles are extracted and included in the Group returned by the getRoleSets method.

The LoginModule could be also used to retrieve and validate SAML assertion token from HTTP request header.

11.7.1.5.3.4. Module Options

This module defines the following module options:

- **configFile** - this property identifies the properties file that will be used to establish communication with the external security token service.
- **cache.invalidation** : set it to true if you require invalidation of JBoss Auth Cache at SAML Principal expiration.
- **jboss.security.security_domain** -security domain at which Principal will expire if cache.invalidation is used.
- **roleKey** : key of the attribute name that we need to use for Roles from the SAML assertion. This can be a comma-separated string values such as (Role,Membership)
- **localValidation** : if you want to validate the assertion locally for signature and expiry
- **localValidationSecurityDomain** : the security domain for the trust store information (via the JaasSecurityDomain)
- **tokenEncodingType** : encoding type of SAML token delivered via http request's header. Possible values are:
  - base64 - content encoded as base64. In case of encoding will vary between base64 and gzip use base64 and LoginModule will detect gzipped data.
  - gzip - gzipped content encoded as base64
  - none - content not encoded in any way
- **samlTokenHttpHeader** - name of http request header to fetch SAML token from. For example: "Authorize"
- **samlTokenHttpHeaderRegEx** - Java regular expression to be used to get SAML token from "samlTokenHttpHeader". Example: use: . **"(.**)".* to parse SAML token from header content like this: SAML_assertion="HHDHS=", at the same time set samlTokenHttpHeaderRegExGroup to 1.

- **samlTokenHttpHeaderRegExGroup** - Group value to be used when parsing out value of http request header specified by "samlTokenHttpHeader" using "samlTokenHttpHeaderRegEx".

```
pattern = Pattern.compile(samlTokenHttpHeaderRegEx, Pattern.DOTALL);
Matcher m = pattern.matcher(content);
m.matches();
m.group(samlTokenHttpHeaderRegExGroup)
```

Any properties specified besides the above properties are assumed to be used to configure how the STSClient will connect to the STS. For example, the JBossWS StubExt.PROPERTY_SOCKET_FACTORY can be specified in order to inform the socket factory that must be used to connect to the STS. All properties will be set in the request context of the Dispatch instance used by the STSClient to send requests to the STS.

An example of a configFile can be seen bellow:

```
serviceName=PicketLinkSTS
portName=PicketLinkSTSPort
endpointAddress=[http://localhost:8080/picketlink-sts/PicketLinkSTS]
username=JBoss
password=JBoss
```

The first three properties specify the STS endpoint URL, service name, and port name. The last two properties specify the username and password that are to be used by the application server to authenticate to the STS and have the SAML assertions validated.

NOTE: Sub-classes can use getSTSClient() method to customize the STSClient class to make calls to STS

11.7.1.5.3.5. Examples

Example Configuration 1:

```
<application-policy xmlns="urn:jboss:security-beans:1.0" name="cache-test">
      <authentication>
                                                                                <login-
module       code="org.picketlink.identity.federation.bindings.jboss.auth.SAML2STSLoginModule"
 flag="required">
            <module-option name="password-stacking">useFirstPass</module-option>
            <module-option name="configFile">sts-config.properties</module-option>
            <module-option name="cache.invalidation">true</module-option>
            <module-option name="localValidation">true</module-option>
            <module-option name="localValidationSecurityDomain">MASSL</module-option>
        </login-module>
      </authentication>
   </application-policy>
```

Example Configuration 2 using http header and local validation:

```
   <application-policy xmlns="urn:jboss:security-beans:1.0" name="service">
      <authentication>
                                                                                <login-
module       code="org.picketlink.identity.federation.bindings.jboss.auth.SAML2STSLoginModule"
 flag="required">
            <module-option name="password-stacking">useFirstPass</module-option>
            <module-option name="cache.invalidation">true</module-option>
            <module-option name="localValidation">true</module-option>
          <module-option name="localValidationSecurityDomain">java:jaas/localValidationDomain</
module-option>
            <module-option name="tokenEncodingType">gzip</module-option>
            <module-option name="samlTokenHttpHeader">Auth</module-option>
            <module-option name="samlTokenHttpHeaderRegEx">.*"(.*)".*</module-option>
            <module-option name="samlTokenHttpHeaderRegExGroup">1</module-option>
         </login-module>
                   <login-module  code="org.picketlink.trust.jbossws.jaas.SAMLRoleLoginModule"
 flag="required"/>
      </authentication>
   </application-policy>
```

In case of local validation here is example of jboss-beans.xml file to use to configure JAAS Security Domain for (JBoss AS6 or EAP5):

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns="urn:jboss:bean-deployer:2.0">
   <!-- localValidationDomain bean -->
   <bean name="LocalValidationBean" class="org.jboss.security.plugins.JaasSecurityDomain">
     <constructor>
          <parameter>localValidationDomain</parameter>
     </constructor>
     <property name="keyStoreURL">file://${jboss.server.home.dir}/conf/stspub.jks</property>
     <property name="keyStorePass">keypass</property>
     <property name="keyStoreAlias">sts</property>
    <property name="securityManagement"><inject bean="JNDIBasedSecurityManagement"/></property>
   </bean>
</deployment>
```

For JBoss AS7 or JBoss EAP6 add following security domain to your configuration file:

```
<security-domain name="localValidationDomain">
      <jsse
         keystore-password="keypass"
         keystore-type="JKS"
         keystore-url="file:///${jboss.server.config.dir}/stspub.jks"
         server-alias="sts"/>
</security-domain>
```

and reference this security domain as: <module-option name="localValidationSecurityDomain">localValidationDomain</module-option>.

11.7.1.5.4. SAMLTokenCertValidatingLoginModule

org.picketlink.identity.federation.bindings.jboss.auth. **SAMLTokenCertValidatingLoginModule**

11.7.1.5.4.1. Author:

Peter Skopek

11.7.1.5.4.2. Objective

This LoginModule authenticates clients by validating their SAML assertions locally. If the supplied assertion contains roles, these roles are extracted and included in the Group returned by the getRoleSets method.

The LoginModule is designed to validate SAML token using X509 certificate stored in XML signature within SAML assertion token.

It validates:

1. CertPath against specified truststore. It has to have common valid public certificate in the trusted entries.
2. X509 certificate stored in SAML token didn't expire
3. if signature itself is valid
4. SAML token expiration

11.7.1.5.4.3. Module Options

This module defines the following module options:

- **roleKey** : key of the attribute name that we need to use for Roles from the SAML assertion. This can be a comma-separated string values such as (Role,Membership)
- **localValidationSecurityDomain** : the security domain for the trust store information (via the JaasSecurityDomain)
- **cache.invalidation** - set it to true if you require invalidation of JBoss Auth Cache at SAML Principal expiration.
- **jboss.security.security_domain** -security domain at which Principal will expire if cache.invalidation is used.
- **tokenEncodingType** : encoding type of SAML token delivered via http request's header. Possible values are:
  - base64 - content encoded as base64. In case of encoding will vary between base64 and gzip use base64 and LoginModule will detect gzipped data.
  - gzip - gzipped content encoded as base64

- none - content not encoded in any way

- **samlTokenHttpHeader** - name of http request header to fetch SAML token from. For example: "Authorize"
- **samlTokenHttpHeaderRegEx** - Java regular expression to be used to get SAML token from "samlTokenHttpHeader". Example: use: . **"(.**)".* to parse SAML token from header content like this: SAML_assertion="HHDHS=", at the same time set samlTokenHttpHeaderRegExGroup to 1.
- **samlTokenHttpHeaderRegExGroup** - Group value to be used when parsing out value of http request header specified by "samlTokenHttpHeader" using "samlTokenHttpHeaderRegEx".

```
pattern = Pattern.compile(samlTokenHttpHeaderRegEx, Pattern.DOTALL);
Matcher m = pattern.matcher(content);
m.matches();
m.group(samlTokenHttpHeaderRegExGroup)
```

11.7.1.5.4.4. Examples

Example Configuration 1:

```
<application-policy xmlns="urn:jboss:security-beans:1.0" name="certpath">
      <authentication>
                                                                                <login-module

 flag="required">
            <module-option name="password-stacking">useFirstPass</module-option>
            <module-option name="cache.invalidation">true</module-option>
          <module-option name="localValidationSecurityDomain">java:jaas/localValidationDomain</
module-option>
         </login-module>
      </authentication>
</application-policy>
```

Example Configuration 2 using http header:

```
   <application-policy xmlns="urn:jboss:security-beans:1.0" name="service">
      <authentication>
                                                                                    <login-
module       code="org.picketlink.identity.federation.bindings.jboss.auth.SAML2STSLoginModule"
 flag="required">
            <module-option name="password-stacking">useFirstPass</module-option>
            <module-option name="cache.invalidation">true</module-option>
          <module-option name="localValidationSecurityDomain">java:jaas/localValidationDomain</
module-option>
            <module-option name="tokenEncodingType">gzip</module-option>
            <module-option name="samlTokenHttpHeader">Auth</module-option>
            <module-option name="samlTokenHttpHeaderRegEx">.*"(.*)".*</module-option>
```

```
            <module-option name="samlTokenHttpHeaderRegExGroup">1</module-option>
         </login-module>
      </authentication>
   </application-policy>
```

Example of jboss-beans.xml file to use to configure JAAS Security Domain containing trust store for above examples:

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns="urn:jboss:bean-deployer:2.0">
   <!-- localValidationDomain bean -->
   <bean name="LocalValidationBean" class="org.jboss.security.plugins.JaasSecurityDomain">
     <constructor>
          <parameter>localValidationDomain</parameter>
     </constructor>
     <property name="keyStoreURL">file://${jboss.server.home.dir}/conf/stspub.jks</property>
     <property name="keyStorePass">keypass</property>
     <property name="keyStoreAlias">sts</property>
    <property name="securityManagement"><inject bean="JNDIBasedSecurityManagement"/></property>
   </bean>
</deployment>
```

11.7.1.5.5. STSValidatingLoginModule

11.7.1.5.5.1. FQN:

org.picketlink.identity.federation.core.wstrust.auth.STSValidatingLoginModule

11.7.1.5.5.2. Author:

Daniel Bevenius

11.7.1.5.5.3. Objective/Features:

- Calls the configured STS and validates an available security token.
- A call to STS typically requires authentication. This LoginModule uses credentials from one of the following sources:
  - Its properties file, if the *useOptionsCredentials* module-option is set to true
  - Previous login module credentials if the *password-stacking* module-option is set to useFirstPass
  - From the configured *CallbackHandler* by supplying a *Name* and *Password Callback*
- Upon successful authentication, the SamlCredential is inserted in the Subject's public credentials if one with the same Assertion is not found to be already present there.
- New features included since 1.0.4 based on PLFED-87 [https://jira.jboss.org/browse/PLFED-87] :

- If a Principal MappingProvider is configured, retrieves and inserts the Principal into the Subject
- If a RoleGroup MappingProvider is configured, retrieves and inserts the user roles into the Subject
- Roles can only be returned if they are included in the Security Token. Configure your STS to return roles through an AttributeProvider

11.8. Extensions

11.8.1. Extensions

This page shows all the extensions and customizations available in the PicketLink project.

11.8.2. PicketLinkAuthenticator

11.8.2.1. PicketLinkAuthenticator

11.8.2.1.1. FQN

org.picketlink.identity.federation.bindings.tomcat.PicketLinkAuthenticator

11.8.2.1.2. Objective

An authenticator that delegates actual authentication to a realm, and in turn to a security manager, by presenting a "conventional" identity. The security manager must accept the conventional identity and generate the real identity for the authenticated principal.

11.8.2.1.3. JBoss Application Server 7.x Configuration

Your web.xml will define some security constraints. But it will define a <login-config> that is different from the servlet specifcation mandated BASIC, CLIENT-CERT, FORM or DIGEST methods. We suggest the use of SECURITY_DOMAIN as the method.

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Restricted Access - Get Only</web-resource-name>
        <url-pattern>/*</url-pattern>
        <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
  <role-name>STSClient</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>NONE</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

```
<security-role>
    <role-name>STSClient</role-name>
</security-role>

<login-config>
    <auth-method>SECURITY_DOMAIN</auth-method>
    <realm-name>SECURITY_DOMAIN</realm-name>
    <form-login-config>
 <form-login-page>/login.html</form-login-page>
 <form-error-page>/error.html</form-error-page>
    </form-login-config>
</login-config>
```

> **Important**
>
> Note that we defined two pages in the **<form-login-config>** : **login.html** and **error.html** . Both pages must exists inside your deployment.

Change your WEB-INF/jboss-web.xml to configure the *PicketLinkAuthenticator* as a valve:

```
<jboss-web>
 <security-domain>authenticator</security-domain>
 <context-root>authenticator</context-root>
 <valve>
  <class-name>org.picketlink.identity.federation.bindings.tomcat.PicketLinkAuthenticator
  </class-name>
 </valve>
</jboss-web>
```

We also defined a **<security-domain>** configuration with the name of the security domain that you configured in your standalone.xml:

```
<security-domain name="authenticator" cache-type="default">
    <authentication>
         <login-module code="org.picketlink.test.trust.loginmodules.TestRequestUserLoginModule"
 flag="required">
            <module-option name="usersProperties" value="users.properties"/>
            <module-option name="rolesProperties" value="roles.properties"/>
        </login-module>
    </authentication>
</security-domain>
```

> **Tip**
>
> To use PicketLink you need to define it as a module dependency using the META-INF/jboss-deployment-structure.xml.

11.8.2.1.4. JBoss Application Server 5.x Configuration

Your web.xml will define some security constraints. But it will define a <login-config> that is different from the servlet specifcation mandated BASIC, CLIENT-CERT, FORM or DIGEST methods. We suggest the use of SECURITY-DOMAIN as the method.

Create a context.xml in your WEB-INF directory of your web-archive.

```
<Context>
                                                                                <Valve
 className="org.picketlink.identity.federation.bindings.tomcat.PicketLinkAuthenticator" />
</Context>
```

Your web.xml may look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
   xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
   http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

   <description>Sales Application</description>

   <security-constraint>
      <display-name>Restricted</display-name>
      <web-resource-collection>
         <web-resource-name>Restricted Access</web-resource-name>
         <url-pattern>/*</url-pattern>
      </web-resource-collection>
      <auth-constraint>
         <role-name>Sales</role-name>
      </auth-constraint>
      <user-data-constraint>
         <transport-guarantee>NONE</transport-guarantee>
      </user-data-constraint>
   </security-constraint>

   <security-role>
      <role-name>Sales</role-name>
   </security-role>

   <login-config>
      <auth-method>SECURITY-DOMAIN</auth-method>
   </login-config>
</web-app>
```

Warning

NOTE: The use of SECURITY-DOMAIN as the auth-method.

The war should be packaged as a regular web archive.

11.8.2.1.4.1. Default Configuration at Global Level

If you have a large number of web applications and it is not practical to include context.xml in all the war files, then you can configure the "authenticators" attribute in the war-deployers-jboss-beans.xml file in /server/default/deployers/jbossweb.deployer/META-INF of your JBoss AS instance.

```
<property name="authenticators">
                            <map   class="java.util.Properties"   keyClass="java.lang.String"
 valueClass="java.lang.String">
            <entry>
               <key>BASIC</key>
               <value>org.apache.catalina.authenticator.BasicAuthenticator</value>
            </entry>
            <entry>
               <key>CLIENT-CERT</key>
               <value>org.apache.catalina.authenticator.SSLAuthenticator</value>
            </entry>
            <entry>
               <key>DIGEST</key>
               <value>org.apache.catalina.authenticator.DigestAuthenticator</value>
            </entry>
            <entry>
               <key>FORM</key>
               <value>org.apache.catalina.authenticator.FormAuthenticator</value>
            </entry>
            <entry>
               <key>NONE</key>
               <value>org.apache.catalina.authenticator.NonLoginAuthenticator</value>
            </entry>
               <key>SECURITY-DOMAIN</key>
           <value>org.picketlink.identity.federation.bindings.tomcat.PicketLinkAuthenticator</
value>
            </entry>

         </map>
      </property>
```

11.8.2.1.4.2. Testing

1. Go to the deploy directory.

2. cp -R jmx-console.war test.war

3. In deploy/test.war/WEB-INF/web.xml, change the auth-method element to SECURITY-DOMAIN.

```
<login-config>
      <auth-method>SECURITY-DOMAIN</auth-method>
      <realm-name>JBoss JMX Console</realm-name>
   </login-config>
```

5. Also uncomment the security constraints in web.xml. It should look as follows.

```
<!-- A security constraint that restricts access to the HTML JMX console
   to users with the role JBossAdmin. Edit the roles to what you want and
   uncomment the WEB-INF/jboss-web.xml/security-domain element to enable
   secured access to the HTML JMX console.
   -->
   <security-constraint>
     <web-resource-collection>
       <web-resource-name>HtmlAdaptor</web-resource-name>
       <description>An example security config that only allows users with the
         role JBossAdmin to access the HTML JMX console web application
       </description>
       <url-pattern>/*</url-pattern>
       <http-method>GET</http-method>
       <http-method>POST</http-method>
     </web-resource-collection>
     <auth-constraint>
       <role-name>JBossAdmin</role-name>
     </auth-constraint>
   </security-constraint>
```

7. In the /server/default/conf/jboss-log4j.xml , add trace category for org.jboss.security.

8. Start JBoss AS.

9. Go to the following url: http://localhost:8080/test/

10. You should see a HTTP 403 message.

11. If you look inside the log, log/server.log, you will see the following exception trace:

```
2011-04-20 11:02:01,714 TRACE [org.jboss.security.plugins.auth.JaasSecurityManagerBase.jmx-
console] (http-127.0.0.1-8080-1) Login failure
javax.security.auth.login.FailedLoginException: Password Incorrect/Password Required
                                                                                    at
 org.jboss.security.auth.spi.UsernamePasswordLoginModule.login(UsernamePasswordLoginModule.java:252)
                                                                                    at
 org.jboss.security.auth.spi.UsersRolesLoginModule.login(UsersRolesLoginModule.java:152)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
                                                                                    at
 sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:597)
        at javax.security.auth.login.LoginContext.invoke(LoginContext.java:769)
```

```
        at javax.security.auth.login.LoginContext.access$000(LoginContext.java:186)
        at javax.security.auth.login.LoginContext$4.run(LoginContext.java:683)
        at java.security.AccessController.doPrivileged(Native Method)
        at javax.security.auth.login.LoginContext.invokePriv(LoginContext.java:680)
        at javax.security.auth.login.LoginContext.login(LoginContext.java:579)
                                                                                at
  org.jboss.security.plugins.auth.JaasSecurityManagerBase.defaultLogin(JaasSecurityManagerBase.java:552)
                                                                                at
  org.jboss.security.plugins.auth.JaasSecurityManagerBase.authenticate(JaasSecurityManagerBase.java:486)
                                                                                at
  org.jboss.security.plugins.auth.JaasSecurityManagerBase.isValid(JaasSecurityManagerBase.java:365)
                                                                                at
  org.jboss.security.plugins.JaasSecurityManager.isValid(JaasSecurityManager.java:160)
        at org.jboss.web.tomcat.security.JBossWebRealm.authenticate(JBossWebRealm.java:384)
                                                                                at
  org.picketlink.identity.federation.bindings.tomcat.PicketLinkAuthenticator.authenticate(PicketLinkAuthenticato
                                                                                at
  org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:491)
        at org.jboss.web.tomcat.security.JaccContextValve.invoke(JaccContextValve.java:92)
                                                                                at
  org.jboss.web.tomcat.security.SecurityContextEstablishmentValve.process(SecurityContextEstablishmentValve.java
                                                                                at
  org.jboss.web.tomcat.security.SecurityContextEstablishmentValve.invoke(SecurityContextEstablishmentValve.java:
        at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:127)
        at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:102)
                                                                                at
  org.jboss.web.tomcat.service.jca.CachedConnectionValve.invoke(CachedConnectionValve.java:158)
       at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:109)
        at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:330)
        at org.apache.coyote.http11.Http11Processor.process(Http11Processor.java:829)
                                            at      org.apache.coyote.http11.Http11Protocol
$Http11ConnectionHandler.process(Http11Protocol.java:598)
        at org.apache.tomcat.util.net.JIoEndpoint$Worker.run(JIoEndpoint.java:447)
        at java.lang.Thread.run(Thread.java:662)
```

As you can see from the stack trace, PicketLinkAuthenticator method has been kicked in.

11.9. PicketLink API

11.9.1. Working with SAML Assertions

11.9.1.1. Introduction

This page shows you how to use the PicketLink API to programatically work with SAML Assertions.

The examples above demonstrates the following scenarios:

- How to parse a XML to a PicketLink AssertionType
- How to sign SAML Assertions
- How to validate SAML Assertions

The following API classes were used:

- org.picketlink.identity.federation.saml.v2.assertion.AssertionType
- org.picketlink.identity.federation.core.saml.v2.util.AssertionUtil
- org.picketlink.identity.federation.core.parsers.saml.SAMLParser
- org.picketlink.identity.federation.core.saml.v2.writers.SAMLAssertionWriter
- org.picketlink.identity.federation.api.saml.v2.sig.SAML2Signature
- org.picketlink.identity.federation.core.impl.KeyStoreKeyManager

Important

Please, check the javadoc for more informations about these classes.

11.9.1.2. Parsing SAML Assertions

The PicketLink API provides the **org.picketlink.identity.federation.saml.v2.assertion.AssertionType** class to encapsulate the informations parsed from a SAML Assertion.

Let's suppose we have the following SAML Assertion:

```
<saml:Assertion                                xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
     ID="ID_75291c31-93f7-4f7f-8422-aacdb07466ee"     IssueInstant="2012-05-25T10:40:58.912-03:00"
 Version="2.0">
    <saml:Issuer>http: //192.168.1.1:8080/idp-sig/</saml:Issuer>
    <saml:Subject>
             <saml:NameID  Format="urn:oasis:names:tc:SAML:2.0:nameid-format:persistent">user</
saml:NameID>
        <saml:SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
           <saml:SubjectConfirmationData InResponseTo="ID_326a389f-6a8a-4712-b71d-77aa9c36795c"
         NotBefore="2012-05-25T10:40:58.894-03:00"        NotOnOrAfter="2012-05-25T10:41:00.912-03:00"
 Recipient="http://192.168.1.4:8080/fake-sp" />
        </saml:SubjectConfirmation>
    </saml:Subject>
                             <saml:Conditions       NotBefore="2012-05-25T10:40:57.912-03:00"
 NotOnOrAfter="2012-05-25T10:41:00.912-03:00" />
    <saml:AuthnStatement AuthnInstant="2012-05-25T10:40:58.981-03:00">
        <saml:AuthnContext>
                  <saml:AuthnContextClassRef>urn:oasis:names:tc:SAML:2.0:ac:classes:Password</
saml:AuthnContextClassRef>
        </saml:AuthnContext>
    </saml:AuthnStatement>
    <saml:AttributeStatement>
        <saml:Attribute Name="Role">
            <saml:AttributeValue xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:type="xs:string">test-role1</saml:AttributeValue>
        </saml:Attribute>
        <saml:Attribute Name="Role">
            <saml:AttributeValue xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:type="xs:string">test-role2</saml:AttributeValue>
```

```
        </saml:Attribute>
        <saml:Attribute Name="Role">
            <saml:AttributeValue xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:type="xs:string">test-role3</saml:AttributeValue>
        </saml:Attribute>
    </saml:AttributeStatement>
</saml:Assertion>
```

The code to parse this XML is:

```
/**
     * <p>
      * Parses a SAML Assertion XML representation and convert it to a {@link AssertionType}
 instance.
     * </p>
     *
     * @throws Exception
     */
    @Test
    public void testParseAssertion() throws Exception {
        // get a InputStream from the source XML file
        InputStream samlAssertionInputStream = getSAMLAssertion();

        SAMLParser samlParser = new SAMLParser();

        Object parsedObject = samlParser.parse(samlAssertionInputStream);

        Assert.assertNotNull(parsedObject);
        Assert.assertTrue(parsedObject.getClass().equals(AssertionType.class));

        // cast the parsed object to the expected type, in this case AssertionType
        AssertionType assertionType = (AssertionType) parsedObject;

        // checks if the Assertion has expired.
        Assert.assertTrue(AssertionUtil.hasExpired(assertionType));

        // let's write the parsed assertion to the sysout
        ByteArrayOutputStream baos = new ByteArrayOutputStream();

       SAMLAssertionWriter writer = new SAMLAssertionWriter(StaxUtil.getXMLStreamWriter(baos));

        writer.write(assertionType);

        System.out.println(new String(baos.toByteArray()));
    }
```

11.9.1.3. Signing a SAML Assertion

The PicketLink API provides the **org.picketlink.identity.federation.api.saml.v2.sig.SAML2Signature** to help during signature generation/validation for SAML Assertions.

```
/**
```

```
     * <p>
     * Signs a SAML Assertion.
     * </p>
     *
     * @throws Exception
     */
    @Test
    public void testSignAssertion() throws Exception {
        InputStream samlAssertionInputStream = getSAMLAssertion();

        // convert the InputStream to a DOM Document
        Document document = DocumentUtil.getDocument(samlAssertionInputStream);

        SAML2Signature samlSignature = new SAML2Signature();

        // get the key store manager instance.
        KeyStoreKeyManager keyStoreKeyManager = getKeyStoreManager();

        samlSignature.signSAMLDocument(document, keyStoreKeyManager.getSigningKeyPair());

        // let's print the signed assertion to the sysout
        System.out.println(DocumentUtil.asString(document));
    }
```

As you can see, we need to create a instance of **org.picketlink.identity.federation.core.impl.KeyStoreKeyManager** from where the certificates will be retrieved from. The code bellow shows you how to create it:

```
     /**
     * <p>
     * Creates a {@link KeyStoreKeyManager} instance.
     * </p>
     *
     * @throws Exception
     */
    private KeyStoreKeyManager getKeyStoreManager()
        throws TrustKeyConfigurationException, TrustKeyProcessingException {

        KeyStoreKeyManager keyStoreKeyManager = new KeyStoreKeyManager();

        ArrayList<AuthPropertyType> authProperties = new ArrayList<AuthPropertyType>();

                        authProperties.add(createAuthProperty(KeyStoreKeyManager.KEYSTORE_URL,

jbid_test_keystore.jks").getFile()));
        authProperties.add(createAuthProperty(KeyStoreKeyManager.KEYSTORE_PASS, "store123"));

                    authProperties.add(createAuthProperty(KeyStoreKeyManager.SIGNING_KEY_ALIAS,
 "servercert"));
       authProperties.add(createAuthProperty(KeyStoreKeyManager.SIGNING_KEY_PASS, "test123"));

        keyStoreKeyManager.setAuthProperties(authProperties);

        return keyStoreKeyManager;
    }
```

```
    public AuthPropertyType createAuthProperty(String key, String value) {
        AuthPropertyType authProperty = new AuthPropertyType();

        authProperty.setKey(key);
        authProperty.setValue(value);

        return authProperty;
    }
```

11.9.1.4. Validating a Signed SAML Assertion

The code to validate signatures is almost the same for signing. You still need a KeyStoreKeyManager instance.

```
/**
     * <p>
     * Validates a SAML Assertion.
     * </p>
     *
     * @throws Exception
     */
    @Test
    public void testValidateSignatureAssertion() throws Exception {
        InputStream samlAssertionInputStream = getSAMLSignedAssertion();

        KeyStoreKeyManager keyStoreKeyManager = getKeyStoreManager();

        Document signedDocument = DocumentUtil.getDocument(samlAssertionInputStream);

                                                                                boolean
     isValidSignature     =     AssertionUtil.isSignatureValid(signedDocument.getDocumentElement(),
 keyStoreKeyManager.getSigningKeyPair().getPublic());

        Assert.assertTrue(isValidSignature);
    }
```

11.10. 3rd party integration

Common scenario is to use Picketlink as both Identity Provider (IDP) and Service Provider (SP), but sometimes it may be useful to integrate with 3rd party vendors as well. If your company is using services provided by 3rd party vendors like SalesForce or Google Apps, then SSO with these vendors may be real benefit for you.

We support these scenarios:

- Picketlink as IDP, Salesforce as SP [https://docs.jboss.org/author/display/PLINK/Picketlink+as+IDP%2C+Salesforce+as+SP]
- Picketlink as IDP, Google Apps as SP [https://docs.jboss.org/author/display/PLINK/Picketlink+as+IDP%2C+Google+Apps+as+SP]

Picketlink as IDP, Salesforce as SP

- Picketlink as SP, Salesforce as IDP [https://docs.jboss.org/author/display/PLINK/Picketlink+as+SP%2C+Salesforce+as+IDP]

11.10.1. Picketlink as IDP, Salesforce as SP

In first scenario we will use Salesforce as SAML SP and we will use Picketlink application as SAML IDP. In this tutorial, we will reuse application **idp-sig.war** from Picketlink quickstarts [https://docs.jboss.org/author/display/PLINK/PicketLink+Quickstarts#PicketLinkQuickstarts-AbouttheQuickstarts] .

NOTE: Integration is working from Picketlink version 2.1.2.Final and newer

11.10.1.1. Salesforce setup

First you need to perform some actions on Salesforce side. Brief description is here. For more details, you can see Salesforce documentation.

- **Register Salesforce account** - You will need to register in Salesforce with free developer account. You can do it here [http://developer.force.com/] .
- **Register your salesforce domain** - Salesforce supports SP-initiated SAML login workflow or IDP-initiated SAML login workflow. For picketlink integration, we will use SP-initiated login workflow, where user needs to access Salesforce and Salesforce will send SAMLRequest to Picketlink IDP. For achieving this, you need to create Salesforce domain. When registered and logged in www.salesforce.com [http://www.salesforce.com] , you will need to click on Your name in right top corner -> Link **Setup** -> Link **Company Profile** -> Link **My Domain** . Here you can create your Salesforce domain and make it available for testing.
- **SAML SSO configuration** - Now you need again to click on Your name in right top corner -> Link **Setup** -> Link **Security controls** -> Link **Single Sign-On Settings** Then configure it as follows:
  - **SAML Enabled** checkbox needs to be checked
  - **SAML Version** needs to be 2.0
  - **Issuer** needs to be *http://localhost:8080/idp-sig/ [http://localhost:8080/idp-sig/_]* - This identifies issuer, which will be used as IDP for salesforce. NOTE: Be sure that URL really ends with "/" character.
  - **Identity Provider Login URL** also needs to be http://localhost:8080/idp-sig/ [http://localhost:8080/idp-sig/_] - This identifies URL where Salesforce SP will send it's SAMLRequest for login.
  - **Identity Provider Logout URL** points to URL where Salesforce redirects user after logout. You may also use your IDP address or something different according to your needs.

- **Subject mapping** - You need to configure how to map Subject from SAMLResponse, which will be send by Picketlink IDP, to Salesforce user account. In the example, we will use that SAMLResponse will contain information about Subject in "NameIdentifier" element of SAMLResponse and ID of subject will be mapped to Salesforce Federation ID of particular user. So in: **SAML User ID Type** , you need to check option *Assertion contains the Federation ID from the User object* and for **SAML User ID Location** , you need to check *User ID is in the NameIdentifier element of the Subject statement* .

- **Entity ID** - Here we will use https://saml.salesforce.com [https://saml.salesforce.com_] . Whole configuration can look as follows:

Single Sign-On Settings

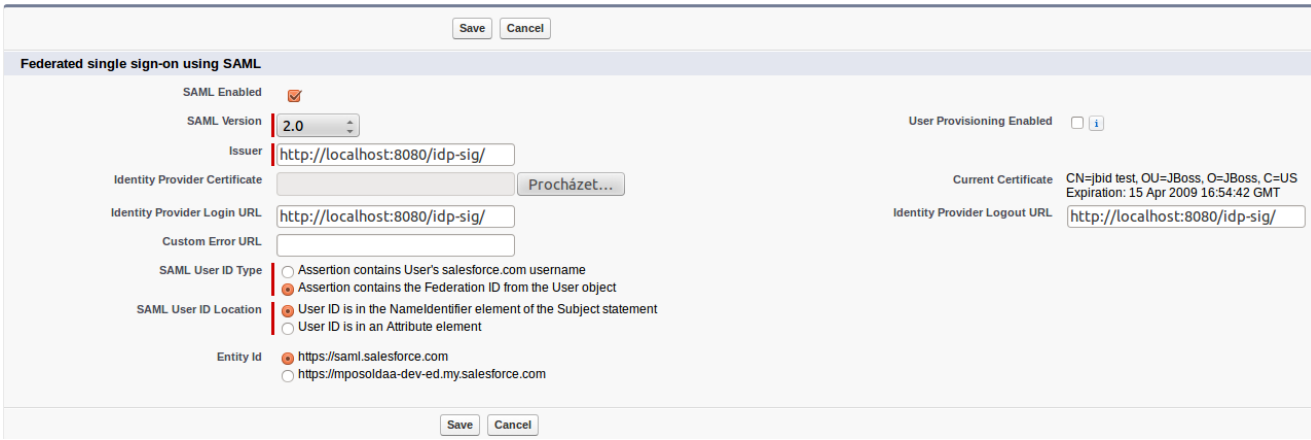


Figure 11.5. TODO InformalFigure image title empty

- **Certificate** - Last very important thing is upload of your certificate to Salesforce, because Salesforce needs to verify signature on SAMLResponse sent from your Picketlink Identity Provider. So first you need to export certificate from your keystore file and then import this certificate into Salesforce. So in **idp-sig.war/WEB-INF/classes** you can run command like:

```
keytool -export -keystore jbid_test_keystore.jks -alias servercert -file test-certificate.crt
```

after typing keystore password *store123* you should see exported certificate in file *test-certificate.crt* .

> **WARNING:** For production environment in salesforce, you should generate your own keystore file and use certificate from your own file instead of the default picketlink *jbid_test_keystore.jks*

Then you can import this certificate *test-certificate.crt* into SalesForce via menu with SSO configuration.

Picketlink as IDP, Salesforce as SP

- **Adding users** - Last action you need to do in Salesforce is to add some users. You can do it in: Link **Setup** -> Link **Manage Users** -> Link **Users** . Now you can create user and fill some values as you want. Please note that username must be in form of email address. Note that *Federation ID* is the value, which is used for mapping the user with the NameIdentifier subject from SAML assertion, which will be sent from Picketlink IDP. So let's use Federation ID with value *tomcat* for our first user.

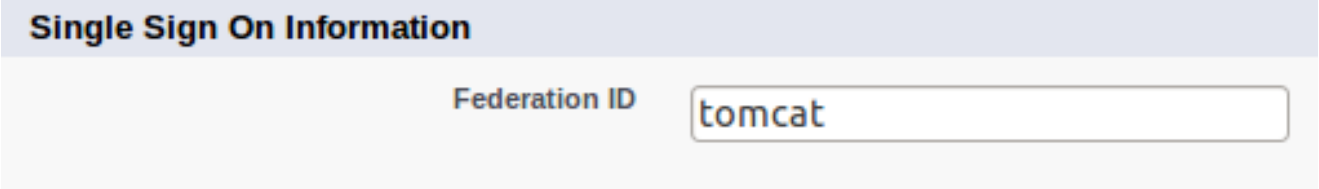


Figure 11.6. TODO InformalFigure image title empty

11.10.1.2. Picketlink IDP setup

- **Download and import Salesforce certificate** - SAMLRequest messages sent from Salesforce are signed with Salesforce certificate. In order to validate them, you need to download Salesforce client certificate from http://wiki.developerforce.com/page/Client_Certificate . Then you need to import the certificate into your keystore:

```
unzip -q /tmp/downloads/certificates/New_proxy.salesforce.com_certificate_chain.zip
keytool  -import  -keystore  jbid_test_keystore.jks  -file  proxy-salesforce-com.123  -alias
 salesforce-cert
```

- **ValidatingAlias update** - You need to update ValidatingAlias section, so the SAMLRequest from Salesforce will be validated with Salesforce certificate. You need to add the line into file **idp-sig.war/WEB-INF/picketlink.xml** :

```
<ValidatingAlias Key="saml.salesforce.com" Value="salesforce-cert" />
```

- **Trusted domain** - update list of trusted domains and add domain "salesforce.com" to the list:

```
<Trust>
  <Domains>localhost,jboss.com,jboss.org,redhat.com,amazonaws.com,salesforce.com</Domains>
</Trust>
```

11.10.1.2.1. Picketlink as IDP, Salesforce as SP

11.10.1.2.2. Single logout

Now you have basic setup done but in order to support single logout, you need to do some additional actions. Especially Salesforce is not using same URL for login and single logout, which means that we need to configure SP metadata on Picketlink side to provide mapping between SP and their URL for logout. Needed actions are:

- **Download SAML metadata** from Salesforce SSO settings. Save downloaded XML file as **idp-sig.war/WEB-INF/sp-metadata.xml**

- **Add SingleLogoutService element** - unfortunately another element needs to be manually added into metadata as Salesforce doesn't use single logout configuration in their metadata. So let's add following element into metadata file after *md:AssertionConsumerService* element:

```
        <md:SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
                                Location="https://login.salesforce.com/saml/logout-request.jsp?
saml=MgoTx78aEPkEM4eGV5ZzptlliwIVkRkOWYKlqXQq2StV_sLo0EiRqKYtIc" index="0" isDefault="true"/>
```

Note that value of Location attribute will be different for your domain. You can see which value to use in Salesforce SSO settings page from element *Salesforce.com Single Logout URL* :

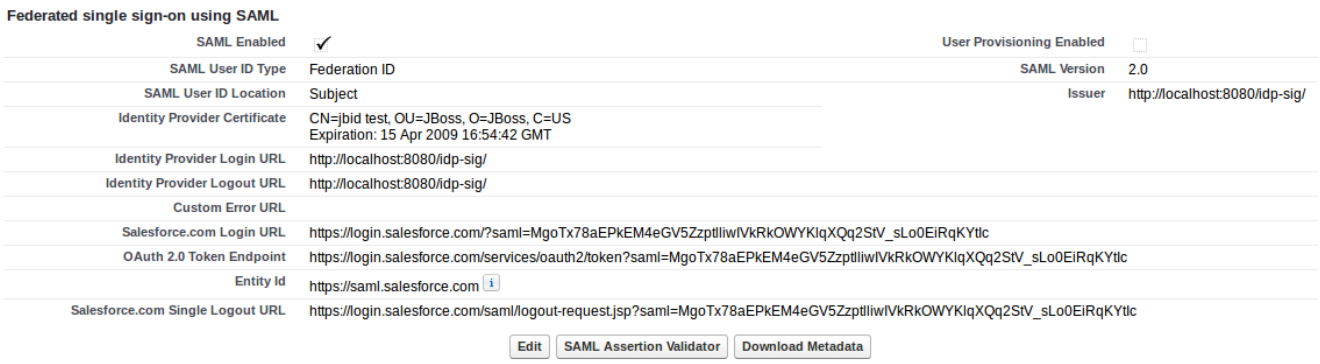


Figure 11.7. TODO InformalFigure image title empty

- **Add md:EntitiesDescriptor element** - Finally you need to add enclosing element *md:EntitiesDescriptor* and encapsulate whole current content into it. This is needed as we may want to use more EntityDescriptor elements in this single metadata file (like another element for Google Apps etc):

```
```

Picketlink
as
IDP,

```
<?xml version="1.0" encoding="UTF-8"?>
<md:EntitiesDescriptor xmlns="urn:oasis:names:tc:SAML:2.0:metadata"
                     xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
                     xmlns:md="urn:oasis:names:tc:SAML:2.0:metadata">
     <md:EntityDescriptor  xmlns:md="urn:oasis:names:tc:SAML:2.0:metadata"  entityID="https://
saml.salesforce.com" ....
  ...
  </md:EntityDescriptor>
</md:EntitiesDescriptor>
```

- **Configure metadata location** - Let's add new MetaDataProvider into file **idp-sig.war/WEB-INF/picketlink.xml** after section with KeyProvider:

```
...
         </KeyProvider>

                                                                          <MetaDataProvider
 ClassName="org.picketlink.identity.federation.core.saml.md.providers.FileBasedEntitiesMetadataProvider">
           <Option Key="FileName" Value="/WEB-INF/sp-metadata.xml"/>
         </MetaDataProvider>
      </PicketLinkIDP>
.....
```

11.10.1.3. Test the setup

- Start the server with Picketlink IDP
- Visit URL of your salesforce domain. It should be likely something like: https://yourdomain.my.salesforce.com/ . Now Salesforce will send SAMLRequest to your IDP and so you should be redirected to login screen on your IDP on http://localhost:8080/idp-sig/
- Login into Picketlink IDP as user *tomcat* . After successful login, SAMLRequest signature is validated by the certificate *salesforce-cert* and IDP produces SAMLResponse for IDP and performs redirection.
- Now Salesforce parse SAMLResponse, validates it signature with imported Picketlink certificate and then you should be redirected to salesforce and logged as user *tomcat* in your Salesforce domain.

11.10.1.4. Troubleshooting

Salesforce provides simple tool in SSO menu, where you can see the status of last SAMLResponse sent to Salesforce SP and you can check what's wrong with the response here.

Good tool for checking communication between SP and IDP is also Firefox plugin SAML Tracer [https://addons.mozilla.org/en-US/firefox/addon/saml-tracer/]

11.10.2. Picketlink as SP, Salesforce as IDP

In this part, we will use Salesforce as IDP and sample application from Picketlink as SP.

NOTE: Integration is working from Picketlink version 2.1.2.Final and newer

11.10.2.1. Salesforce setup

- **Disable Single Sign on** in SSO settings if you enabled it previously. As in this step, we don't want to login into Salesforce through SSO but we want Salesforce to provide SSO for us and act as Identity Provider.

- **Identity provider setup** - In link **Setup** -> **Security controls** -> **Identity provider** you need to setup Salesforce as IDP.

- **Generate certificate** - first generate certificate on first screen. This certificate will be used to sign SAMLResponse messages sent from Salesforce IDP.

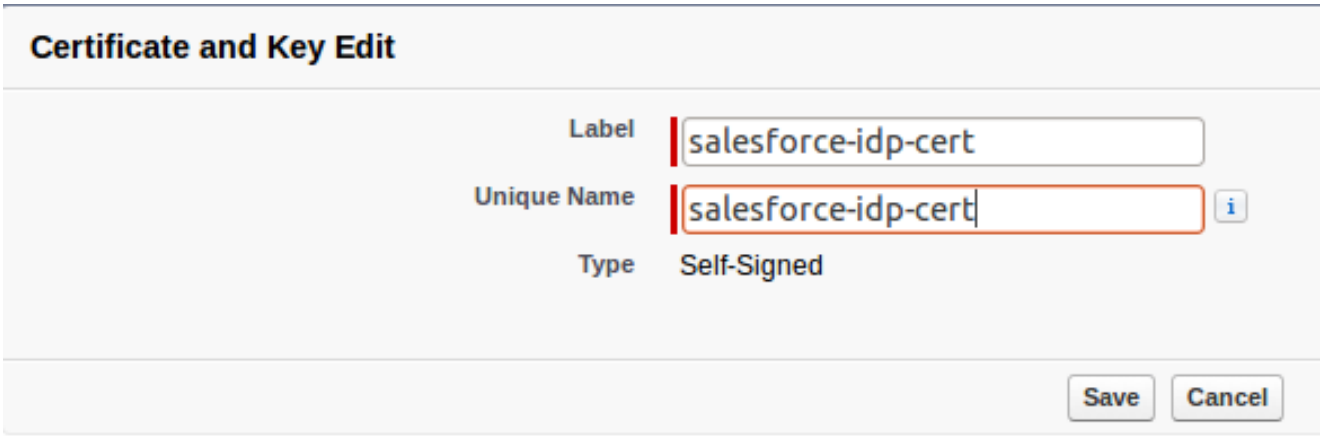


Figure 11.8. TODO InformalFigure image title empty

After certificate will be generated in Salesforce, you can download it to your computer.

- **Configure generated certificate for Identity Provider** - In Identity Provider setup, you need to select the certificate, which you just generated

- **Add service provider** - In section **Setup** -> **Security Controls** -> **Identity Provider** -> **Service providers** you can add your Picketlink application as Service Provider. We will use application **sales-post-sig** from Picketlink quickstarts [https://docs.jboss.org/author/display/

PLINK/PicketLink+Quickstarts#PicketLinkQuickstarts-AbouttheQuickstarts] . So in first screen of configuration of your Service provider, you need to add **ACS URL** and **Entity ID** like *http://localhost:8080/sales-post-sig/* . *Subject type* needs to be *Federation ID* and you also need to upload certificate corresponding to signing key of sales-post-sig application. You first need to export this certificate from your keystore file. See previous tutorial [https://docs.jboss.org/author/display/PLINK/Picketlink+as+IDP%2C+Salesforce+as+SP] for how to do it. In next screen, you can select profile for users, who will be able to login to this Service Provider. By checking first checkbox, you will automatically select all profiles. After confirm this screen, you will have your service provider created. Let's see how your final configuration can looks like after confirming:

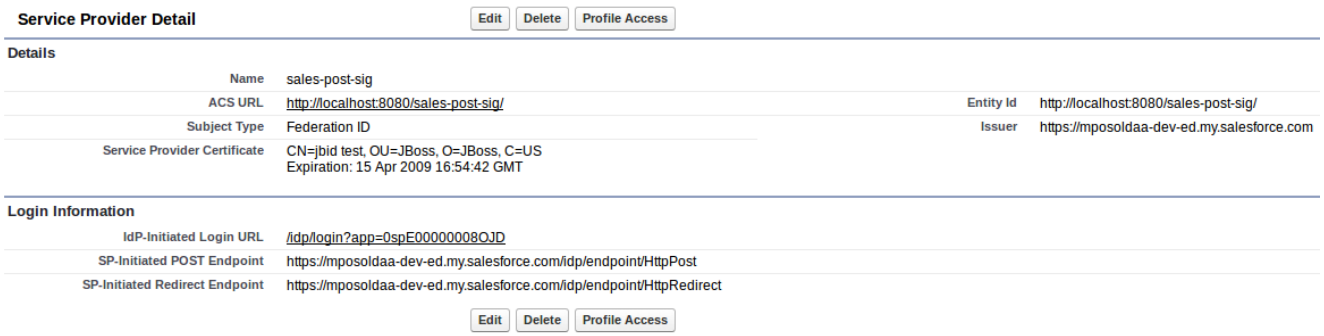


Figure 11.9. TODO InformalFigure image title empty

> **WARNING:** As mentioned in previous tutorial, you should create your own keystore file for Picketlink and not use example keystore *jbid_test_keystore.jks* and certificates from it in production environment. In this tutorial, we will use it only for simplicity and demonstration purposes.

11.10.2.2. Picketlink Setup

As already mentioned, we will use sample application *sales-post-sig.war* from picketlink quickstarts.

- **Import salesforce IDP certificate** - In **sales-post-sig.war/WEB-INF/classes** you need to import downloaded certificate from salesforce into your keystore. You can use command like:

```
keytool -import -file salesforce_idp_cert.cer -keystore jbid_test_keystore.jks -alias
 salesforce-idp
```

- **Identity URL configuration** - In **sales-post-sig.war/WEB-INF/picketlink.xml** you need to change identity URL to something like:

Picketlink
as
SP,

```
<IdentityURL>${idp-sig.url::https://yourdomain.my.salesforce.com/idp/endpoint/HttpPost}
```

- **ValidatingAlias configuration** - In same file, you can add new validating alias for the salesforce host of your domain:

```
<ValidatingAlias Key="yourdomain.my.salesforce.com" Value="salesforce-idp" />
```

- **Roles mapping** - Last very important step is mapping of roles for users, which are logged through Salesforce IDP. Normally when you have Picketlink as both IDP and SP, then SAMLResponse from IDP usually contains *AttributeStatement* as part of SAML assertion and this statement contains list of roles in attribute *Role* . Picketlink SP is then able to parse list of roles from statement and then it leverages *SAML2LoginModule* to assign these roles to JAAS Subject of logged principal. Thing is that SAML Response from Salesforce IDP does not contain any attribute statement with roles, so you need to handle roles assignment by yourself. Easiest way could be to chain SAML2LoginModule with another login module (like *UsersRolesLoginModule* for instance), which will ensure that assigning of JAAS roles is delegated from SAML2LoginModule to the second Login Module in chain. Needed steps:
  - In **sales-post-sig.war/WEB-INF/jboss-web.xml** you can change security-domain from value *sp* to something different like *sp-salesforce*

```
<security-domain>sp-salesforce</security-domain>
```

- Create new application policy for this security domain. It differs in each application server, for example in JBoss 7 you need to edit **JBOSS_HOME/standalone/configuration/standalone.xml** and add this policy to particular section:

```
<security-domain name="sp-salesforce" cache-type="default">
  <authentication>
                                                                    <login-module
 flag="required">
      <module-option name="password-stacking" value="useFirstPass"/>
    </login-module>
    <login-module code="org.jboss.security.auth.spi.UsersRolesLoginModule" flag="required">
      <module-option name="password-stacking" value="useFirstPass"/>
      <module-option name="usersProperties" value="users.properties"/>
```

```
      <module-option name="rolesProperties" value="roles.properties"/>
    </login-module>
  </authentication>
</security-domain>
```

- In **sales-post-sig.war/WEB-INF/classes** you need to create empty file **users.properties** and non-empty file **roles.properties** where you need to map roles. For example you can add line like:

```
tomcat=manager,employee,sales
```

where **tomcat** is Federation ID of some user from Salesforce, which you will use for login.

11.10.2.3. Test the integration

Now after server restart, let's try to access: http://localhost:8080/sales-post-sig/ . You should be redirected to salesforce login page with SAMLRequest sent from your Picketlink sales-post-sig application. Now let's login into Salesforce with username and password of some Salesforce user from your domain (like tomcat user). Make sure that this user has Federation ID and this Federation ID is mapped in file roles.properties on Picketlink SP side like described in previous steps. Now you should be redirected to http://localhost:8080/sales-post-sig/ as logged user.

11.10.3. Picketlink as IDP, Google Apps as SP

Google Apps is another known business solution from Google. Google Apps supports SAML SSO in role of SAML SP, so you need to use your own application as SAML IDP. In this sample, we will again use *idp-sig.war* application from Picketlink quickstarts as IDP similarly like in this tutorial [https://docs.jboss.org/author/display/PLINK/Picketlink+as+IDP%2C+Salesforce+as+SP] .

> NOTE: Integration is working from Picketlink version 2.1.2.Final and newer

11.10.3.1. Google Apps setup

- **Creating Google Apps domain** - you need to create Google Apps domain on http://www.google.com/apps . Follow the instructions on google page on how to do it.
- **Add some users** - let's add some users, which will be available to login into your domain. So let's add user *tomcat* first. In Google & Apps control panel, you need to click **Organization & Users** -> **Create new user** and add him email **tomcat@yourdomain.com** . This will ensure that nick of new user will be *tomcat* . See screenshot:

Picketlink
as
IDP,

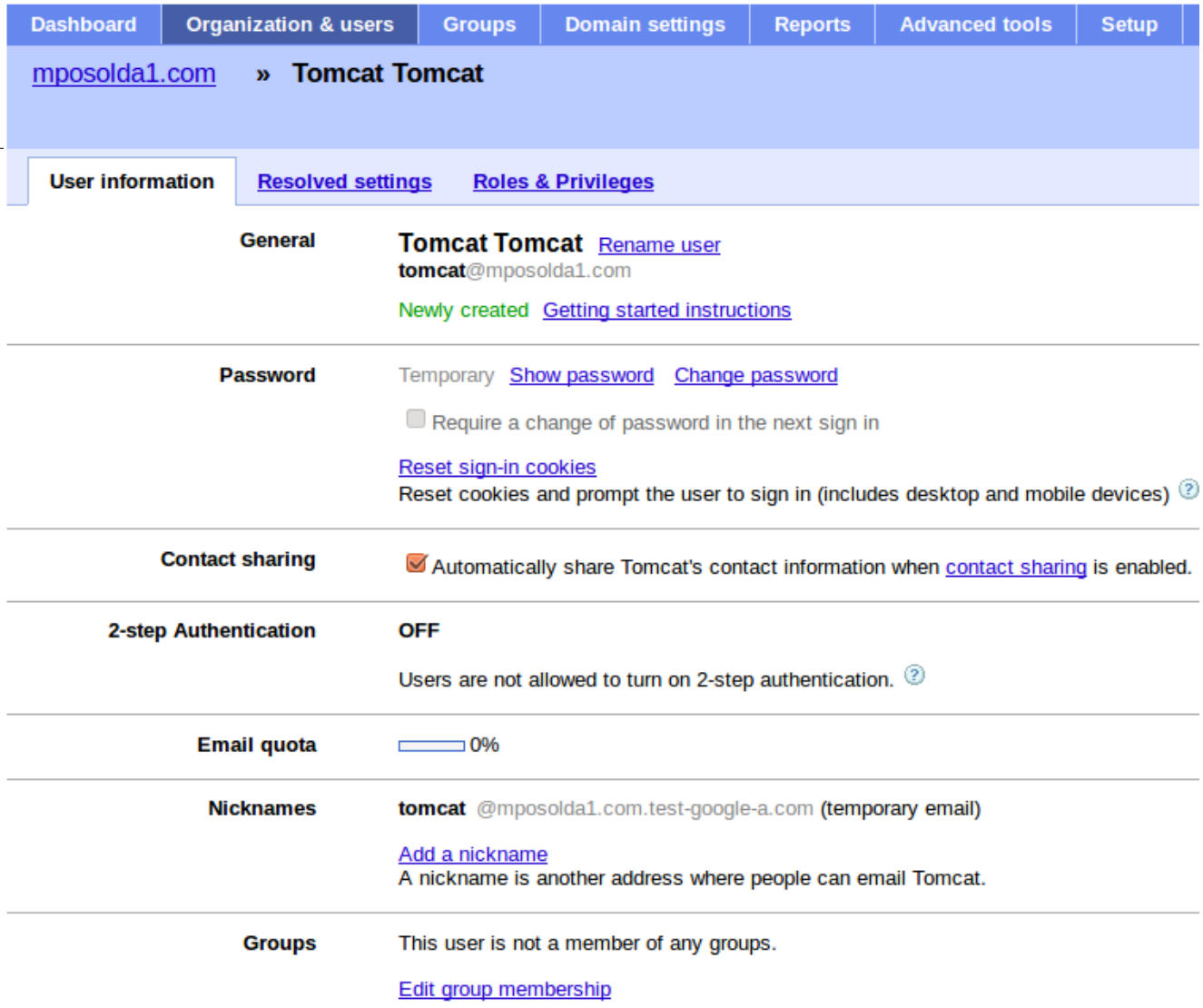


Figure 11.10. TODO InformalFigure image title empty

- **Configure SAML SSO** - In menu **Advanced tools** -> **Set up single sign-on (SSO)** you can setup SSO settings. For our testing purposes, you can set it like done on screenshot . Especially it's important to set Sign-in page to *http://localhost:8080/idp-sig/* . *Sign-out page can be also set but Google Apps don't support SAML Single Logout profile, so this is only page where will be users redirected after logout. Let's click checkbox _Use a domain specific issuer* to true.

- **Certificate upload** - you also need to upload certificate exported from your picketlink keystore in similar way, like done for Salesforce in previous tutorials [https://docs.jboss.org/author/display/PLINK/Picketlink+as+IDP%2C+Salesforce+as+SP] . So let's upload *test-certificate.crt* into Google Apps.

Picketlink
as
IDP,

> WARNING: Once again, you shouldn't use picketlink test keystore file jbid_test_keystore.jks in production environment. We use it here only for simplicity and for demonstration purposes.

as

Figure 11.11. TODO InformalFigure image title empty

11.10.3.2. Picketlink IDP configuration

- **Trusted domains configuration** - update domains in **idp-sig.war/WEB-INF/picketlink.xml**

```
<Trust>
    <Domains>localhost,jboss.com,jboss.org,redhat.com,amazonaws.com,salesforce.com,google.com</
Domains>
</Trust>
```

- **Metadata configuration** - We don't want SAMLRequest from Google Apps to be validated, because it's not signed. So let's add another metadata for Google Apps, which will specify that SAMLRequest from Google Apps Service Provider won't be signed. So let's add another *EntityMetadataDescriptor* entry for your domain *google.com/a/yourdomain.com* into

sp-metadata.xml file created in previous tutorial [https://docs.jboss.org/author/display/PLINK/Picketlink+as+IDP%2C+Salesforce+as+SP] (you may need to create new metadata file from scratch if not followed previous tutorial). Important attribute is especially *AuthnRequestsSigned* , which specifies that SAMLRequest from Google Apps are not signed.

```
<md:EntitiesDescriptor xmlns="urn:oasis:names:tc:SAML:2.0:metadata"
                     xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
                     xmlns:md="urn:oasis:names:tc:SAML:2.0:metadata">
     <md:EntityDescriptor  xmlns:md="urn:oasis:names:tc:SAML:2.0:metadata"  entityID="https://
saml.salesforce.com" validUntil="2022-06-18T14:08:08.052Z">

......
  </md:EntityDescriptor>
   <md:EntityDescriptor xmlns:md="urn:oasis:names:tc:SAML:2.0:metadata" entityID="google.com/a/
yourdomain.com" validUntil="2022-06-13T21:46:02.496Z">
                <md:SPSSODescriptor   AuthnRequestsSigned="false"   WantAssertionsSigned="true"
 protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol" />
   </md:EntityDescriptor>
</md:EntitiesDescriptor>
```

11.10.3.3. Test it

Now logout from Google Apps and start server. And now you can do visit https://mail.google.com/a/yourdomain.com . After that Google Apps will send SAMLRequest and redirects you to http://localhost:8080/idp-sig . Please note that Google Apps is using SAML HTTP Redirect binding, so you can see SAMLRequest in browser URL. Also note that SAMLRequest is not signed, but this is not a problem as we configured it in metadata that requests from Google Apps are not signed. So after login into IDP as user tomcat, you should be automatically logged into your Google Apps as user "tomcat" as well.

Chapter 12. PicketLink Quickstarts

12.1. Overview

Quickstarts are self-contained, concise examples that generally demonstrate at most one or two features. The PicketLink quickstarts at GitHub provide working, buildable code that shows the usage of a number of authentication, authorization and identity management features. They are a nice way to communicate the design, common and best practices and the usage of PicketLink.

The PicketLink Quickstarts are part of the JBoss Developer Framework(http://www.jboss.org/jdf/quickstarts/get-started/ [http://www.jboss.org/jdf/quickstarts/get-started/]), formerly known as JDF. There you can find a lot of useful stuff about Java, JavaEE and of course all JBoss projects and products. Also, there are a lot of useful information and tutorials that will guide you to configure and prepare your environment to start using any of the available quickstarts.

All quickstarts are available at GitHub. So you can download them, fork the repository (if you have an GitHub account, of course) or even just clone the repository. The latter option is fully described along the README.md file for each quickstart, as well as a lot of additional information, configuration requirements, how to deploy and undeploy using the JBoss Enterprise Application Platform 6(and beyond) and so forth.

The repository is located at https://github.com/jboss-developer/jboss-picketlink-quickstarts [https://github.com/jboss-developer/jboss-picketlink-quickstarts].

> **Note**
>
> You don't need to be a Git expert in order to get the quickstarts. All the necessary commands are fully covered in the README.md file for each of them. For any additional information take a look at the JDF site too.

12.2. Available Quickstarts

For a complete list of all available quickstarts, access the PicketLink Quickstart Repository.

12.3. PicketLink Federation Quickstarts

We have a plenty of quickstarts covering some of the most important aspects of PicketLink Federation: SAML based SSO, WS-Trust support and so forth. The repository for those quickstarts is located at https://github.com/picketlink2/picketlink-quickstarts [https://github.com/picketlink2/picketlink-quickstarts].

The best way to get those quickstarts up and running is using the Section 1.6, “PicketLink Installer”. Usually, when using PicketLink Federation you don't ship the libraries inside your deployment, but you get them from the container where it is deployed. That is exactly what the installer does (among

other things), it will prepare your JBoss EAP installation with the PicketLink JBoss Modules. The installer also performs some additional configuration to an existing JBoss EAP 6 installation in order to get the quickstarts up and running, such as changing your standalone.xml and deploying the example applications. Is just a matter of run the installer, start the server and start using the examples !

> **Note**
>
> Users of PicketLink 2.1 series should start using the 2.5 version of the federation libraries. Once JBoss EAP is updated with the new module organization for 2.5 version, the 2.1 series will be deprecated.

12.4. Contributing

PicketLink can be used to solve the most simple as well some of the most advanced security use cases. The main objective of the quickstarts is to cover most of use cases as we can, so people can quickly understand and start solving their own security-related problems based on what we're offering.

We would be very glad to have your contribution to our quickstarts list. If you have any suggestion about a new example, please create a JIRA and describe what you're looking for: https://issues.jboss.org/browse/PLINK [https://issues.jboss.org/browse/PLINK].

You can also contribute by sending a Pull Request to the PicketLink Quickstarts repository which the code of your example application. Just follow the Contributing Guidelines from http://site-jdf.rhcloud.com/quickstarts/get-involved/ [http://site-jdf.rhcloud.com/quickstarts/get-involved/].

Glossary

This chapter lists the glossary of terms used throughout the PicketLink Reference Documentation.

A

API	Application Programming Interface - A set of interfaces and classes designed for direct usage by the developer.

C

CDI	Contexts and Dependency Injection (JSR-299)