PicketLink Reference Documentation

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

PicketLink Reference Documentation

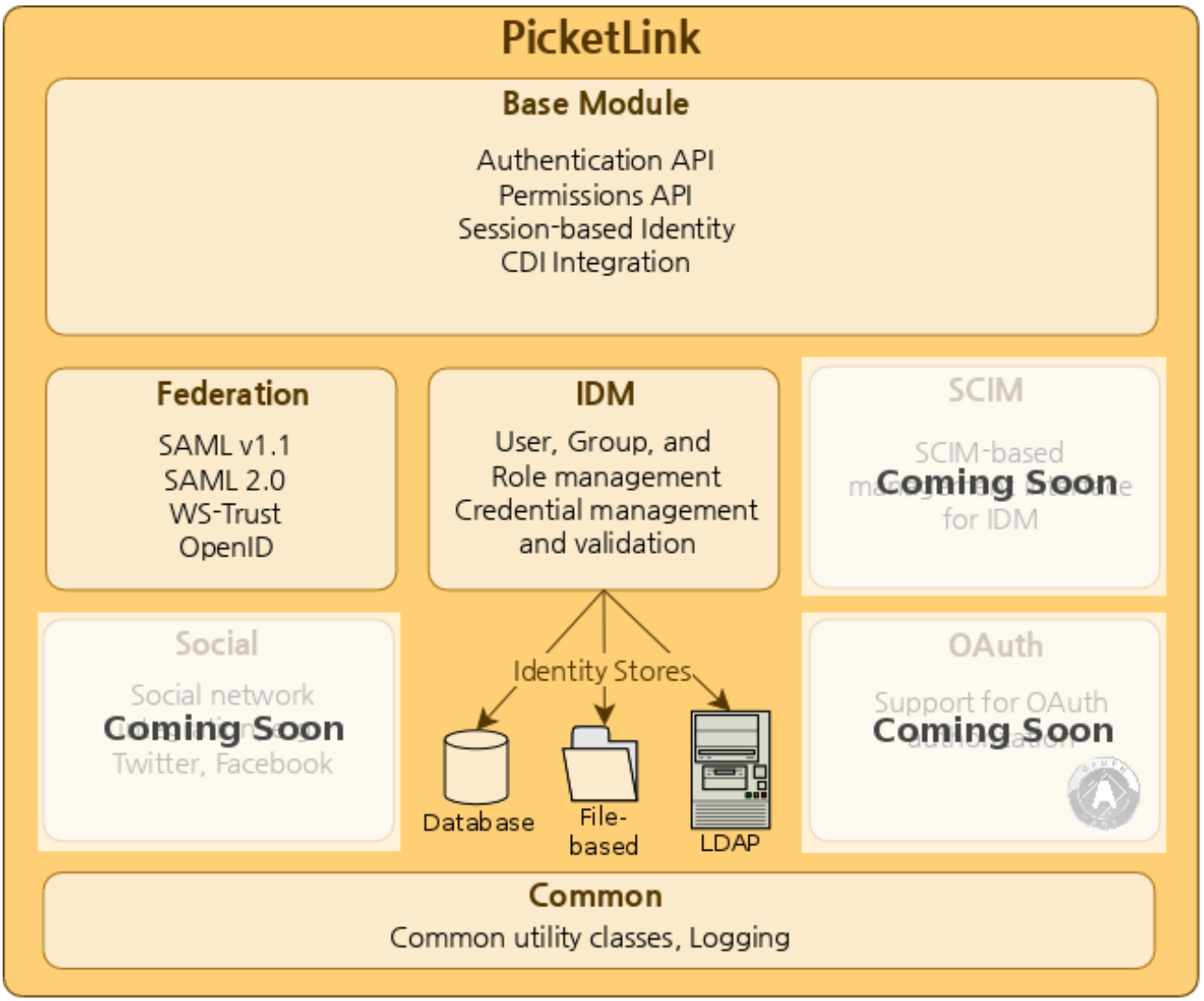
by

Version 2.5.0.Beta4

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.



1.2. Modules

1.2.1. 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 also the "glue" that integrates all of the PicketLink modules together to provide a cohesive API.

The base module libraries are as follows:

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

1.2.2. 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.

1.2.3. 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.

1.3. 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].

1.4. Maven Dependencies

The PicketLink libraries are available from the Maven Central Repository. To use PicketLink in your Maven-based project, it is recommended that you first define a version property for PicketLink in your project's `pom.xml` file like so:

```
<properties>
```

```
  <picketlink.version>2.5.0.Beta4</picketlink.version>
</properties>
```

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

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

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

The identity management library is a required dependency of the base module and so will be automatically included.

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-schema</artifactId>
    <version>${picketlink.version}</version>
  </dependency>
```

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. The Authentication API

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();

Agent getAgent();
```

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 will return a result of `SUCCESS`, otherwise it will return a result of `FAILED`. By default, the `Identity` bean is session-scoped, 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 `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 `getAgent()` method can be used to retrieve the currently authenticated agent (or user). If the current session is not authenticated, then `getAgent()` 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.agent.loginName}
```

> **Note**
>
> If you're wondering what an `Agent` is, it is simply a representation of the external entity that is interacting with your application, whether that be a human user or some third party (non-human) system. The `Agent` interface is actually the superclass of `User` - 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:

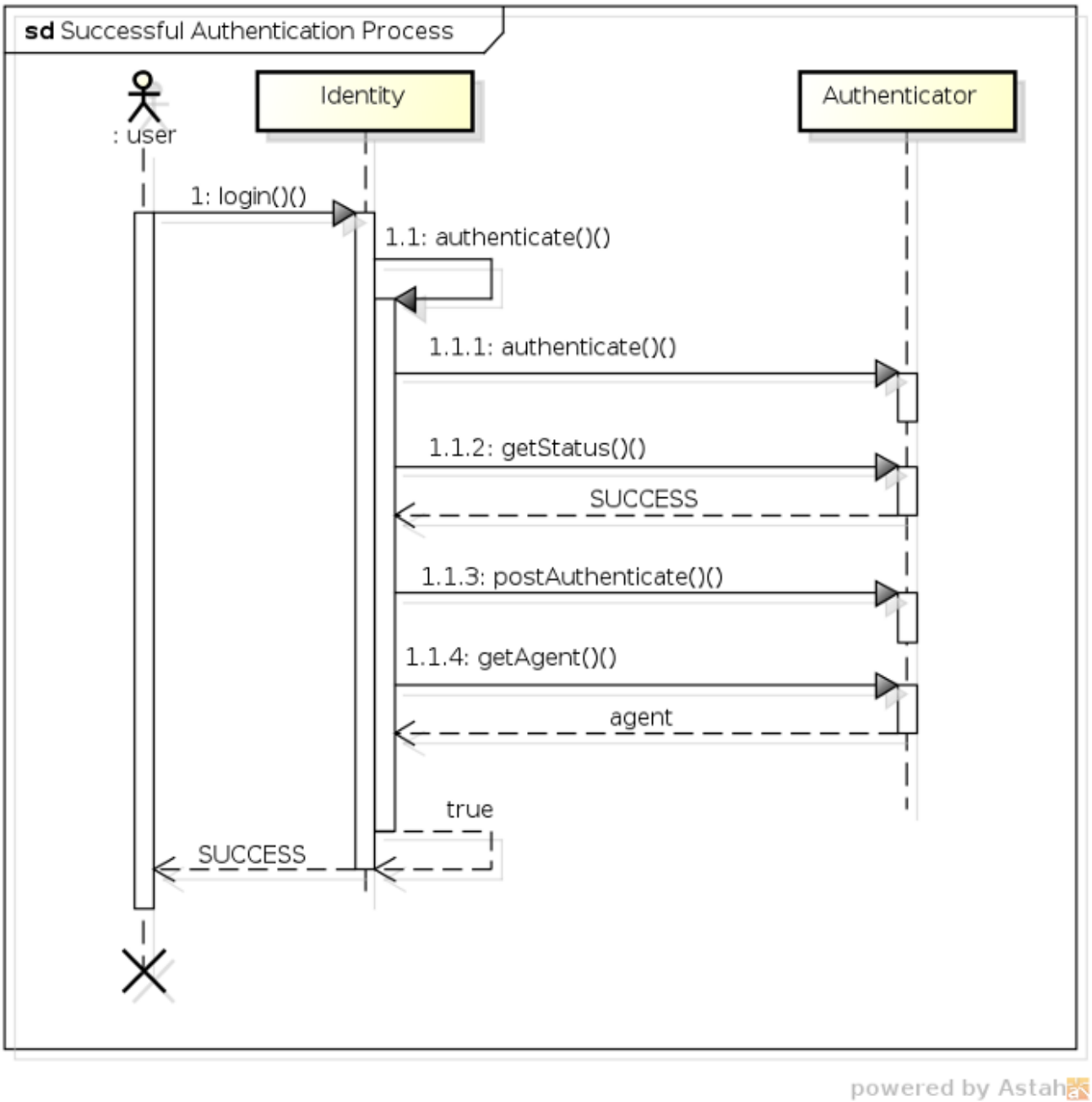
```
public interface Authenticator {
    public enum AuthenticationStatus {
        SUCCESS,
        FAILURE,
        DEFERRED
    }

    void authenticate();

    void postAuthenticate();

    AuthenticationStatus getStatus();

    Agent getAgent();
}
```

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 `getAgent()` method will return the authenticated `Agent` object and the `postAuthenticate()` method will be invoked also. If the authentication was not a success, `getAgent()` 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 `getAgent()` method, which returns an `Agent` 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);
            setUser(new SimpleUser("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 `getAgent()` methods (while also providing matching `setStatus()` and `setAgent()` 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 `setUser()` method is used to set the user (in this case by creating a new instance of `SimpleUser`). 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; }
}
```

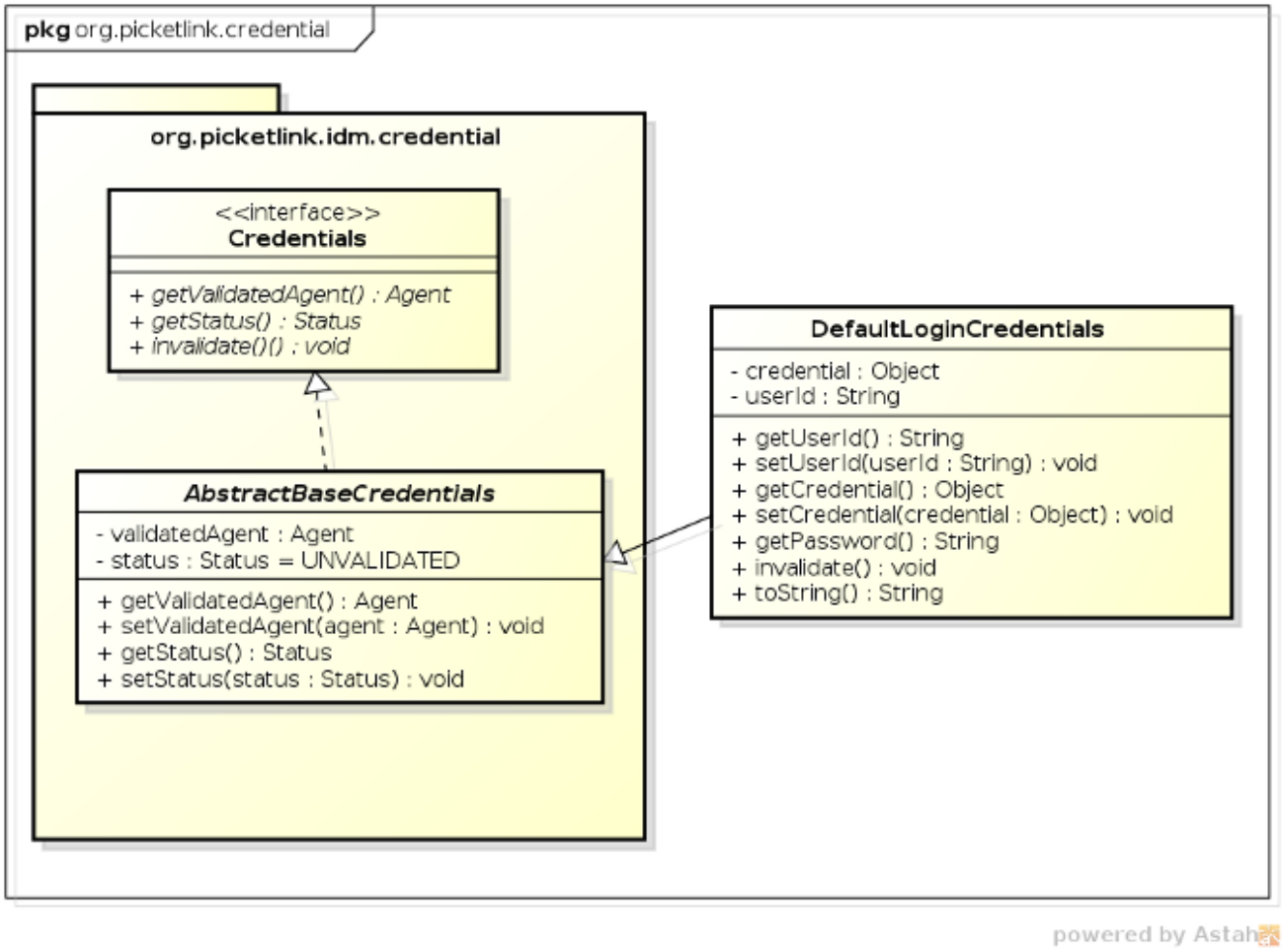
And then inject that into the `Authenticator` bean instead:

```
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`:

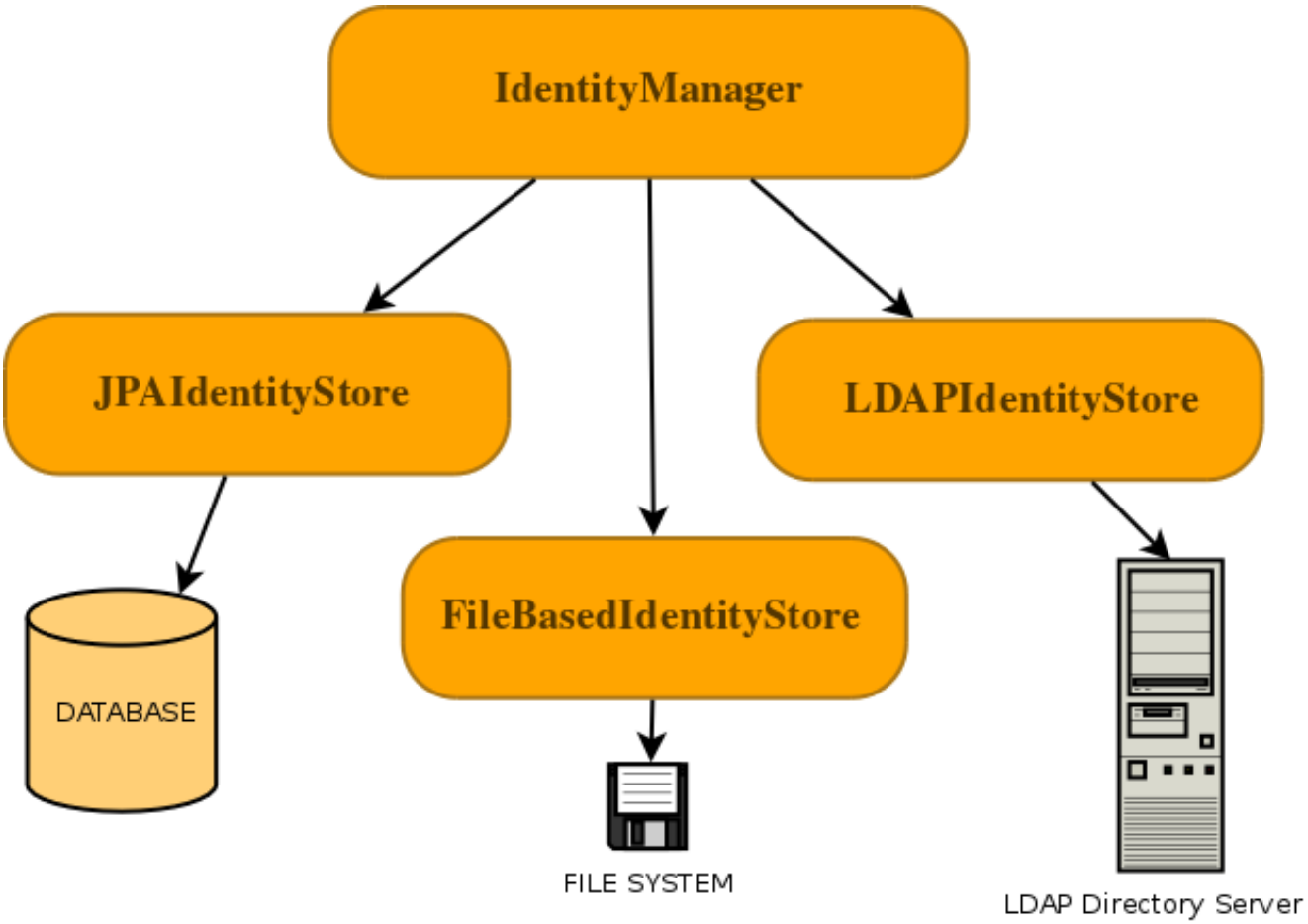
```
<div class="loginRow">
  <h:outputLabel for="name" value="Username" styleClass="loginLabel"/>
  <h:inputText id="name" value="#{loginCredentials.userId}"/>
</div>

<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

3.1. Overview

PicketLink's Identity Management (IDM) features provide a rich and extensible API for managing the users, groups and roles of your applications and services. The `org.picketlink.idm.IdentityManager` interface declares all the methods required to create, update and delete Identity objects and create relationships between them such as group and role memberships.

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.



3.2. Getting Started in 5 Minutes

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 `IdentityManager` bean into your own classes and start using it immediately:

```
@Inject IdentityManager identityManager;
```

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

```
User user = new SimpleUser("jane");
user.setFirstName("Jane");
user.setLastName("Doe");
identityManager.add(user);

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

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

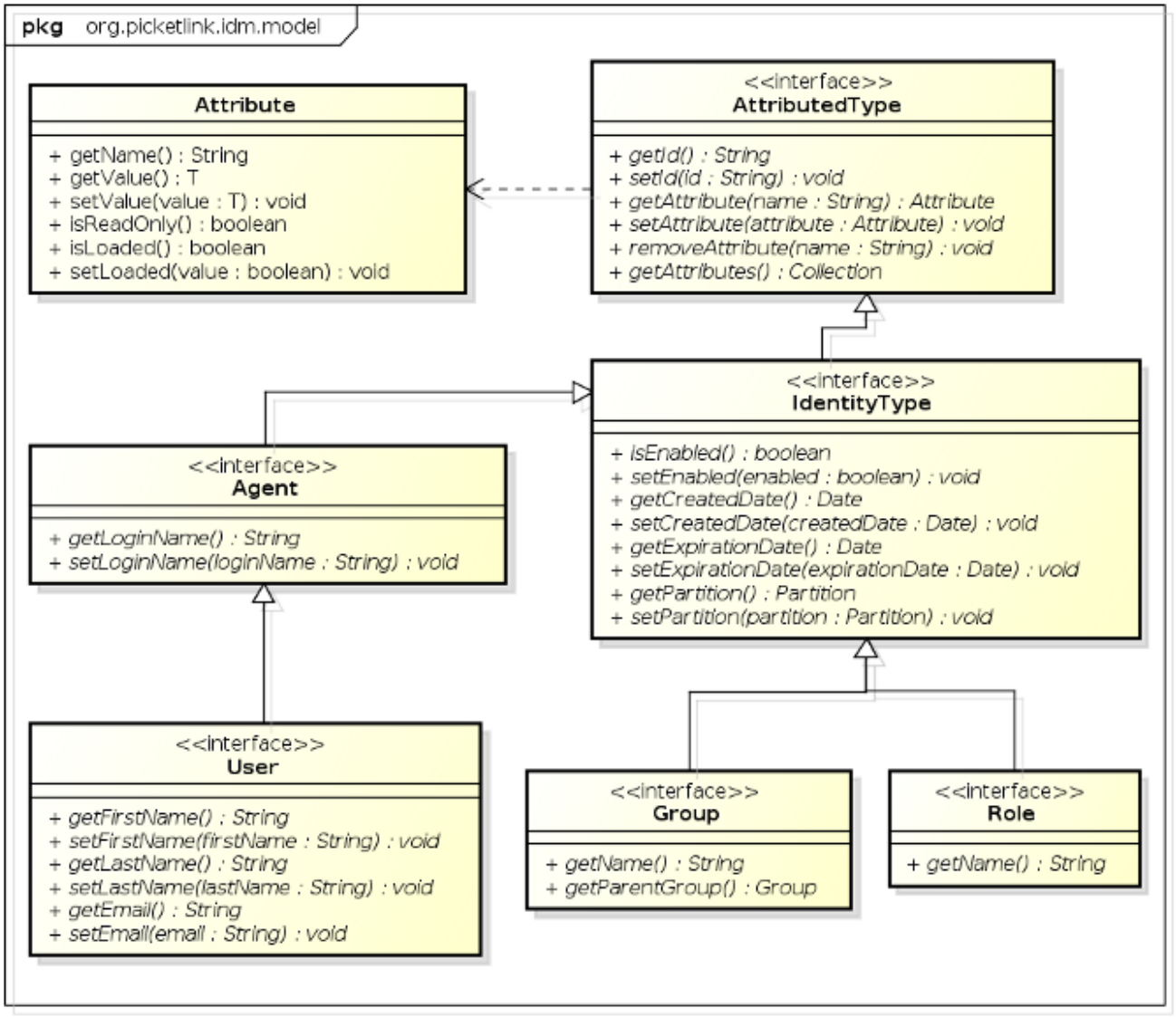
It is also quite easy to override the default behaviour and provide your own configuration - simply provide a producer method in your application that returns an `IdentityConfiguration` object:

```
@Produces
public IdentityConfiguration createIdentityConfiguration() {
    IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();
    builder
        .stores()
            .file()
                .supportAllFeatures();
    return builder.build();
}
```

The code above essentially produces a configuration identical to the default configuration. If you'd like to use LDAP or JPA to store your identity data, you'll also find example configurations later in this chapter.

3.3. Identity Model

PicketLink's identity model consists of a number of core interfaces that define the fundamental identity types upon which much of the Identity Management API is based. The following class diagram shows the classes and interfaces in the `org.picketlink.idm.model` package that form the base identity model.



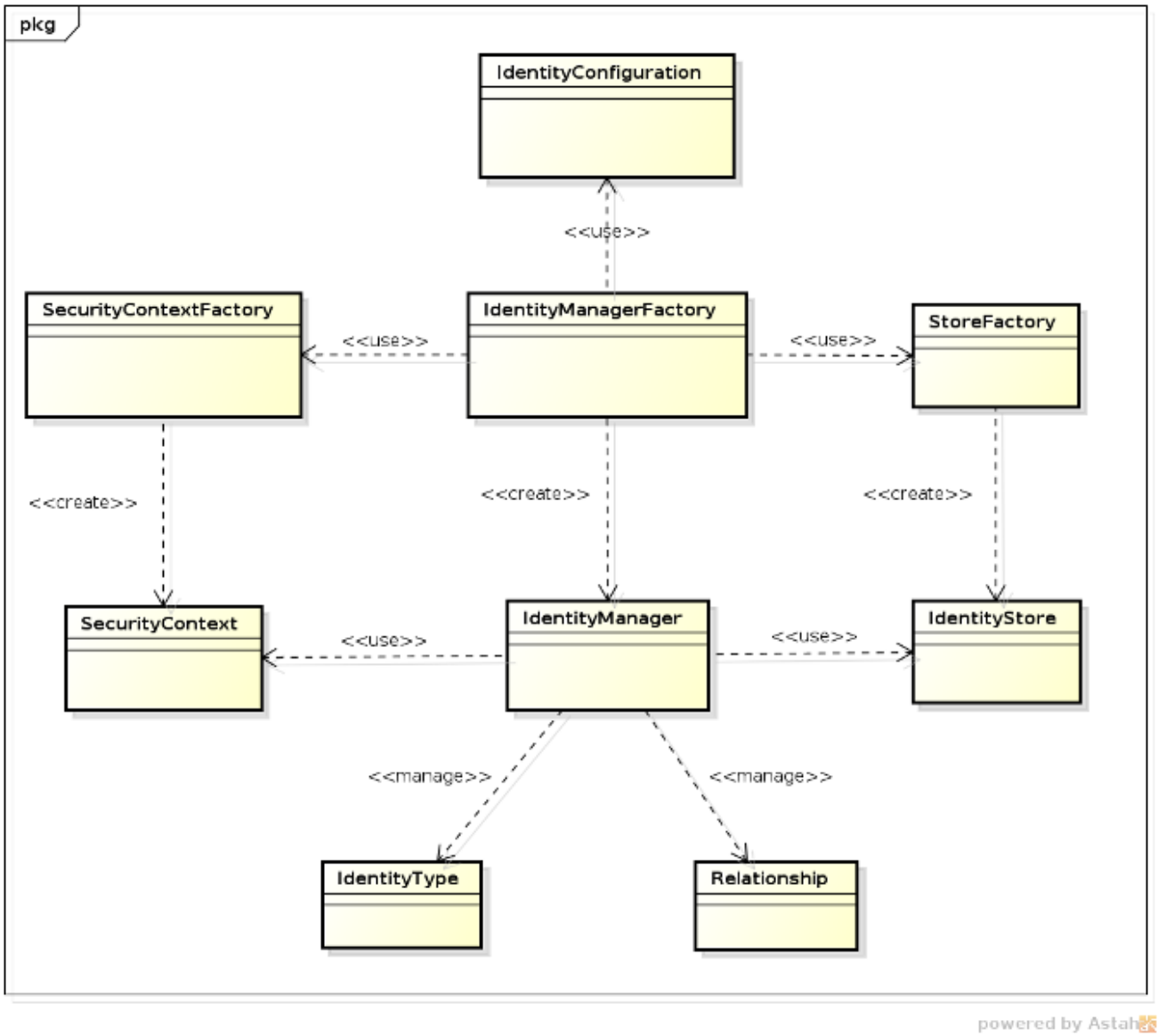
- `AttributedType` is the base interface for the identity model. It declares a number of methods for managing a set of attribute values, plus `getId()` and `setId()` methods for setting a unique UUID value.
- `Attribute` is used to represent an attribute value. An attribute has a name and a (generically 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.
- `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`.
- `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.

3.3.1. Architectural Overview

The following diagram shows the main components that realize PicketLink Identity Management:



- `IdentityConfiguration` is the the class responsible for holding all PicketLink configuration options. This class is usually built using the Configuration Builder API, which we'll cover in the next sections. Once created and populated with the configuration options, an instance is used to create a `IdentityManagerFactory`.
- `IdentityManagerFactory` is the class from which `IdentityManager` instances are created for a specific *realm*, considering all configurations provided by a `IdentityConfiguration` instance.
- `SecurityContextFactory` is an interface that provides methods for creating `SecurityContext` instances. This component knows how to properly create and prepare the context that will be propagated during identity management operations.

- `SecurityContext` is the class that holds context data that will be used during the execution of identity management operations. Once created, the context is used to create `IdentityStore` instances and to invoke their methods.

  This component allows to share data between the `IdentityManager` and `IdentityStore` instances. And also provides direct access for some IDM subsystems such as: event handling, caching and so on.

  Beyond that, this component is critical when access to external resources are required, such as the current `EntityManager` when using a JPA-based store.

  Each `IdentityManager` instance is associated with a single `SecurityContext`.

- `StoreFactory` is an interface that provides methods for creating `IdentityStore` instances. Instances are created considering the *Feature Set* supported by each identity store and also the current `SecurityContext` instance.

- `IdentityStore` is an interface that provides a contract for implementations that store data using a specific repository such as: LDAP, databases, file system, etc.

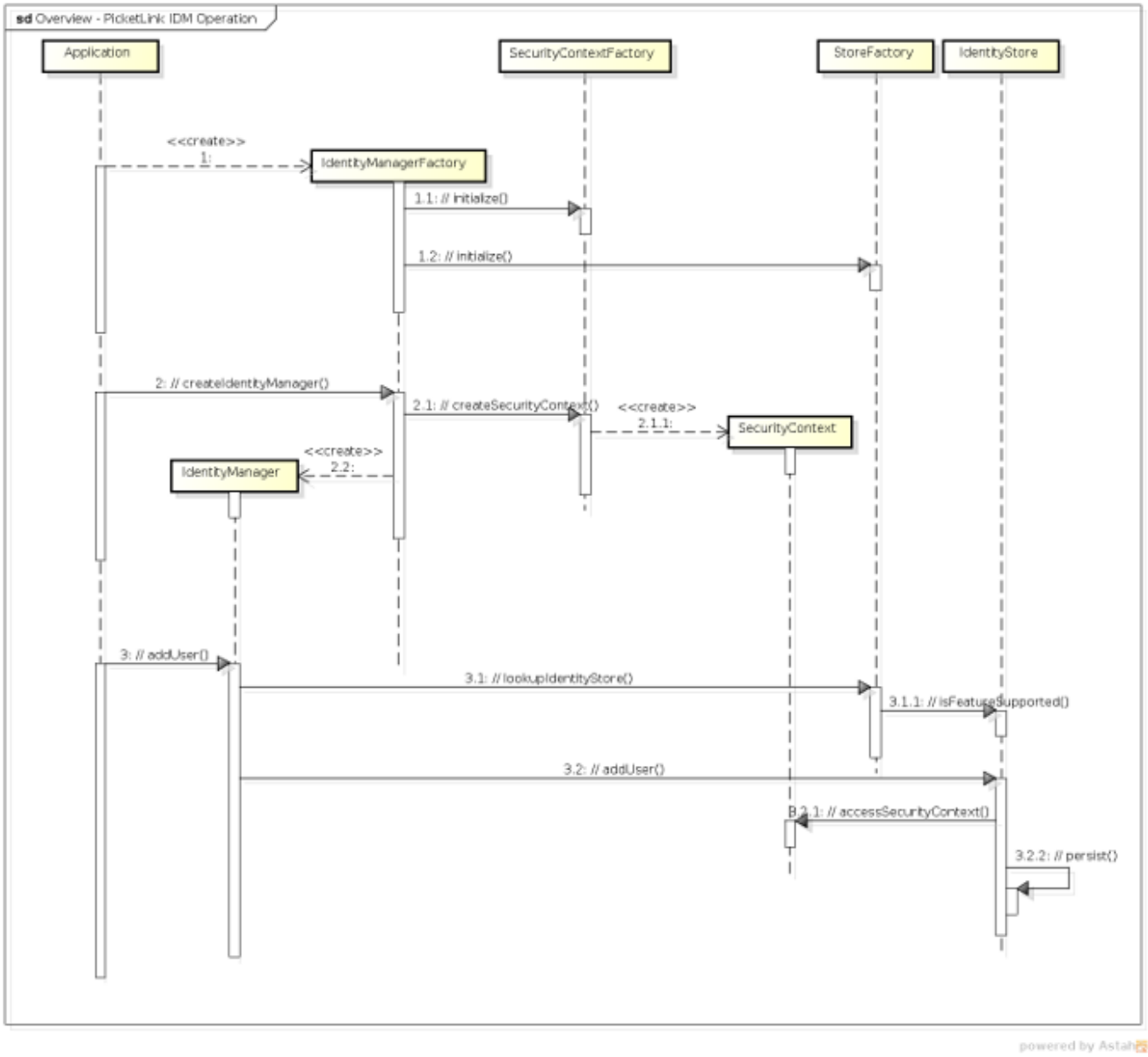
  It is a critical component as it provides all the necessary logic about how to store data.

- `IdentityManager` is an interface that provides a simple access for all identity management operations using one or more of the configured identity stores.

  All functionality provided by PicketLink is available from this interface, from where applications will interact most of the time.

For most use cases, users will only work with the `IdentityManagerFactory` and `IdentityManager` classes. Only advanced use cases may require a deep knowledge about other components in order to customize the default behaviour/implementation to suit a specific requirement.

The diagram below shows an overview about how a specific identity management operation is realized:



- *1* - The *Application* creates an `IdentityManagerFactory` instance from a previously created `IdentityConfiguration`. At this point, the factory reads the configuration and bootstraps the identity management ecosystem.
- *1.1* - The `IdentityManagerFactory` initializes the `SecurityContextFactory`.
- *1.2* - The `IdentityManagerFactory` initializes the `StoreFactory`.
- *2* - With a fully initialized `IdentityManagerFactory` instance, the *Application* is able to create `IdentityManager` instances and execute operations. `IdentityManager` instances are created for a specific *realm*, in this specific case we're creating an instance using the default realm.
- *2.1 and 2.1.1* - An `IdentityManager` instance is always associated with a `SecurityContext`. The `SecurityContext` is created and set into the `IdentityManager` instance. The same

security context is used during the entire lifecycle of the `IdentityManager`, it will be used to share state with the underlying identity stores and provide access to external resources (if necessary) in order to execute operations.

At this time, the `IdentityManager` is also configured to hold a reference to the `StoreFactory` in order to execute the operations against the underlying/configured `IdentityStore` instances.

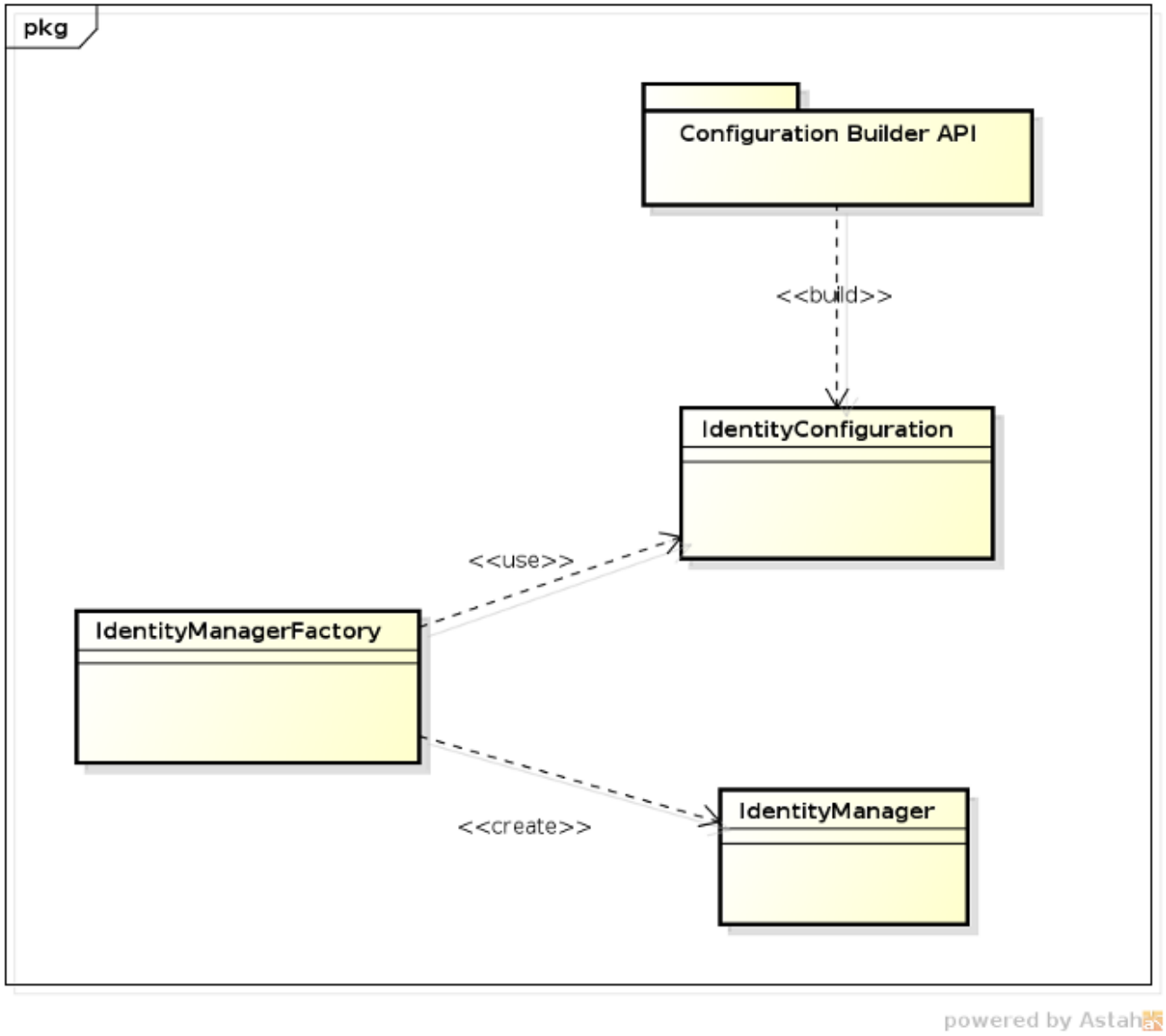
- *3* - Now the application holds a reference to the `IdentityManager` instance and it is ready to perform identity management operations (eg.: add an user, queries, validate credentials, etc).
- *3.1 and 3.1.1* - But before executing the operations, the `IdentityManager` needs to obtain from the `StoreFactory` the `IdentityStore` instance that should be used to execute a specific operation. Identity stores are selected by examining the configuration to see which store configuration supports a specific operation or feature.
- *3.2* - Now that the `IdentityManager` have selected which `IdentityStore` instance should be used, this last is invoked in order to process the operation.
- *3.2.1* - Usually, during the execution of an operation, the `IdentityStore` uses the current `SecurityContext`. The `SecurityContext` can hold some state that may be useful during the execution (eg.: the JPA store uses the security context to gain access to the current `EntityManager` instance) and also provide access for some IDM internal services like event handling, caching, etc.
- *3.2.2* - Finally, the `IdentityStore` executes the operation and persist or retrieve identity data from the underlying repository.

PicketLink IDM design is quite flexible and allows you to configure or even customize most of the behaviours described above. As stated earlier, most use cases require minimal knowledge about these details and the default implementation should be enough to satisfy the majority of requirements.

3.4. Configuration

3.4.1. 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 `IdentityManagerFactory` instance passing the previously created configuration. The `IdentityManagerFactory` can then be used to create `IdentityManager` instances via the `createIdentityManager()` method.



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

> **Note**
>
> For now, all configuration is set programmatically using the Configuration Builder API only. Later versions will also support a declarative configuration in a form of XML documents.

Each `IdentityManager` instance has its own *security context*, represented by the `SecurityContext` class. The security context contains temporary state which is maintained for one or more identity management operations within the scope of a single realm or tier. The `IdentityManager` (and its associated `SecurityContext`) is typically modelled as a request-

scoped object (for environments which support such a paradigm, such as a servlet container), or alternatively as an actor within the scope of a transaction. In the latter case, the underlying resources being utilised by the configured identity stores (such as a JPA `EntityManager`) would participate in the active transaction, and changes made as a result of any identity management operations would either be committed or rolled back as appropriate for the logic of the encapsulating business method.

The following sections describe various ways that configuration may be performed in different environments.

3.4.2. Programmatic Configuration

Configuration for Identity Management 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, 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 `IdentityManagerFactory` from where we can create `IdentityManager` instances and start to perform Identity Management operations:

```
IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

builder
  .stores()
  .file()
  .supportAllFeatures();

IdentityConfiguration configuration = builder.build();

IdentityManagerFactory identityManagerFactory = new IdentityManagerFactory(configuration);

IdentityManager identityManager = identityManagerFactory.createIdentityManager();

User user = new SimpleUser("john");

identityManager.add(user);
```

3.4.2.1. IdentityConfigurationBuilder for Programmatic Configuration

The `IdentityConfigurationBuilder` is the entry point for PicketLink configuration. It is a very simple class with some meaningful methods for all supported configuration options.

```
IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

builder
  .stores() // supported identity stores configuration
```

```
    .file()
      // file-based identity store configuration
    .jpa()
      // JPA-based identity store configuration
    .ldap()
      // LDAP-based identity store configuration
  .contextFactory(...); // for custom SecurityContextFactory implementations
```

In the next sections we'll cover each supported Identity Store and their specific configuration.

3.4.3. Security Context Configuration

The `SecurityContext` plays an important role in the PicketLink IDM achitecture. As discussed in the Architectural Overview, 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, UUID generator for `IdentityTpe` and `Relationship` 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`. All operations are executed for a specific `Realm` or `Tier`
- Access the current `IdentityManager` instance, from which the operation was executed.
- 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 the *Credential Handler API* in order to be able to update and validate credentials.
- Access to external resources, provided before the operation is executed and initialized by a `ContextInitializer`.

3.4.3.1. Initializing the SecurityContext

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 `SecurityContext` 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(SecurityContext 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 `SecurityContext` for a specific `IdentityStore`.

The configuration is also very simple, you just need to provide the following configuration:

```
IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

builder
  .stores()
    .file()
      .addContextInitializer(new MySecurityContextInitializer());
}
```

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.

3.4.3.2. Configuring how SecurityContext instances are created

`SecurityContext` instances are created by the `SecurityContextFactory`. If for some reason you need to change how `SecurityContext` instances are created, you can provide an implementation of this interface and configure it as follows:

```
IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

builder
  .stores()
    .contextFactory(mew MySecurityContextFactory());
}
```

3.4.4. Identity Store Feature Set

When configuring identity stores you must tell which features and operations should be executed by them. Features and operations are a key concept if you want to mix stores in order to execute operations against different repositories.

PicketLink provides a Java `enum`, called `FeatureGroup`, in which are defined all supported features. The table bellow summarize them:

Table 3.1. Identity class fields

Feature	
`FeatureGroup.agent`	
`FeatureGroup.user`	
`FeatureGroup.role`	
`FeatureGroup.group`	
`FeatureGroup.relationship`	
`FeatureGroup.credential`	
`FeatureGroup.realm`	
`FeatureGroup.tier`	

The features are a determinant factor when choosing an identity store to execute a specific operation. For example, if an identity store is configured with `FeatureGroup.user` we're saying that all `User` operations should be executed by this identity store. The same goes for `FeatureGroup.credential`, we're just saying that credentials can also be updated and validated using the identity store.

Beside that, provide only the feature is not enough. We must also tell the identity store which operations are supported by a feature. For example, we can configure a identity store to support only read operations for users, which is very common when using the LDAP identity store against a read-only tree. Operations are also defined by an `enum`, called `FeatureOperation`, as follows:

Table 3.2. Identity class fields

Operation	
`Featureoperation.create`	
`Featureoperation.read`	
`Featureoperation.update`	
`Featureoperation.delete`	
`Featureoperation.validate`	

During the configuration you can provide which features and operations should be supported using the *Configuration API*. You don't need to be forced to specify them individually, if you want to support all features and operations for a particular identity store you can use:

```
IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

builder
  .stores()
    .file()
```

```
      .supportAllFeatures();
}
```

For a more granular configuration you can also use:

```
IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

builder
  .stores()
    .file()
      .supportFeature(
        FeatureGroup.agent,
        FeatureGroup.user,
        FeatureGroup.role,
        FeatureGroup.group)
}
```

The configuration above defines the features individually. In this case the configured features are also supporting all operations. If you want to specify which operation should be supported by a feature you can use:

```
IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

builder
  .stores()
    .file()
      .supportFeature(FeatureGroup.agent, FeatureOperation.read)
      .supportFeature(FeatureGroup.user, FeatureOperation.read))
      .supportFeature(FeatureGroup.role, FeatureOperation.create))
      .supportFeature(FeatureGroup.role, FeatureOperation.read))
      .supportFeature(FeatureGroup.role, FeatureOperation.update))
      .supportFeature(FeatureGroup.role, FeatureOperation.delete))
      .supportFeature(FeatureGroup.group, FeatureOperation.create))
      .supportFeature(FeatureGroup.group, FeatureOperation.read))
      .supportFeature(FeatureGroup.group, FeatureOperation.update))
      .supportFeature(FeatureGroup.group, FeatureOperation.delete))
}
```

For a more complex configuration evolving multiple identity stores with a different feature set, look at the example bellow:

```
IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

builder
  .stores()
    .ldap()
      .supportFeature(FeatureGroup.agent)
      .supportFeature(FeatureGroup.user)
      .supportFeature(FeatureGroup.credential)
    .jpa()
```

```
        .supportFeature(FeatureGroup.role)
        .supportFeature(FeatureGroup.group)
        .supportFeature(FeatureGroup.relationship)
}
```

The configuration above shows how to use LDAP to store only agents, users and credentials and database for roles, groups and relationships.

> **Note**
>
> Remember that identity stores must have their features and operations configured. If you don't provide them you won't be able to build the configuration.

3.4.5. Identity Store Configurations

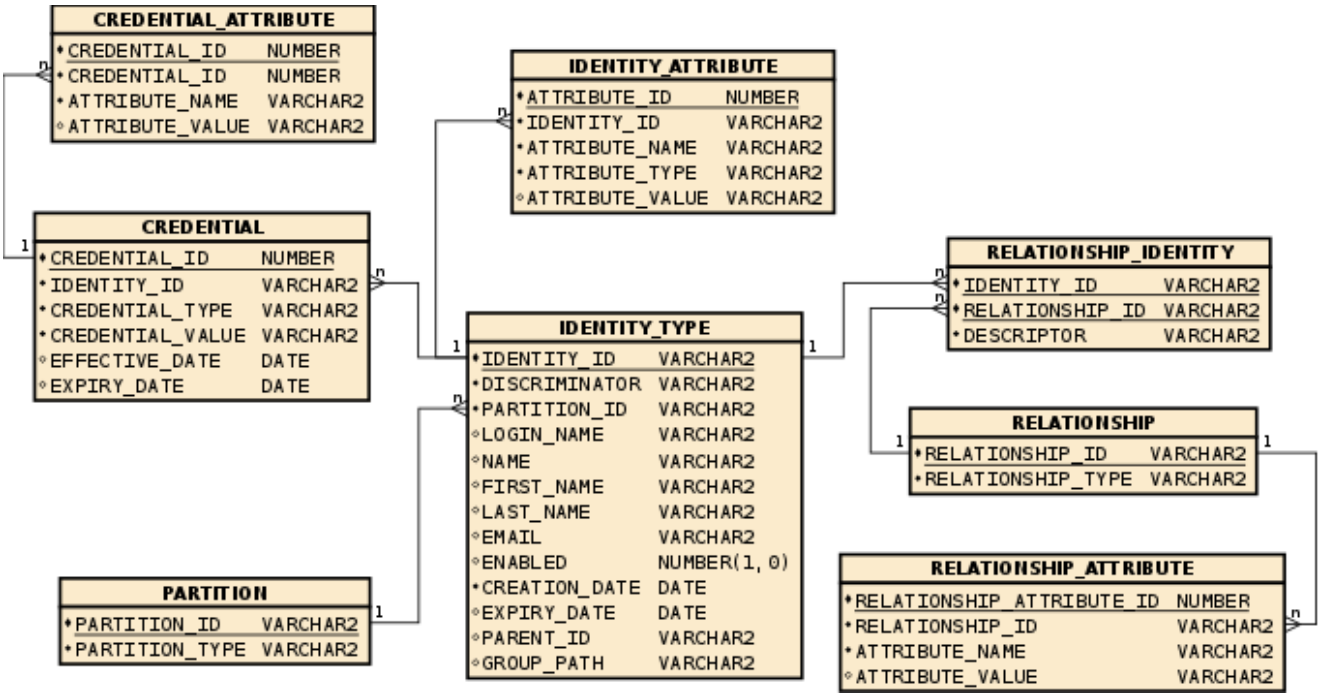
For each of the built-in `IdentityStore` implementations there is a corresponding `IdentityStoreConfiguration` implementation - the following sections describe each of these in more detail.

3.4.6. 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.

3.4.6.1. Recommended Database Schema

The following schema diagram is an example of a suitable database structure for storing IDM-related data:



Please note that the data types shown in the above diagram might not be available in your RDBMS; if that is the case please adjust the data types to suit.

3.4.6.2. 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-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-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.schema.IdentityObject</class>
<class>org.picketlink.idm.jpa.schema.PartitionObject</class>
<class>org.picketlink.idm.jpa.schema.RelationshipObject</class>
<class>org.picketlink.idm.jpa.schema.RelationshipIdentityObject</class>
<class>org.picketlink.idm.jpa.schema.RelationshipObjectAttribute</class>
<class>org.picketlink.idm.jpa.schema.IdentityObjectAttribute</class>
<class>org.picketlink.idm.jpa.schema.CredentialObject</class>
```

```
<class>org.picketlink.idm.jpa.schema.CredentialObjectAttribute</class>
```

3.4.6.3. 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;
```

3.4.6.4. Configuring the Identity class

The Identity class is the entity bean that is used to store the record for users, roles and groups. It should be annotated with `@IdentityType` and declare the following field values:

Table 3.3. Identity class fields

Property	Annotation	Description
ID	`@Identifier`	The unique identifier value for the identity (can also double as the primary key value)
Discriminator	`@Discriminator`	Indicates the identity type (i.e. user, agent, group or role) of the identity.
Partition	`@IdentityPartition`	The partition (realm or tier) that the identity belongs to
Login name	`@LoginName`	The login name for agent and user identities (for other identity types this will be null)
Name	`@IdentityName`	The name for group and role identities (for other identity types this will be null)
First Name	`@FirstName`	The first name of a user identity
Last Name	`@LastName`	The last name of a user identity
E-mail	`@Email`	The primary e-mail address of a user identity

Property	Annotation	Description
Enabled	`@Enabled`	Indicates whether the identity is enabled
Creation date	`@CreationDate`	The creation date of the identity
Expiry date	`@ExpiryDate`	The expiry date of the identity
Group parent	`@Parent`	The parent group (only used for Group identity types, for other types will be null)
Group path	`@GroupPath`	Represents the full group path (for Group identity types only)

The following code shows an example of an entity class configured to store Identity instances:

Example 3.1. Example Identity class

```
@IdentityType
@Entity
public class IdentityObject implements Serializable {

    @Discriminator
    private String discriminator;

    @ManyToOne
    @IdentityPartition
    private PartitionObject partition;

    @Identifier
    @Id
    private String id;

    @LoginName
    private String loginName;

    @IdentityName
    private String name;

    @FirstName
    private String firstName;

    @LastName
    private String lastName;

    @Email
    private String email;

    @Enabled
    private boolean enabled;

    @CreationDate
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;
```

```
    @ExpiryDate
    @Temporal(TemporalType.TIMESTAMP)
    private Date expiryDate;

    @ManyToOne
    @Parent
    private IdentityObject parent;

    @GroupPath
    private String groupPath;

    // getters and setters
}
```

3.4.6.5. Configuring the Attribute class

The Attribute class is used to store Identity attributes, and should be annotated with `@IdentityAttribute`

Table 3.4. Attribute class fields

Property	Annotation	Description
Identity	`@Parent`	The parent identity object to which the attribute value belongs
Name	`@AttributeName`	The name of the attribute
Value	`@AttributeValue`	The value of the attribute
Type	`@AttributeType`	The fully qualified classname of the attribute value class

Example 3.2. Example Attribute class

```
@Entity
@IdentityAttribute
public class IdentityAttribute implements Serializable {
    @Id @GeneratedValue private Long id;

    @ManyToOne @JoinColumn
    @Parent
    private IdentityObject identityObject;

    @AttributeName
    private String name;

    @AttributeValue
    private String value;

    @AttributeType
    private String type;
```

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

3.4.6.6. Configuring the Credential class

The credential entity is used to store user credentials such as passwords and certificates, and should be annotated with `@IdentityCredential`.

Table 3.5. Credential class fields

Property	Annotation	Description
Type	`@CredentialType`	The fully qualified classname of the credential type
Value	`@CredentialValue`	The value of the credential
Effective Date	`@EffectiveDate`	The effective date of the credential
Expiry Date	`@ExpiryDate`	The expiry date of the credential
Identity	`@Parent`	The parent identity to which the credential belongs

Example 3.3. Example Credential class

```
@Entity
@IdentityCredential
public class IdentityCredential implements Serializable {
    @Id @GeneratedValue private Long id;

    @CredentialType
    private String type;

    @CredentialValue
    private String credential;

    @EffectiveDate
    @Temporal (TemporalType.TIMESTAMP)
    private Date effectiveDate;

    @ExpiryDate
    @Temporal (TemporalType.TIMESTAMP)
    private Date expiryDate;

    @Parent
    @ManyToOne
    private IdentityObject identityType;

    // getters and setters
}
```

3.4.6.7. Configuring the Credential Attribute class

The Credential Attribute class is used to store arbitrary attribute values relating to the credential. It should be annotated with `@CredentialAttribute`.

Table 3.6. Credential Attribute class fields

Property	Annotation	Description
Credential Object	`@Parent`	The parent credential to which this attribute belongs
Attribute Name	`@AttributeName`	The name of the attribute
Attribute Value	`@AttributeValue`	The value of the attribute

Example 3.4. Example Credential Attribute class

```
@Entity
@CredentialAttribute
public class IdentityCredentialAttribute implements Serializable {
    @Id @GeneratedValue private Long id;

    @ManyToOne @JoinColumn
    @Parent
    private IdentityCredential credential;

    @AttributeName
    private String name;

    @AttributeValue
    private String value;

    // getters and setters
}
```

3.4.6.8. Configuring the Relationship class

Relationships are used to define typed associations between two or more identities. The Relationship class should be annotated with `@Relationship`.

Table 3.7. Relationship class fields

Property	Annotation	Description
Identifier	`@Identifier`	Unique identifier that represents the specific relationship (can also double as the primary key)
Relationship Class	`@RelationshipClass`	The fully qualified class name of the relationship type

Example 3.5. Example Relationship class

```
@Relationship
@Entity
public class Relationship implements Serializable {
    @Id
    @Identifier
    private String id;

    @RelationshipClass
    private String type;

    // getters and setters
}
```

3.4.6.9. Configuring the Relationship Identity class

The Relationship Identity class is used to store the specific identities that participate in a relationship. It should be annotated with `@RelationshipIdentity`.

Table 3.8. Relationship Identity class fields

Property	Annotation	Description
Relationship Descriptor	`@Discriminator`	Denotes the role of the identity in the relationship
Relationship Identity	`@Identity`	The identity that is participating in the relationship
Relationship	`@Parent`	The parent relationship object to which the relationship identity belongs

Example 3.6. Example Relationship Identity class

```
@RelationshipIdentity
@Entity
public class RelationshipIdentityObject implements Serializable {
    @Id @GeneratedValue private Long id;

    @Discriminator
    private String descriptor;

    @RelationshipIdentity
    @ManyToOne
    private IdentityObject identityObject;

    @Parent
    @ManyToOne
    private RelationshipObject relationshipObject;
```

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

3.4.6.10. Configuring the Relationship Attribute class

The Relationship Attribute class is used to store arbitrary attribute values that relate to a specific relationship. It should be annotated with `@RelationshipAttribute`.

Table 3.9. Relationship Attribute class fields

Property	Annotation	Description
Relationship	`@Parent`	The parent relationship object to which the attribute belongs
Attribute Name	`@AttributeName`	The name of the attribute
Attribute value	`@AttributeValue`	The value of the attribute

Example 3.7. Example Relationship Attribute class

```
@Entity
@RelationshipAttribute
public class RelationshipObjectAttribute implements Serializable {
    @Id @GeneratedValue private Long id;

    @ManyToOne @JoinColumn
    @Parent
    private Relationship relationship;

    @AttributeName
    private String name;

    @RelationshipValue
    private String value;

    // getters and setters
}
```

3.4.6.11. Configuring the Partition class

The Partition class is used to store information about partitions, i.e. Realms and Tiers. It should be annotated with `@Partition`.

Table 3.10. Partition class fields

Property	Annotation	Description
ID	`@Identifier`	The unique identifier value for the partition

Property	Annotation	Description
Type	`@Discriminator`	The type of partition, either Realm or Tier
Parent	`@Parent`	The parent partition (only used for Tiers)

Example 3.8. Example Partition class

```
@Entity
@Partition
public class PartitionObject implements Serializable {
    @Id @Identifier
    private String id;

    @Discriminator
    private String type;

    @ManyToOne
    @Parent
    private PartitionObject parent;

    // getters and setters
}
```

3.4.6.12. 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 `SecurityContext` by providing a `ContextInitializer` implementation, as discussed in Security Context Configuration. The `JPAContextInitializer` is provided by PicketLink and can be used to initialize the security context with a specific `EntityManager` instance. You can always extend this class and provide your own way to obtain the `EntityManager` from your application's environment.

```
IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

builder
  .stores()
    .file()
      .addContextInitializer(new JPAContextInitializer(emf) {
        @Override
        public EntityManager getEntityManager() {
          // logic goes here
        }
      });
}
```

By default, the `JPAContextInitializer` creates a `EntityManager` from the `EntityManagerFacatory` provided when creating a new instance.

3.4.7. LDAPIdentityStoreConfiguration

The LDAP identity store allows an LDAP directory server to be used to provide identity state. You can use this store in read-only or write-read mode, depending on your permissions on the server.

3.4.7.1. Configuration

The LDAP identity store can be configured by providing the following configuration:

```
IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

builder
  .stores()
    .ldap()
      .baseDN("dc=jboss,dc=org")
      .bindDN("uid=admin,ou=system")
      .bindCredential("secret")
      .url("ldap://localhost:10389")
      .userDNSuffix("ou=People,dc=jboss,dc=org")
      .roleDNSuffix("ou=Roles,dc=jboss,dc=org")
      .groupDNSuffix("ou=Groups,dc=jboss,dc=org")
      .supportAllFeatures();
```

The following table describes all configuration options:

Table 3.11. LDAP Configuration Options

Option	Description	Required
`baseDN`	Sets the fixed DN of the context from where identity types are stored.	Yes
`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.	Yes
`bindCredential`	Sets the password for the bindDN.	Yes
`url`	Sets the url that should be used to connect to the server. Eg.: ldap://<<server>>:389.	Yes

Option	Description	Required
`userDNSuffix`	Sets the fixed DN of the context where users should be read/stored from.	Yes
`agentDNSuffix`	Sets the fixed DN of the context where agents should be read/stored from. If not provided, will be used the context provided by the `setUserDNSuffix`	No
`roleDNSuffix`	Sets the fixed DN of the context where roles should be read/stored from.	Yes
`groupDNSuffix`	Sets the fixed DN of the context where groups should be read/stored from.	Yes

3.4.7.1.1. Mapping Groups to different contexts

Sometimes may be useful to map a specific group to a specific context or DN. By default, all groups are stored and read from the DN provided by the `setGroupDNSuffix` method, which means that you can not have groups with the same name.

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

```
LDAPIdentityStoreConfiguration ldapStoreConfig = new LDAPIdentityStoreConfiguration();

ldapStoreConfig
  .addGroupMapping("/QA Group", "ou=QA,dc=jboss,dc=org");
```

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

```
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
```

3.4.8. FileIdentityStoreConfiguration

This identity store uses the file system to persist identity state. The configuration for this identity store provides control over where to store identity data and if the state should be preserved between initializatons.

Identity data is stored using the *Java Serialization API*.

3.4.8.1. Filesystem Structure

Identity data is stored in the filesystem using the following structure:

```
${WORKING_DIR}/
pl-idm-partitions.db
pl-idm-relationships.db
  <<partition_name_directory>>
    pl-idm.agents.db
    pl-idm.roles.db
    pl-idm.groups.db
    pl-idm.credentials.db
  <<another_partition_directory>>
    ...
```

By default, files are stored in the *${java.io.tmpdir}/pl-idm* directory. For each partition there is a corresponding directory where agents, roles groups and credentials are stored in specific files.

3.4.8.2. Configuration

The file identity store can be easily configured by providing the following configuration:

```
IdentityConfigurationBuilder builder = new IdentityConfigurationBuilder();

builder
  .stores()
  .file()
  .preserveState(false)
  .addRealm(Realm.DEFAULT_REALM, "Testing")
  .addTier("Application")
  .supportAllFeatures()
  .supportRelationshipType(CustomRelationship.class, Authorization.class);
```

3.4.8.2.1. Preserving State Between Initializations

By default, during the initialization, the working directory is re-created. If you want to preserve state between initializations you should use the following configuration:

```
builder
.stores()
  .file()
```

```
  .preserveState(true) // preserve data
  .supportAllFeatures();
```

3.4.8.2.2. Changing the Working Directory

If you want to change the working directory, where files are stored, you can use the following configuration:

```
builder
  .stores()
    .file()
      .workingDir("/tmp/pl-idm")
      .supportAllFeatures();
```

3.4.9. Providing a Custom IdentityStore

TODO

3.5. Java EE Environments

In Java EE 6.0 and higher environments, basic configuration is performed automatically with a set of sensible defaults. During application deployment, PicketLink will scan all deployed entity beans for any beans annotated with `@IDMEntity`, and if found will use a configuration based on the `JPAIdentityStore`. If no entity beans have been configured for identity management and no other configuration is provided, a file-based identity store will be automatically configured to provide basic identity management features backed by the file system.

3.6. Using the IdentityManager

The `org.picketlink.idm.IdentityManager` interface provides access to the bulk of the IDM features supported by PicketLink. To get access to the `IdentityManager` depends on which environment you are using. The following two sections describe how to access the `IdentityManager` in both Java EE and Java SE environments.

3.6.1. Accessing the `IdentityManager` in Java EE

In a Java EE environment, PicketLink provides a producer method for `IdentityManager`, so getting a reference to it is as simply as injecting it into your beans:

```
@Inject IdentityManager identityManager;
```

3.6.1.1. Configuring the Application Realm

By default, an `IdentityManager` for the *default* realm will be injected. If the application should use a realm other than the default, then this must be configured via a producer method with the

`@PicketLink` qualifier. The following code shows an example of a configuration bean that sets the application realm to *acme*:

```
@ApplicationScoped
public class RealmConfiguration {
    private Realm applicationRealm;

    @Inject IdentityManagerFactory factory;

    @Init
    public void init() {
        applicationRealm = factory.getRealm("acme");
    }

    @Produces
    @PicketLink
    public Realm getApplicationRealm() {
      return applicationRealm;
    }
}
```

3.6.2. Accessing the `IdentityManager` in Java SE

3.7. 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` package. The following sections provide examples that show these implementations in action.

3.7.1. 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 SimpleUser("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 = identityManager.getUser("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 = identityManager.getUser("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 = identityManager.getUser("jsmith");
identityManager.remove("jsmith");
```

3.7.2. Managing Groups

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

```
Group employees = new SimpleGroup("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 SimpleGroup("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 = identityManager.getGroup("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 = identityManager.getGroup("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:

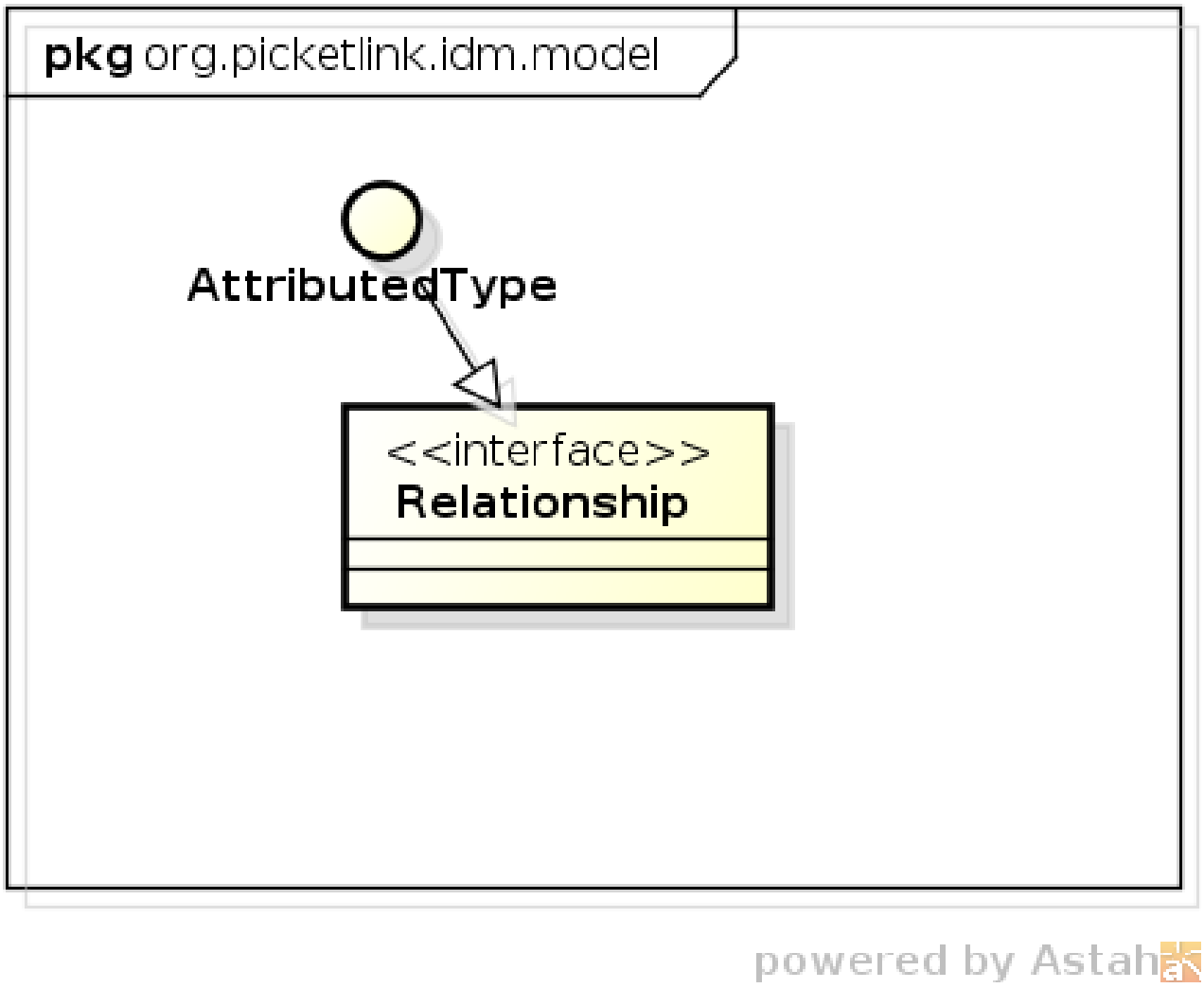
```
Group employees = identityManager.getGroup("employees");
employees.setEnabled(false);
identityManager.update(employees);
```

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

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

3.8. 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.

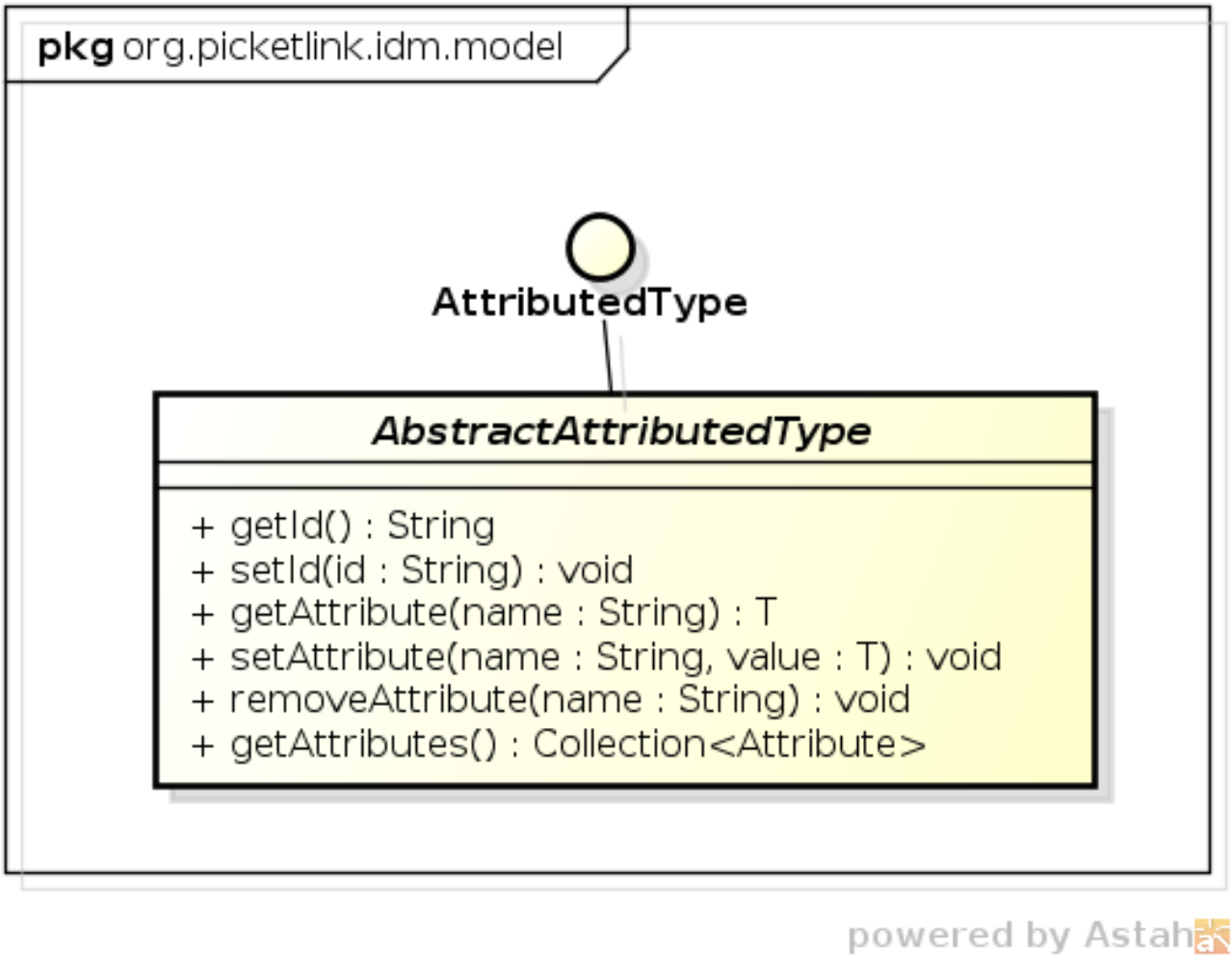
Note

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.

3.8.1. Built In Relationship Types

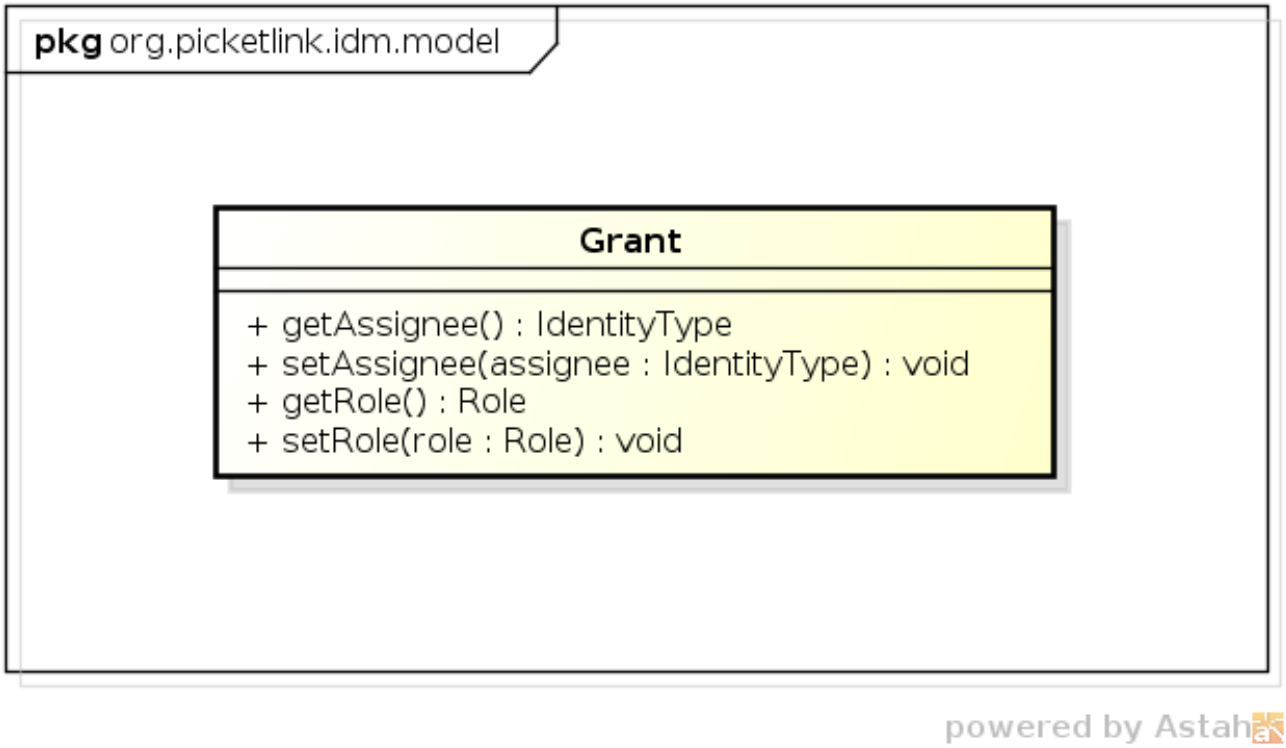
PicketLink provides a number of built-in relationship types, designed to address the most common requirements of a typical application. The following sections describe the built-in relationships and how they are intended to be used. Every built-in relationship type extends the `AbstractAttributedType` 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.

3.8.1.1. Application Roles

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



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

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

The above code is equivalent to the following:

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

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

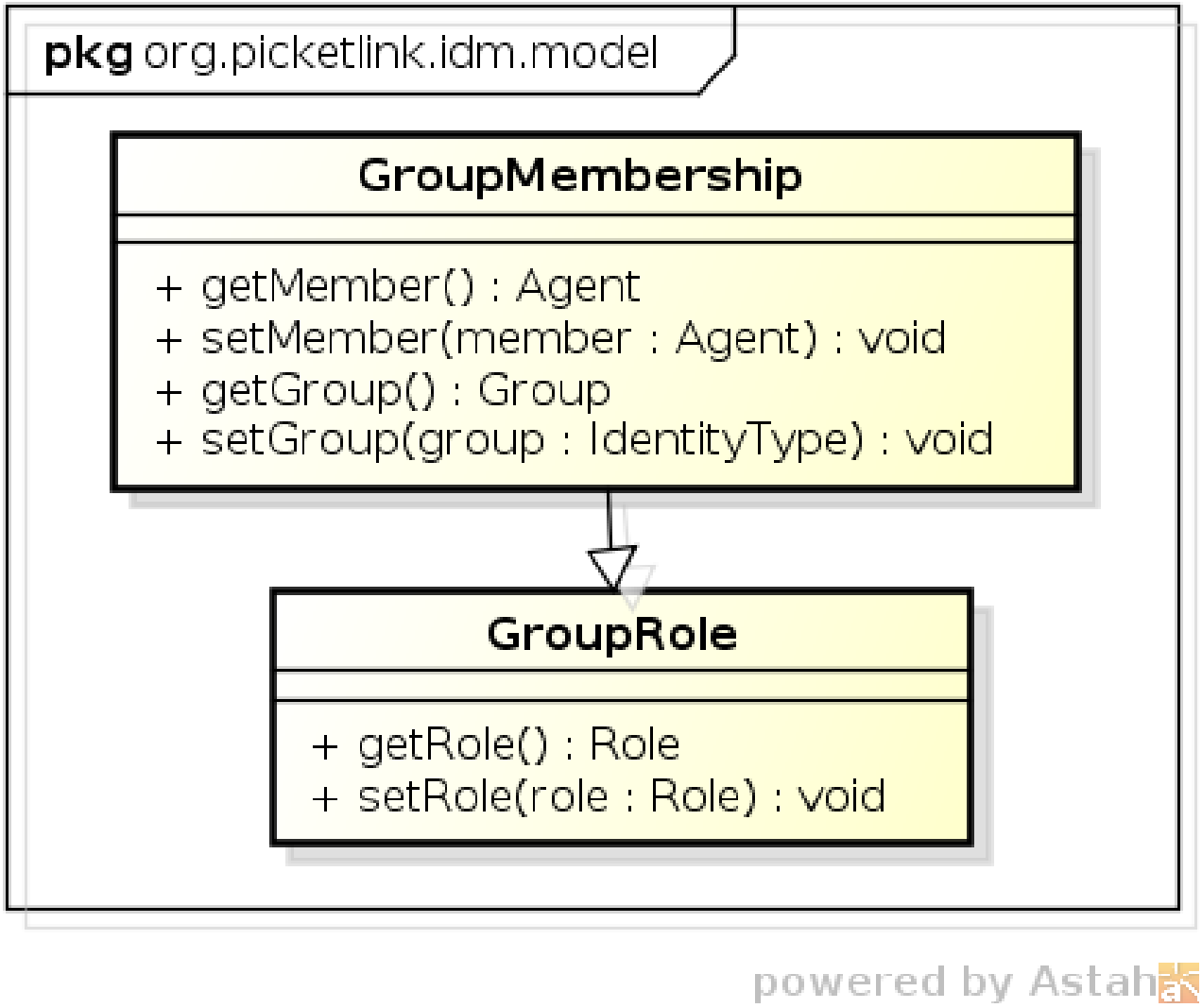
```
User bob = identityManager.getUser("bob");
Role superuser = identityManager.getRole("superuser");
identityManager.revokeRole(bob, superuser);
```

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

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

3.8.1.2. 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

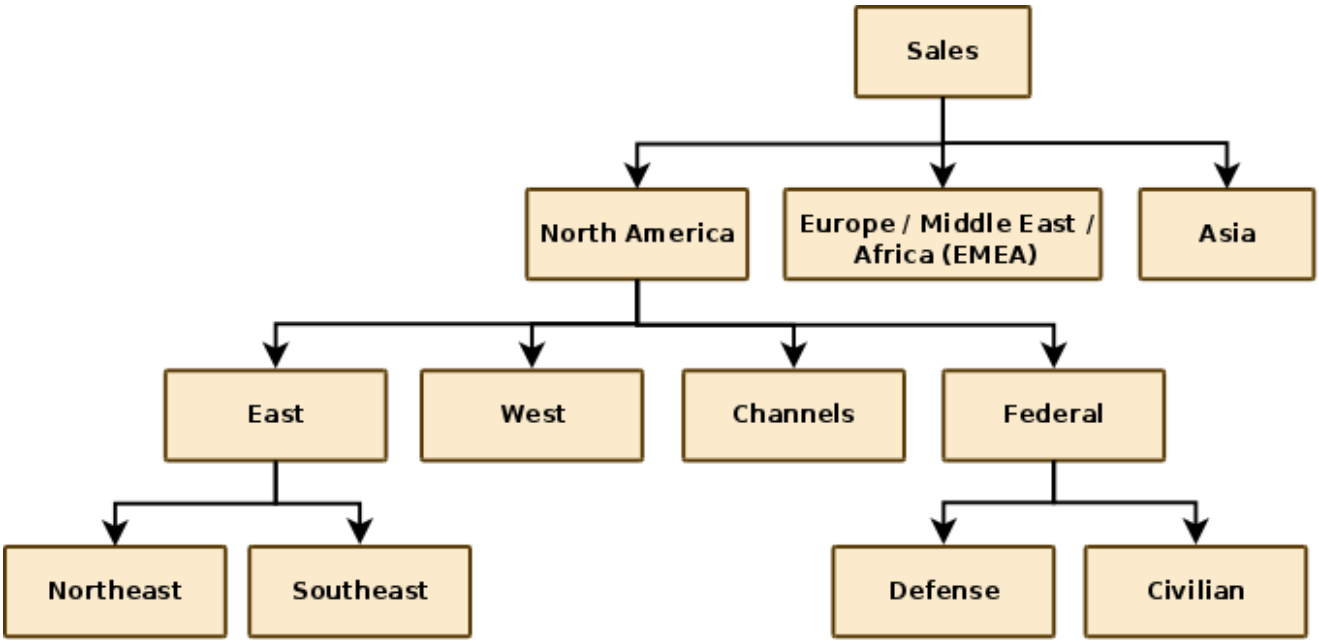
While the `GroupRole` relationship type extends `GroupMembership`, it does *not* mean that a member of a `GroupRole` automatically receives `GroupMembership`

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 SimpleGroup("Sales");
identityManager.add(sales);
```

We can then proceed to create its subgroups:

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

The second parameter of the `SimpleGroup()` 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 = identityManager.getRole("administrator");
User user = identityManager.getUser("jsmith");
Group group = identityManager.getGroup("Northeast");
identityManager.grantGroupRole(user, admin, group);
```

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

```
identityManager.revokeGroupRole(user, admin, group);
```

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

```
boolean isUserAGroupAdmin = identityManager.hasGroupRole(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 = identityManager.getUser("rbrown");
Group group = identityManager.getGroup("Northeast");
identityManager.addToGroup(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:

```
identityManager.removeFromGroup(user, group);
```

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

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

3.8.2. 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;
}
```

3.9. 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 the authentication chapter of this documentation 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
    };

   Agent getValidatedAgent();

   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
- `getValidatedAgent()` - If the credential was successfully validated, this method returns the `Agent` 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
}
```

3.10. Managing Credentials

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

```
void updateCredential(Agent agent, Object credential);
void updateCredential(Agent agent, Object credential, Date effectiveDate, Date expiryDate);
```

Both of these methods essentially do the same thing; they update a credential value for a specified `Agent` (or `User`). 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.

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 3.12. Built-in credential types

Credential type	Description	Supported by
`org.picketlink.idm.credential.Digest`	Used for digest-based authentication	`JPAIdentityStore` `FileBasedIdentityStore`

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

`java.security.cert.X509Certificate`	Used for X509 certificate based authentication	`JPAIdentityStore` `FileBasedIdentityStore`

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

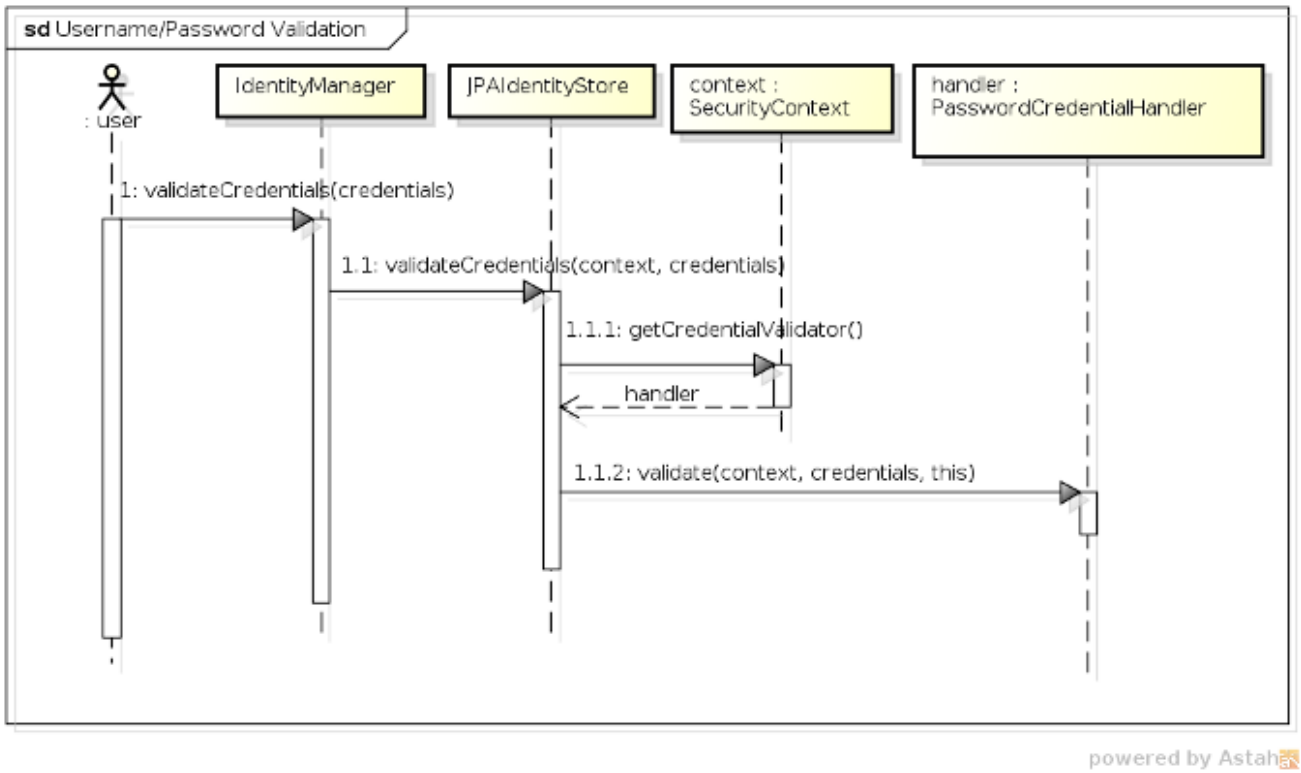
```
User user = identityManager.getUser("jsmith");
identityManager.updateCredential(user, new Password("abcd1234"));
```

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

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

3.11. 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 `SecurityContext` and the credentials to validate.
- *1.1.1* - In `JPAIdentityStore`'s implementation of the `validateCredentials()` method, the `SecurityContext` 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(SecurityContext context, Credentials credentials,
                  IdentityStore<?> identityStore);

    void update(SecurityContext context, Agent agent, 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({ UsernamePasswordCredentials.class, Password.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 {
```

3.11.1. 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(SecurityContext context, Agent agent,
                        CredentialStorage storage);
   <T extends CredentialStorage> T retrieveCurrentCredential(SecurityContext context,
                                                Agent agent, Class<T> storageClass);
   <T extends CredentialStorage> List<T> retrieveCredentials(SecurityContext context,
                                                Agent agent, Class<T> storageClass);
```

```
}
```

The `CredentialStorage` interface is quite simple and only declares two methods, `getEffectiveDate()` and `getExpiryDate()`:

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

The most important 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. 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;
    }

}
```

3.12. Built-in Credential Handlers

This section describes each of the built-in credential handlers, and any configuration parameters that may be set for them. Specific credential handler options can be set when creating a new `IdentityConfiguration`. Configured options are always specific to a particular identity store configuration, allowing different options to be specified between two or more identity stores. The `IdentityStoreConfiguration` interface provides a method called `getCredentialHandlersConfig()` that provides access to a `Map` which allows configuration options to be set for the identity store's credential handlers:

```
public interface IdentityStoreConfiguration {
    Map<String, Object> getCredentialHandlerProperties();
}
```

To gain access to the `IdentityStoreConfiguration` object before PicketLink is initialized, there are a couple of options. The first option is to provide an `IdentityConfiguration` object itself via a producer method.

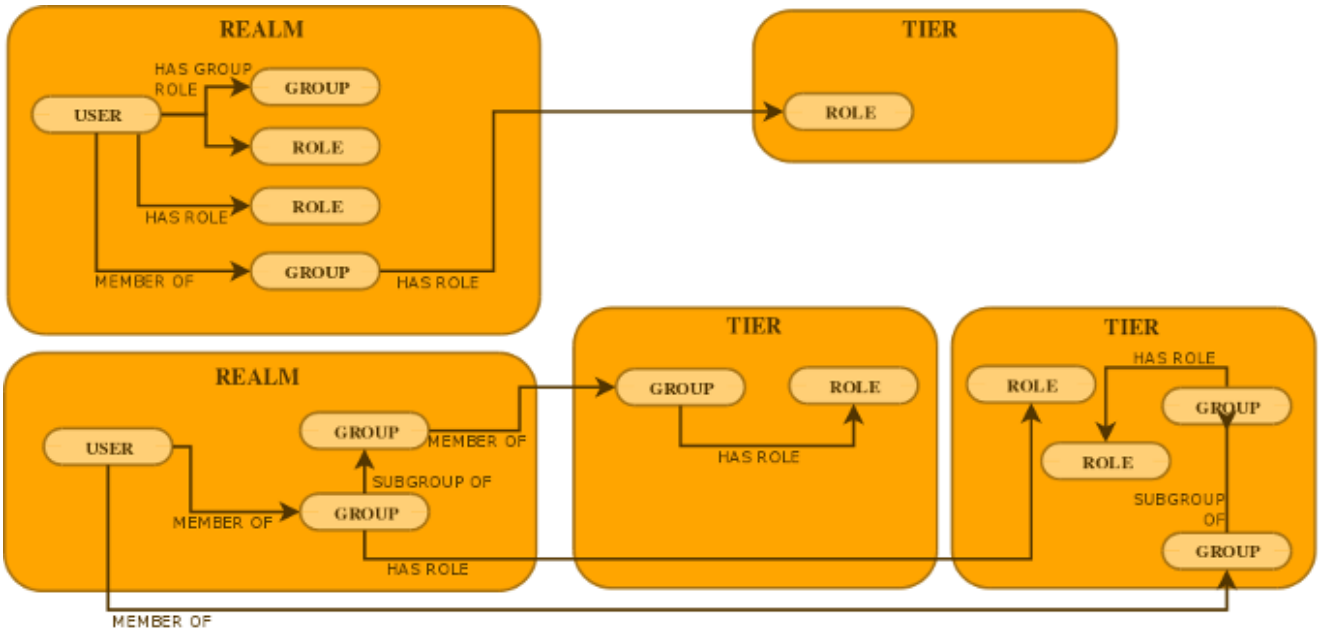
```

```

3.12.1.

3.13. Advanced Topics

3.13.1. Multi Realm Support

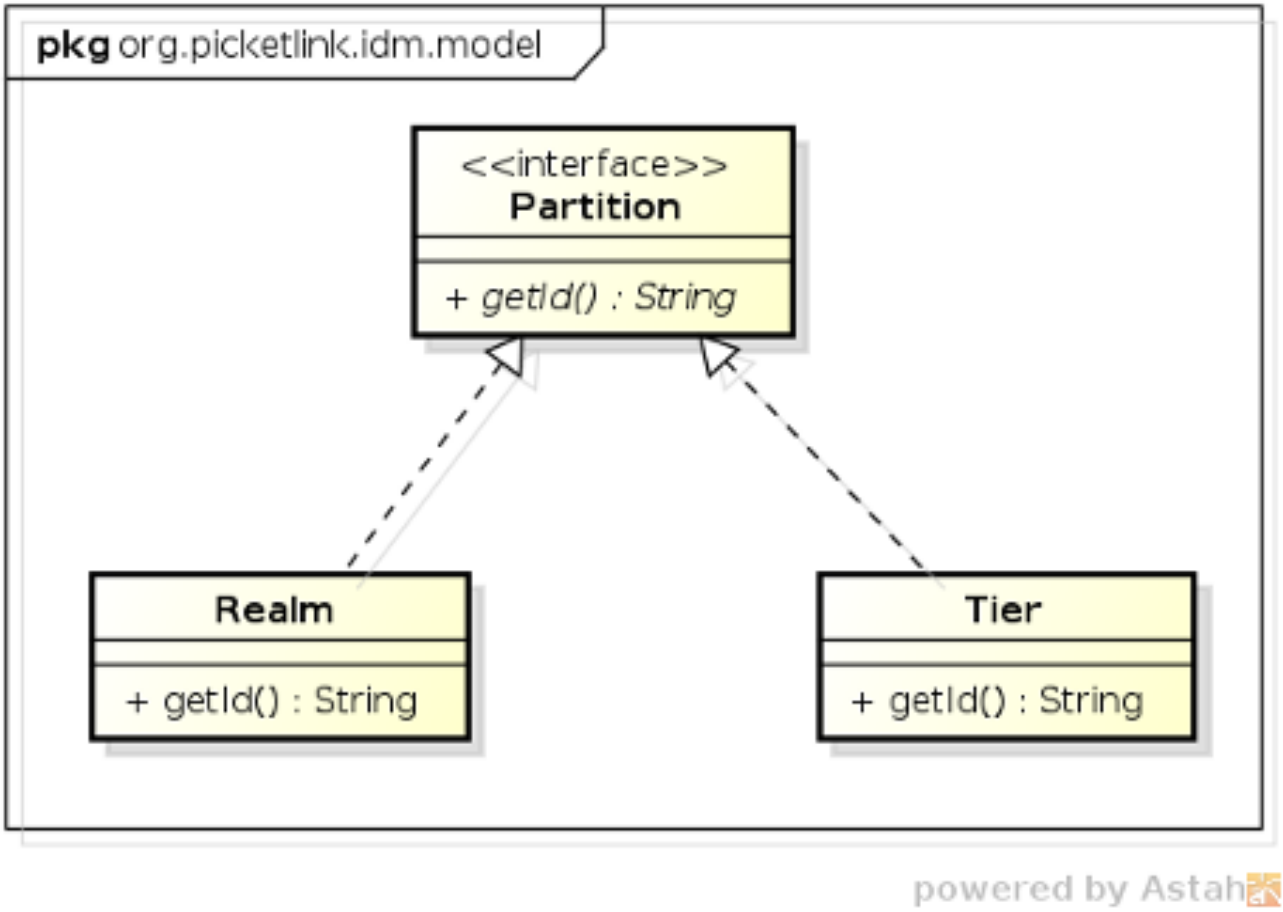
PicketLink has been designed from the ground up to support a system of *partitioning*, allowing the users, groups and roles of an application to be divided into *Realms* and *Tiers*.



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.

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



Selecting the specific partition that the identity management operations are performed in is controlled by specifying the partition when creating the `IdentityManager` via the `IdentityManagerFactory`'s overloaded `createIdentityManager()` methods:

```
IdentityManager createIdentityManager();
  IdentityManager createIdentityManager(Partition partition);
```

The first method (without parameters) will create an `IdentityManager` instance for the default realm. The second parameter allows a `Partition` object to be specified. Once the `IdentityManager` has been created, any identity management methods invoked on it will be performed within the selected partition. To look up the partition object, the `IdentityManagerFactory` provides two additional methods:

```
Realm getRealm(String id);
  Tier getTier(String id);
```

Here's an example demonstrating how a new user called "bob" is created in a realm called *acme*:

```
Realm acme = identityManagerFactory.getRealm("acme");
  IdentityManager im = identityManagerFactory.createIdentityManager(acme);
  im.add(new SimpleUser("bob"));
```

Chapter 4. Federation

4.1. Overview

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

4.2. 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.

4.3. SAML Web Browser Profile

PicketLink supports the following standard bindings:

- SAML HTTP Redirect Binding
- SAML HTTP POST Binding

4.4. 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.

4.5. SAML v2.0

4.5.1. Which Profiles are supported ?

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

4.5.2. 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

4.5.3. PicketLink Identity Provider (PIDP)

4.5.3.1. 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.

4.5.3.2. 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 4.5.3.4, “Identity Provider Configuration”
- Section 4.5.3.3, “Identity Provider Authenticators”
- Section 4.5.3.5, “Identity Stores”

4.5.3.3. Identity Provider Authenticators

4.5.3.3.1. Introduction

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

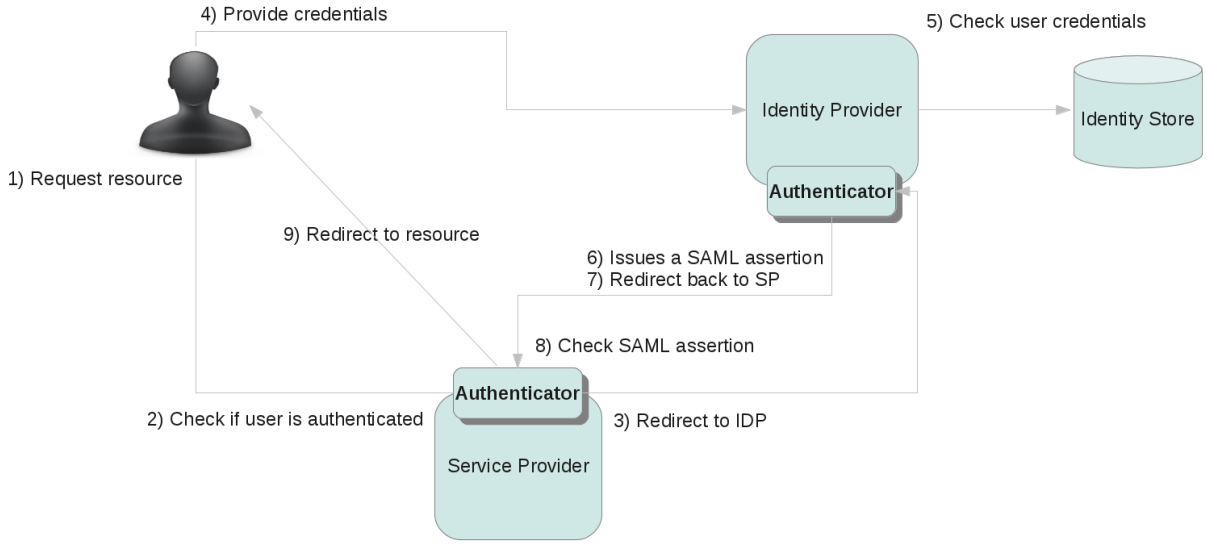


Figure 4.1. TODO InformalFigure image title empty

4.5.3.3.2. 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.

4.5.3.3.2.1. 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.

4.5.3.3.2.2. 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.

4.5.3.3.2.3. 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.

4.5.3.3.3. 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.

4.5.3.3.4. IDPWebBrowserSSOValve

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

4.5.3.3.4.1. Configuration

4.5.3.3.4.1.1. JBoss Application Server v6 and v5.x

Configure in WEB-INF/context.xml

4.5.3.3.4.1.2.

4.5.3.3.4.1.3. Apache Tomcat 5.5 and 6

Configure in META-INF/context.xml

4.5.3.3.4.1.4.

4.5.3.3.4.1.5. Example:

Example 4.1. context.xml

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

4.5.3.3.4.2.

4.5.3.3.4.3. 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.**

4.5.3.4. Identity Provider Configuration

4.5.3.4.1. 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** .

4.5.3.4.2. 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.

4.5.3.4.3. 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 .

4.5.3.4.3.1. 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.**

4.5.3.4.3.1.1. IdentityURL Element

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

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

4.5.3.4.3.1.2. 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.

4.5.3.4.3.1.3. 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 4.5.7.11, “SAML2SignatureGenerationHandler” and the Section 4.5.7.12, “SAML2SignatureValidationHandler” in the handlers chain (Handler Element).
3. Configure a **KeyProvider** * *element.

4.5.3.4.3.1.4. SAML Encryption Configuration

To enable encryption for SAML assertions you need to configure:

1. Set the **Encrypt** attribute to true;
2. Add the **Section 4.5.7.8, “SAML2EncryptionHandler”** and the Section 4.5.7.12, “SAML2SignatureValidationHandler” in the handlers chain (Handler Element).
3. Configure a **KeyProvider** * *element.

4.5.3.4.3.2. 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.

4.5.3.4.3.3. 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 4.5.3.6, “Security Token Service Configuration” .

4.5.3.5. Identity Stores

4.5.3.5.1. 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

4.5.3.5.2. 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.

4.5.3.6. Security Token Service Configuration

4.5.3.6.1. 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	Value
CanonicalizationMethod	Sets the canonicalization method.	Defaults to http://www.w3.org/2001/10/xml-exc-c14n#WithComments

4.5.3.6.1.1. 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.

4.5.4. PicketLink Service Provider (PSP)

4.5.4.1. 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.

4.5.4.2. 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 4.5.4.3, “Service Provider Configuration”
- Section 4.5.4.4, “Service Provider Authenticators”
- Configuring a SAML Security Domain

4.5.4.3. Service Provider Configuration

4.5.4.3.1. 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** .

4.5.4.3.2. 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.

4.5.4.3.3. 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 .

4.5.4.3.3.1. 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 REDIRECT 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	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.	

4.5.4.3.3.1.1. IdentityURL Element

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

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

4.5.4.3.3.1.2. ServiceURL Element

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

Eg.: http://localhost:8080/sales/

4.5.4.3.3.2. 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 4.5.7.11, “SAML2SignatureGenerationHandler” and the Section 4.5.7.12, “SAML2SignatureValidationHandler” in the handlers chain (Handler Element).
3. Configure a **KeyProvider** * *element.

4.5.4.3.3.3. 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.

4.5.4.4. Service Provider Authenticators

4.5.4.4.1. Introduction

PicketLink Service Providers Authenticators are important components responsible for the authentication of users using the SAML Assertion previously issued by an Identity Provider.

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.

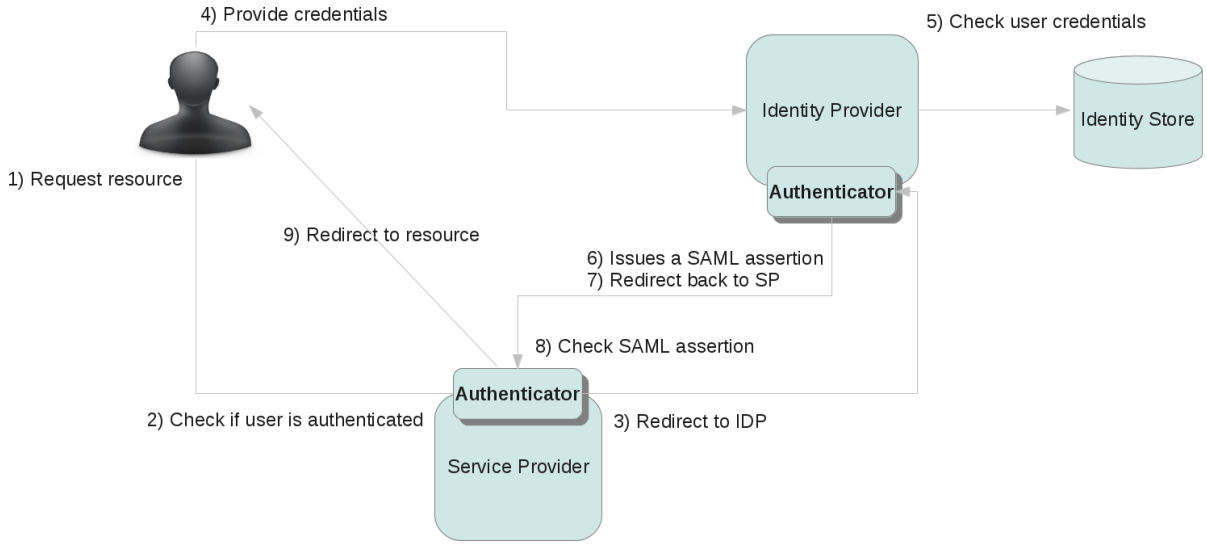


Figure 4.2. TODO InformalFigure image title empty

4.5.4.4.2. 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.

4.5.4.4.2.1. 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.

4.5.4.4.2.2. 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.

4.5.4.4.2.3. 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.

4.5.4.4.3. 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	*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** .

4.5.4.4.4. ServiceProviderAuthenticator

As of PicketLink v2.1, the ServiceProviderAuthenticator is the preferred Service Provider configuration to the deprecated Section 4.5.4.4.8, “SPPostFormAuthenticator” , Section 4.5.4.4.6, “SPRedirectFormAuthenticator” , Section 4.5.4.4.7, “SPPostSignatureFormAuthenticator” and Section 4.5.4.4.5, “SPRedirectSignatureFormAuthenticator” .

4.5.4.4.4.1. Configuration

https://docs.jboss.org/author/display/PLINK/Service+Provider+Configuration

4.5.4.4.4.2.

4.5.4.4.5. SPRedirectSignatureFormAuthenticator

> **Warning**
>
> As of PicketLink v2.1, the Section 4.5.4.4.4, “ServiceProviderAuthenticator” is the preferred Service Provider configuration to the **deprecated** Section 4.5.4.4.8, “SPPostFormAuthenticator” , Section 4.5.4.4.6, “SPRedirectFormAuthenticator” , Section 4.5.4.4.7, “SPPostSignatureFormAuthenticator” and Section 4.5.4.4.5, “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 4.5.4.4.6, “SPRedirectFormAuthenticator” .

4.5.4.4.5.1. Binding

HTTP/Redirect Binding (along with signature/encryption support)

4.5.4.4.5.2. Configuration

4.5.4.4.5.2.1. JBoss Application Server v5.x/6

Configure in WEB-INF/context.xml

4.5.4.4.5.2.2. Apache Tomcat v5.5/6.x

Configure in META-INF/context.xml

4.5.4.4.5.2.3.

4.5.4.4.5.2.4. Example:

Example 4.2. context.xml

```
<Context>
                                                                                <Valve
 className="org.picketlink.identity.federation.bindings.tomcat.sp.SPRedirectSignatureFormAuthenticator"
  />
</Context>
```

4.5.4.4.5.2.5. 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
7	idpAddress	String	optional - If the request.getRemoteAddr is not exactly	2.0

#	Name	Type	Objective	Since
			the IDP address that you have keyed in your deployment descriptor for keystore alias, you can configure it explicitly	

4.5.4.4.6. SPRedirectFormAuthenticator

> **Warning**
>
> As of PicketLink v2.1, the Section 4.5.4.4.4, “ServiceProviderAuthenticator” is the preferred Service Provider configuration to the **deprecated** Section 4.5.4.4.8, “SPPostFormAuthenticator” , Section 4.5.4.4.6, “SPRedirectFormAuthenticator” , Section 4.5.4.4.7, “SPPostSignatureFormAuthenticator” and Section 4.5.4.4.5, “SPRedirectSignatureFormAuthenticator” .

SPRedirectFormAuthenticator provides the SAMLv2 HTTP/Redirect binding support for service provider (SP) applications.

4.5.4.4.6.1. Binding

SAMLv2 HTTP/Redirect Binding

4.5.4.4.6.2. Configuration

4.5.4.4.6.2.1. JBoss Application Server v5.x/6

Configure in WEB-INF/context.xml

4.5.4.4.6.2.2. Apache Tomcat v5.5/6.x

Configure in META-INF/context.xml

4.5.4.4.6.2.3.

4.5.4.4.6.2.4. Example:

Example 4.3. context.xml

```
<Context>
                                                                                <Valve
 className="org.picketlink.identity.federation.bindings.tomcat.sp.SPRedirectFormAuthenticator"
  />
</Context>
```

4.5.4.4.6.2.5. 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

4.5.4.4.7. SPPostSignatureFormAuthenticator

> **Warning**
>
> As of PicketLink v2.1, the Section 4.5.4.4.4, “ServiceProviderAuthenticator” is the preferred Service Provider configuration to the **deprecated** Section 4.5.4.4.8, “SPPostFormAuthenticator” , Section 4.5.4.4.6, “SPRedirectFormAuthenticator” , Section 4.5.4.4.7, “SPPostSignatureFormAuthenticator” and Section 4.5.4.4.5, “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 4.5.4.4.8, “SPPostFormAuthenticator” .

4.5.4.4.7.1. Binding

HTTP/POST Binding (along with signature/encryption support)

4.5.4.4.7.2. Configuration

4.5.4.4.7.2.1. JBoss Application Server v5.x/6

Configure in WEB-INF/context.xml

4.5.4.4.7.2.2. Apache Tomcat v5.5/6.x

Configure in META-INF/context.xml

4.5.4.4.7.2.3.

4.5.4.4.7.2.4. Example:

Example 4.4. context.xml

```
<Context>
                                                                                <Valve
 className="org.picketlink.identity.federation.bindings.tomcat.sp.SPPostSignatureFormAuthenticator"
  />
</Context>
```

4.5.4.4.7.2.5. 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

4.5.4.4.8. SPPostFormAuthenticator

> **Warning**
>
> As of PicketLink v2.1, the Section 4.5.4.4.4, “ServiceProviderAuthenticator” is the preferred Service Provider configuration to the **deprecated** Section 4.5.4.4.8, “SPPostFormAuthenticator” , Section 4.5.4.4.6, “SPRedirectFormAuthenticator”

, Section 4.5.4.4.7, “SPPostSignatureFormAuthenticator” and Section 4.5.4.4.5, “SPRedirectSignatureFormAuthenticator” .

SPPostFormAuthenticator is the main authenticator used to configure a service provider (SP) application for SAMLv2.0

4.5.4.4.8.1. Binding

SAMLv2 HTTP/Post Binding

4.5.4.4.8.2. Configuration

4.5.4.4.8.2.1. JBoss Application Server v5.x/6

Configure in WEB-INF/context.xml

4.5.4.4.8.2.2. Apache Tomcat v5.5/6.x

Configure in META-INF/context.xml

4.5.4.4.8.2.3.

4.5.4.4.8.2.4. Example:

Example 4.5. context.xml

```
<Context>
                                                                                <Valve
 className="org.picketlink.identity.federation.bindings.tomcat.sp.SPPostFormAuthenticator"
  />
</Context>
```

4.5.4.4.8.2.5. 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

4.5.4.5. Service Provider Security Domain

4.5.4.5.1. 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>
```

4.5.5. SAML Authenticators (Tomcat,JBossAS)

4.5.5.1. 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

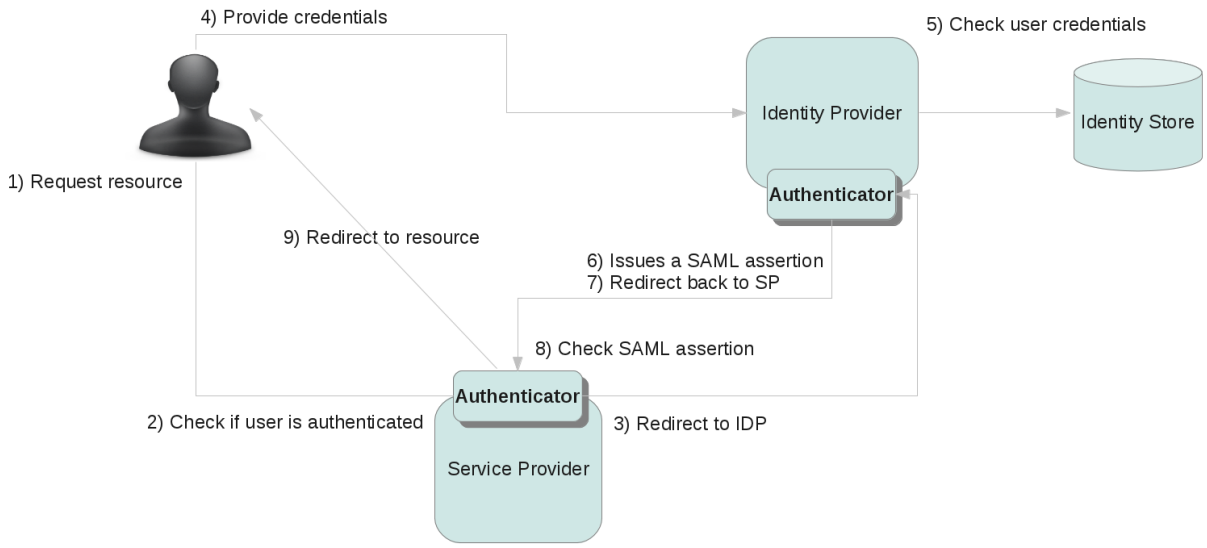


Figure 4.3. TODO InformalFigure image title empty

4.5.5.2. 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.

4.5.5.2.1. Authenticators/Valves for Identity Provider (IDP)

1. Section 4.5.3.3.4, “IDPWebBrowserSSOValve”

4.5.5.2.2. Authenticators/Valves for Service Provider (SP)

1. Section 4.5.4.4.4, “ServiceProviderAuthenticator”

4.5.5.2.2.1. Deprecated (as of PicketLink v2.1)

1. Section 4.5.4.4.8, “SPPostFormAuthenticator”

2. Section 4.5.4.4.6, “SPRedirectFormAuthenticator”
3. Section 4.5.4.4.7, “SPPostSignatureFormAuthenticator”
4. Section 4.5.4.4.5, “SPRedirectSignatureFormAuthenticator”

4.5.5.3.

4.5.5.4. Useful Information

- Tomcat Character Encoding (UTF-8 etc) [http://wiki.apache.org/tomcat/FAQ/CharacterEncoding]

4.5.6. Digital Signatures in SAML Assertions

4.5.6.1. 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.

4.5.6.2. SImple Example Scenario

4.5.6.2.1. 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.

4.5.6.2.2. 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.

4.5.7. SAML2 Handlers

4.5.7.1. 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.

4.5.7.2. 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>
```

4.5.7.2.1. 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.**

4.5.7.2.2. 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.

4.5.7.3. 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.

4.5.7.4. 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 4.5.7.7, “SAML2AuthenticationHandler”
2. Section 4.5.7.6, “SAML2AttributeHandler”
3. Section 4.5.7.5, “RolesGenerationHandler”
4. Section 4.5.7.9, “SAML2IssuerTrustHandler”
5. SAML2LogOutHandler

4.5.7.5. RolesGenerationHandler

4.5.7.5.1. Objective

Handler dealing with attributes for SAML2

4.5.7.5.2. Fully Qualified Name

org.picketlink.identity.federation.web.handlers.saml2.RolesGenerationHandler

4.5.7.5.3. Configuration

Should be configured in WEB-INF/picketlink-handlers.xml

4.5.7.5.3.1. Example:

Example 4.6. 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>
```

4.5.7.5.4. 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		

4.5.7.5.4.1. Example:

Example 4.7. 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>
```

4.5.7.6. SAML2AttributeHandler

4.5.7.6.1. 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.

4.5.7.6.2. Fully Qualified Name

org.picketlink.identity.federation.web.handlers.saml2.SAML2AttributeHandler

4.5.7.6.3. Configuration

Should be configured in WEB-INF/picketlink-handlers.xml

4.5.7.6.3.1. Example:

Example 4.8. 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>
```

4.5.7.6.4. 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

4.5.7.6.4.1. Example:

Example 4.9. WEB-INF/picketlink-handlers.xml

```
<Handler class="org.picketlink.
                    identity.federation.
                    web.handlers.
                    saml2.SAML2AttributeHandler">
<Option Key="ATTRIBUTE_CHOOSE_FRIENDLY_NAME" Value="true"/>
</Handler>
```

4.5.7.6.4.2.

4.5.7.6.4.3. Example:

4.5.7.6.4.4.

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));

4.5.7.6.4.5.

4.5.7.6.4.6. Additional References

- PicketLink IDP using LDAP Attributes [https://community.jboss.org/wiki/PicketLinkIDPUsingLDAPAttributes]

4.5.7.7. SAML2AuthenticationHandler

4.5.7.7.1. Objective

Handler handles the SAML request at the IDP and the SAML response at the SP.

4.5.7.7.2. Fully Qualified Name

org.picketlink.identity.federation.web.handlers.saml2.SAML2AuthenticationHandler

4.5.7.7.3. Configuration

Should be configured in WEB-INF/picketlink-handlers.xml

4.5.7.7.3.1. Example:

Example 4.10. 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>
```

4.5.7.7.4. 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.		

4.5.7.7.4.1. Example:

Example 4.11. WEB-INF/picketlink-handlers.xml

```
<Handler class="org.picketlink.identity.
                    federation.web.
                    handlers.saml2.SAML2AuthenticationHandler">
<Option Key="DISABLE_ROLE_PICKING" Value="true"/>
</Handler>
```

4.5.7.7.4.2. 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**

4.5.7.8. SAML2EncryptionHandler

4.5.7.8.1. Objective

Handles SAML Assertions Encryption and Signature Generation. This handler uses the configuration provided in the KeyProvider to encrypt and sign SAML Assertions.

4.5.7.8.2. Fully Qualified Name

org.picketlink.identity.federation.web.handlers.saml2.SAML2EncryptionHandler

4.5.7.8.3. 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 4.5.7.11, “SAML2SignatureGenerationHandler” and the Section 4.5.7.12, “SAML2SignatureValidationHandler” handlers.
- *Do not use this handler with the __ Section 4.5.7.11, “SAML2SignatureGenerationHandler” _* configured in the same chain. Otherwise SAML messages will be signed several times._

4.5.7.8.4. Configuration

Should be configured in WEB-INF/picketlink.xml:

4.5.7.8.4.1. Example:

4.5.7.8.4.2.

```
<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>
```

4.5.7.8.5. Configuration Parameters

#	Name	Type	Objective	Default Value	SP/IDP	Since Version

4.5.7.8.5.1.

4.5.7.9. SAML2IssuerTrustHandler

4.5.7.9.1. Objective

Handles Issuer trust.Trust decisions are based on the url of the issuer of the saml request/ response sent to the handler chain.

4.5.7.9.2. Fully Qualified Name

org.picketlink.identity.federation.web.handlers.saml2.SAML2IssuerTrustHandler

4.5.7.9.3. Configuration

Should be configured in WEB-INF/picketlink-handlers.xml

4.5.7.9.3.1. Example:

Example 4.12. 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>
```

4.5.7.9.4. Configuration Parameters

#	Name	Type	Objective	Default Value	SP/IDP	Since Version

4.5.7.9.4.1.

4.5.7.10. SAML2LogOutHandler.java

4.5.7.10.1. Objective

Handler for SAML2 Logout Profile.

4.5.7.10.2. Fully Qualified Name

org.picketlink.identity.federation.web.handlers.saml2.SAML2LogOutHandler.java

4.5.7.10.3. Configuration

Should be configured in WEB-INF/picketlink-handlers.xml

4.5.7.10.3.1. Example:

Example 4.13. 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>
```

4.5.7.10.4. Configuration Parameters

#	Name	Type	Objective	Default Value	SP/IDP	Since Version

4.5.7.10.4.1.

4.5.7.11. SAML2SignatureGenerationHandler

4.5.7.11.1. Objective

Handles SAML Signature Generation. This handler uses the configuration provided in the KeyProvider to sign SAML messages.

4.5.7.11.2. Fully Qualified Name

org.picketlink.identity.federation.web.handlers.saml2.SAML2SignatureGenerationHandler

4.5.7.11.3. Configuration

Should be configured in WEB-INF/picketlink.xml.

4.5.7.11.3.1. Example:

4.5.7.11.3.2.

```
<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>
```

4.5.7.11.4. Configuration Parameters

#	Name	Type	Objective	Default Value	SP/IDP	Since Version

4.5.7.11.4.1.

4.5.7.12. SAML2SignatureValidationHandler

4.5.7.12.1. Objective

Handles SAML Signature Validation. This handler uses the configuration provided in the KeyProvider to process signature validation.

4.5.7.12.2. Fully Qualified Name

org.picketlink.identity.federation.web.handlers.saml2.SAML2SignatureValidationHandler

4.5.7.12.3. Configuration

Should be configured in WEB-INF/picketlink.xml.

4.5.7.12.3.1. Example:

4.5.7.12.3.2.

```
<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>
```

4.5.7.12.4. Configuration Parameters

#	Name	Type	Objective	Default Value	SP/IDP	Since Version

4.5.7.12.4.1.

4.5.8. 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.

4.5.8.1. 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

4.5.8.2. 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.

4.5.9. 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)

4.5.9.1. Configuration providers at the IDP

4.5.9.1.1. 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.

4.5.9.2. Configuration Providers at the SP

4.5.9.2.1. SPPostMetadataConfigurationProvider

Fully Qualified Name: org.picketlink.identity.federation.web.config.SPPostMetadataConfigurationProvider

Binding Supported: SAML2/HTTP-POST

4.5.9.2.1.1. 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.

4.5.9.2.2. SPRedirectMetadataConfigurationProvider

Fully Qualified Name: org.picketlink.identity.federation.web.config.SPRedirectMetadataConfigurationProvider

Binding Supported: SAML2/HTTP-Redirect

4.5.9.2.2.1. 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.

4.5.9.2.3. 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

4.5.10. Metadata Support

Table of Contents

4.5.10.1. 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* .

4.5.10.2. 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

4.5.10.2.1. 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>
```

4.5.10.2.2. 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

4.5.10.3. 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

4.5.11. Token Registry

4.5.11.1. 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.

4.5.11.2. 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

4.5.11.3. 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 4.5.3.6, "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.

4.5.11.3.1. 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.

4.5.11.3.2. 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.**

4.5.11.4. 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;
    }

}
```

4.5.12. 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 .

4.5.13. 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.

4.5.13.1. Service Provider Configuration

In your web.xml, configure a Section 4.5.13.6, “SPFilter” as shown below as an example:

Example 4.14. 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>
     <dispatcher>REQUEST</dispatcher>
   </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);

4.5.13.2. 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 4.15. 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 4.16. 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 4.17. 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>
```

4.5.13.3. Other References

1. http://community.jboss.org/wiki/PicketLinkSAMLSSOForWebContainers

4.5.13.4. IDPLoginServlet

IDPLoginServlet provides the login capabilities for IDP applications running on any servlet container.

4.5.13.4.1. 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

4.5.13.4.2. 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.

4.5.13.5. IDPServlet

IDPServlet supports the SAMLv2 HTTP/POST binding for an IDP running on any servlet container.

4.5.13.5.1. 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
			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

4.5.13.5.2. Configuration

The Section 4.5.13.4, “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).

4.5.13.6. 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.

4.5.13.6.1. 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

4.6. SAML v1.1

4.6.1. SAML v1.1

Please refer to the wikipedia page [http://en.wikipedia.org/wiki/SAML_1.1] for more information.

4.6.2. PicketLink SAML v1.1 Support

Please read it at http://community.jboss.org/wiki/PicketLinkSAMLV11Support

4.7. Trust

4.7.1. Security Token Server (STS)

4.7.1.1. 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.

4.7.1.2. References

PicketLink STS Dashboard [http://community.jboss.org/wiki/PicketLinkSTSDashboard]

4.7.1.3. 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

4.7.1.3.1. BinaryTokenHandler

4.7.1.3.1.1. Fully Qualified Name

org.picketlink.trust.jbossws.handler.BinaryTokenHandler

4.7.1.3.1.2. 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.

4.7.1.3.1.3. Author

Anil Saldhana

4.7.1.3.1.4. 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 see also section.See Also:setHttpHeaderName(String)setHttpCookieName(String)setEncodingType(String)setValueType(String)setValue

4.7.1.3.1.5. Test Case

http://anonsvn.jboss.org/repos/picketlink/integration-tests/trunk/picketlink-trust-tests/src/test/java/org/picketlink/test/trust/tests/STSWSBinaryTokenTestCase.java

4.7.1.3.2. SAML2Handler

4.7.1.3.2.1. Full Name:

org.picketlink.trust.jbossws.handler.SAML2Handler

4.7.1.3.2.2. Authors:

- Marcus Moyses
- Anil Saldhana

4.7.1.3.2.3. 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).

4.7.1.3.2.3.1. 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 4.18. 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.

4.7.1.3.2.3.2. Inbound:

This is the behavior when the handler is configured on the server side.

The server side setting is as follows:

Example 4.19. 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.

4.7.1.3.2.4. 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]

4.7.1.3.3. WSAuthenticationHandler

4.7.1.3.3.1. FQN:

org.picketlink.trust.jbossws.handler.WSAuthenticationHandler

4.7.1.3.3.2. Objective:

Perform authentication for POJO based webservices.

4.7.1.3.3.3. 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");
   }
}
```

4.7.1.3.4. WSAuthorizationHandler

4.7.1.3.4.1. FQN:

org.picketlink.trust.jbossws.handler.WSAuthorizationHandler

4.7.1.3.4.2. Objective:

Provide authorization capabilities to POJO based web services.

4.7.1.3.4.3. Example Usage:

Please refer to the documentation on WSAuthenticationHandler.

> **Important**
>
> The example is in WSAuthenticationHandler [https://docs.jboss.org/author/display/PLINK/WSAuthenticationHandler] section.

4.7.1.4. Protecting EJB Endpoints

4.7.1.4.1. 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

4.7.1.4.1.1. 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.

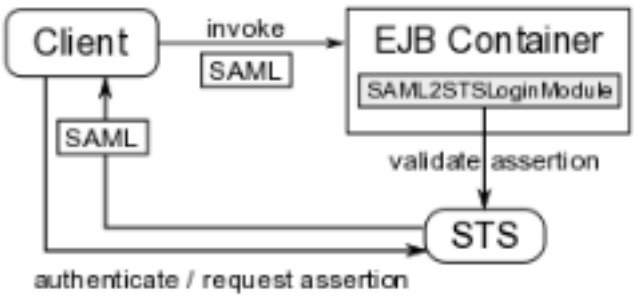


Figure 4.4. TODO Gliffy image title empty

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 4.7.1.5.3, “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.

4.7.1.4.2. 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

4.7.1.4.2.1. 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 4.7.1.5.3, “SAML2STSLoginModule”

3. Change the Remoting Connector to use the new Security Realm

4.7.1.4.2.1.1. 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.

4.7.1.4.2.1.2. 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.

4.7.1.4.2.1.3. 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.

4.7.1.4.2.1.4. 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 "));
```

4.7.1.4.3. References

- JBoss AS 5 : https://community.jboss.org/wiki/SAMLEJBIntegrationWithPicketLinkSTS

4.7.1.5. STS Login Modules

This page references the PicketLink Login Modules for the Security Token Server.

4.7.1.5.1. References

> **Tip**
>
> PicketLink STS Login Modules [http://community.jboss.org/wiki/PicketLinkSTSLoginModules] has the required details.

4.7.1.5.2. JBWSTokenIssuingLoginModule

4.7.1.5.2.1. Fully Qualified Name

org.picketlink.trust.jbossws.jaas. **JBWSTokenIssuingLoginModule**

4.7.1.5.2.2. 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.

4.7.1.5.2.3. 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.

4.7.1.5.2.3.1. 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).

4.7.1.5.2.4. Example configurations

Either you specify the module options directly or you can use a properties file for the STS related properties.

4.7.1.5.2.4.1. 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>
```

4.7.1.5.2.4.2. 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>
```

4.7.1.5.2.4.3. 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>
```

4.7.1.5.3. SAML2STSLoginModule

4.7.1.5.3.1. FQN

org.picketlink.identity.federation.bindings.jboss.auth. **SAML2STSLoginModule**

4.7.1.5.3.2. Author:

Stefan Guilhen

4.7.1.5.3.3. 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.

4.7.1.5.3.4. 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

4.7.1.5.3.5. 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>.

4.7.1.5.4. SAMLTokenCertValidatingLoginModule

org.picketlink.identity.federation.bindings.jboss.auth. **SAMLTokenCertValidatingLoginModule**

4.7.1.5.4.1. Author:

Peter Skopek

4.7.1.5.4.2. 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

4.7.1.5.4.3. 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)
```

4.7.1.5.4.4. 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>
```

4.7.1.5.5. STSValidatingLoginModule

4.7.1.5.5.1. FQN:

org.picketlink.identity.federation.core.wstrust.auth.STSValidatingLoginModule

4.7.1.5.5.2. Author:

Daniel Bevenius

4.7.1.5.5.3. 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

4.8. Extensions

4.8.1. Extensions

This page shows all the extensions and customizations available in the PicketLink project.

4.8.2. PicketLinkAuthenticator

4.8.2.1. PicketLinkAuthenticator

4.8.2.1.1. FQN

org.picketlink.identity.federation.bindings.tomcat.PicketLinkAuthenticator

4.8.2.1.2. 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.

4.8.2.1.3. 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.

4.8.2.1.4. 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.

4.8.2.1.4.1. 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>
```

4.8.2.1.4.2. 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.

4.9. PicketLink API

4.9.1. Working with SAML Assertions

4.9.1.1. 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.

4.9.1.2. 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()));
    }
```

4.9.1.3. 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;
    }
```

4.9.1.4. 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);
    }
```

4.10. 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 SP, Salesforce as IDP [https://docs.jboss.org/author/display/PLINK/Picketlink+as+SP%2C+Salesforce+as+IDP]

4.10.1. 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

4.10.1.1. 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

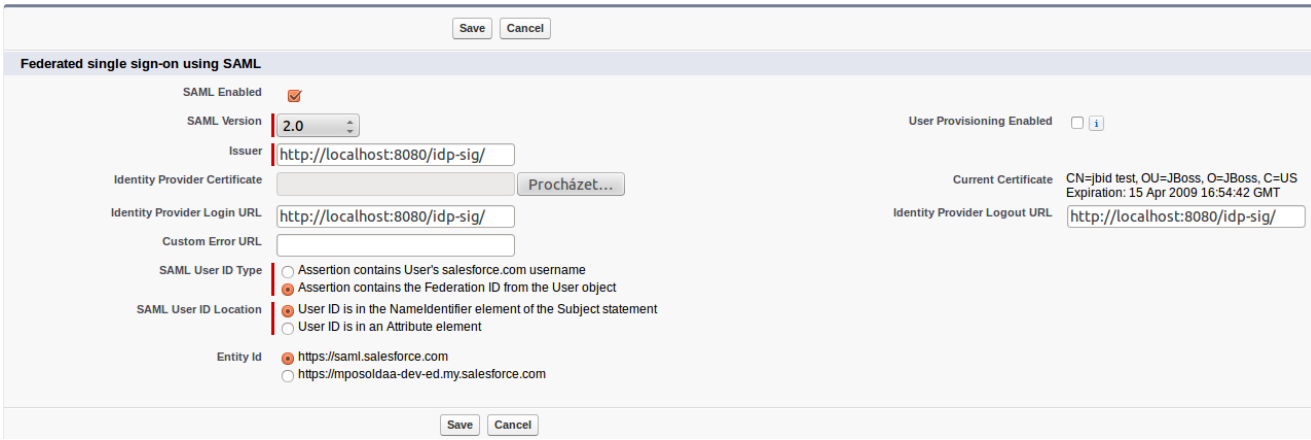


Figure 4.5. TODO InformalFigure image title empty

- **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.

- **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.

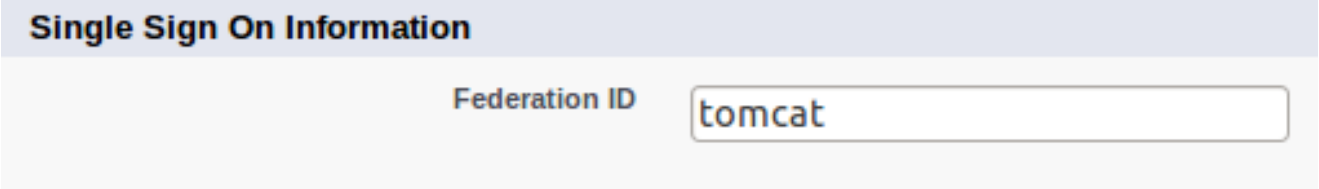


Figure 4.6. TODO InformalFigure image title empty

4.10.1.2. 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>
```

4.10.1.2.1.

4.10.1.2.2. 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* :

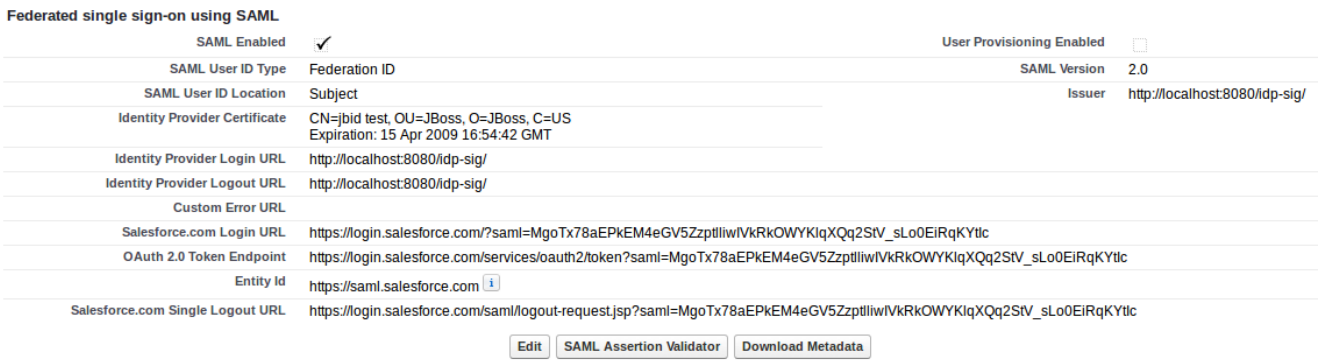


Figure 4.7. TODO InformalFigure image title empty

- **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):

```
```

```
<?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>
.....
```

4.10.1.3. 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.

4.10.1.4. 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/]

4.10.2. 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

4.10.2.1. 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.

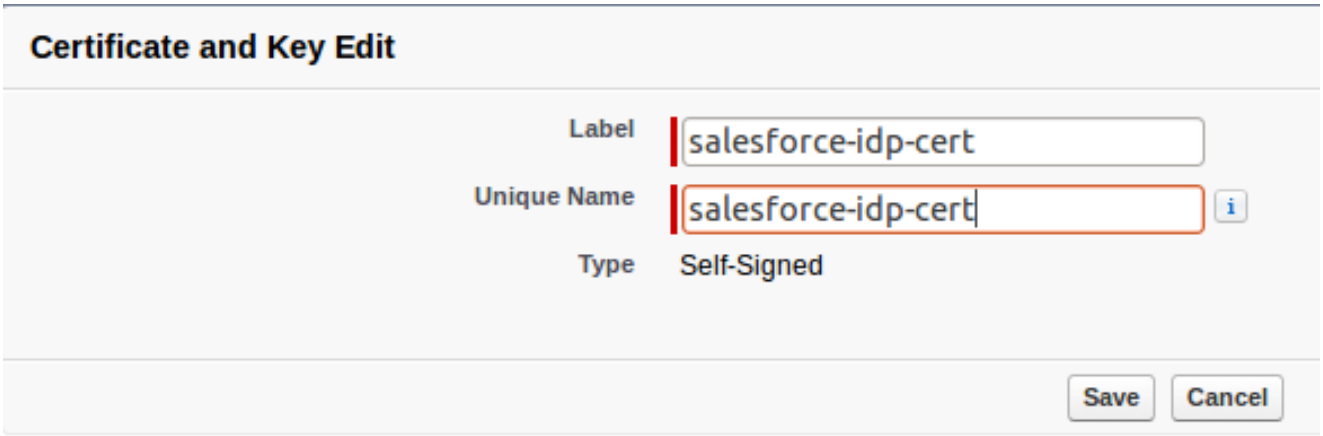


Figure 4.8. TODO InformalFigure image title empty

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:

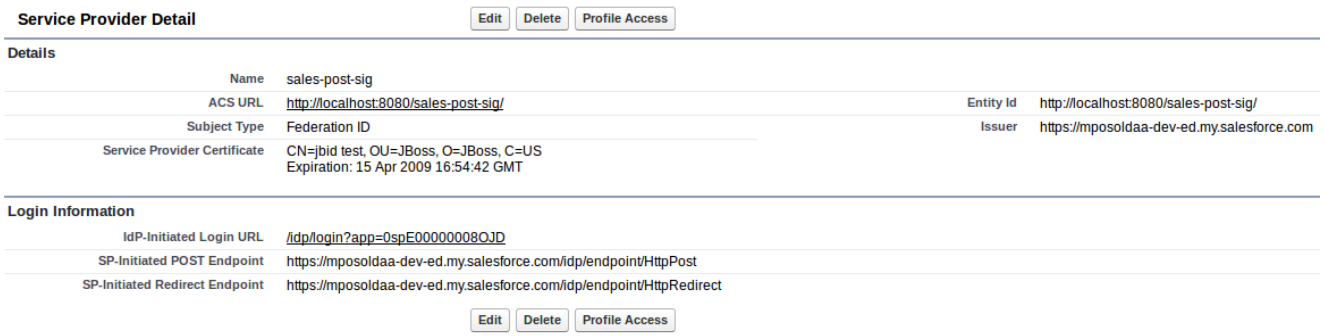


Figure 4.9. TODO InformalFigure image title empty

> **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.

4.10.2.2. 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:

```
<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.

4.10.2.3. 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.

4.10.3. 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

4.10.3.1. 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:

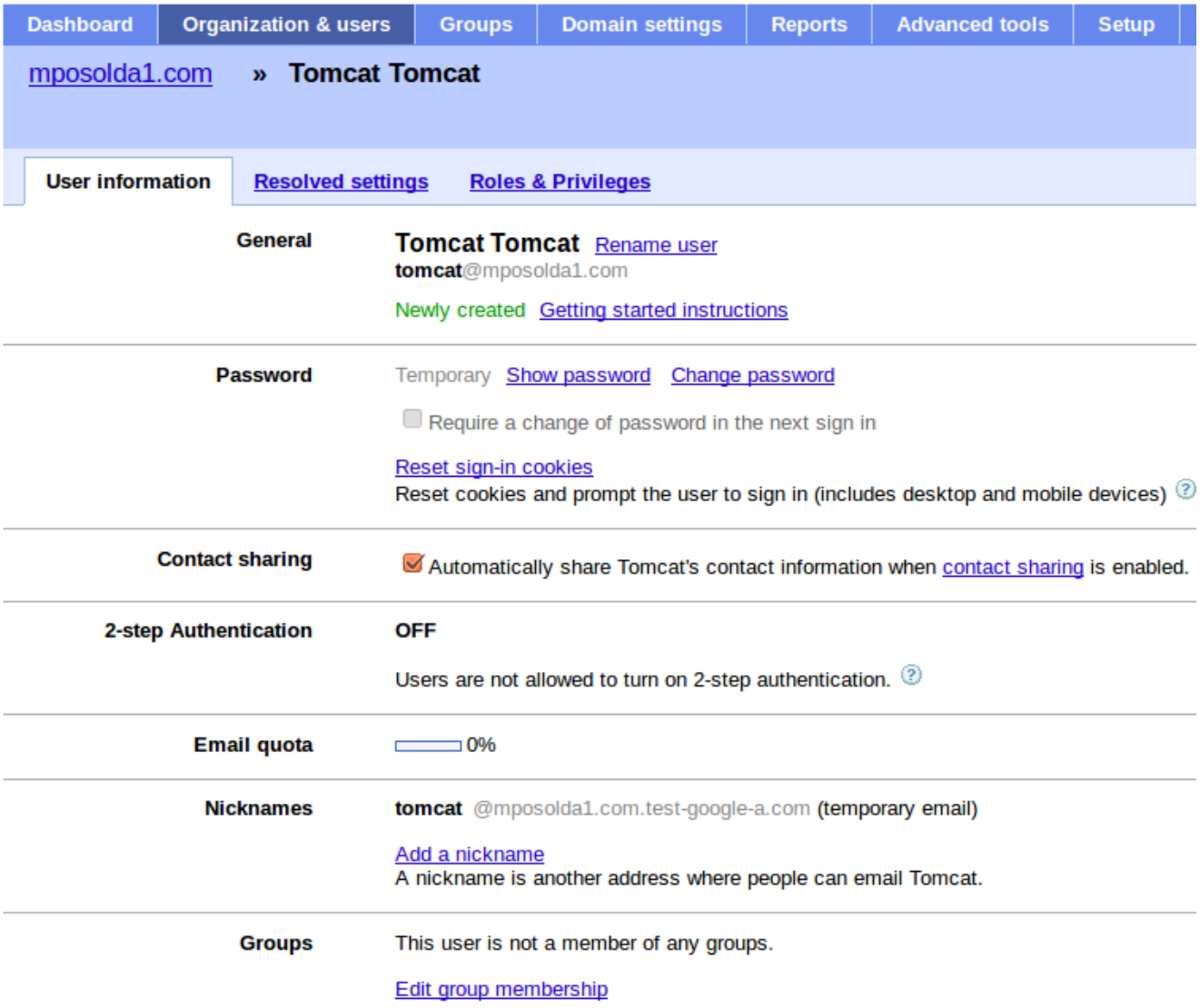


Figure 4.10. TODO InformalFigure image title empty

- **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.

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.

Figure 4.11. TODO InformalFigure image title empty

4.10.3.2. 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>
```

4.10.3.3. 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.