Seam Security

Security - Introduction

1.1. Overview

The Seam Security module provides a number of useful features for securing your Java EE application, which are briefly summarised in the following sections. The rest of the chapters contained in this documentation each focus on one major aspect of each of the following features.

1.1.1. Authentication

Authentication is the act of establishing, or confirming, the identity of a user. In many applications a user confirms their identity by providing a username and password (also known as their *credentials*). Seam Security allows the developer to control how users are authenticated, by providing a flexible *Authentication API* that can be easily configured to allow authentication against any number of sources, including but not limited to databases, LDAP directory servers or some other external authentication service.

If none of the built-in authentication providers are suitable for your application, then it is also possible to write your own custom Authenticator implementation.

1.1.2. Identity Management

Identity Management is a set of useful APIs for managing the users, groups and roles within your application. The identity management features in Seam are provided by PicketLink IDM, and allow you to manage users stored in a variety of backend security stores, such as in a database or LDAP directory.

1.1.3. External Authentication

Seam Security contains an external authentication sub-module that provides a number of features for authenticating your application users against external authentication services, such as OpenID and SAML.

1.1.4. Authorization

While *authentication* is used to confirm the identity of the user, *authorization* is used to control which actions a user may perform within your application. Authorization can be roughly divided into two categories; coarse-grained and fine-grained. An example of a coarse-grained restriction is allowing only members of a certain group or role to perform a privileged operation. A fine-grained restriction on the other hand may allow only a certain individual user to perform a specific action on a specific object within your application.

There are also rule-based permissions, which bridge the gap between fine-grained and coarse-grained restrictions. These permissions may be used to determine a user's privileges based on certain business logic.

1.2. Configuration

1.2.1. Maven Dependencies

The Maven artifacts for all Seam modules are hosted within the JBoss Maven repository. Please refer to the *Maven Getting Started Guide* [http://community.jboss.org/wiki/MavenGettingStarted-Users] for information about configuring your Maven installation to use the JBoss repository.

To use Seam Security within your Maven-based project, it is advised that you import the Seam BOM (Bill of Materials) which declares the versions for all Seam modules. First declare a property value for `${seam.version}` as follows:

```
<properties>
  <seam.version>3.1.0.Final</seam.version>
</properties>
```

You can check the *JBoss Maven Repository* [https://repository.jboss.org/nexus/content/groups/public/org/jboss/seam/seam-bom/] directly to determine the latest version of the Seam BOM to use.

Now add the following lines to the list of dependencies within the `dependencyManagement` section of your project's `pom.xml` file:

```
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>seam-bom</artifactId>
  <version>${seam.version}</version>
  <type>pom</type>
  <scope>import</scope>
</dependency>
```

Once that is done, add the following dependency (no version is required as it comes from `seam-bom`):

```
<dependency>
  <groupId>org.jboss.seam.security</groupId>
  <artifactId>seam-security</artifactId>
</dependency>
```

If you wish to use the external authentication module in your application to allow authentication using OpenID or SAML, then add the following dependency also:

```
<dependency>
  <groupId>org.jboss.seam.security</groupId>
  <artifactId>seam-security-external</artifactId>
</dependency>
```

1.2.2. Enabling the Security Interceptor

To enable many of the features of Seam Security, the Security interceptor must be configured in your application's `beans.xml` file. Add the following configuration to your `beans.xml` to enable the Security Interceptor:

```
    <interceptors>
        <class>org.jboss.seam.security.SecurityInterceptor</class>
    </interceptors>
```

Security - Authentication

2.1. Basic Concepts

The majority of the Security API is centered around the `Identity` bean. This bean represents the identity of the current user, the default implementation of which is a session-scoped, named bean. This means that once logged in, a user's identity is scoped to the lifecycle of their current session. The two most important methods that you need to know about at this stage in regard to authentication are `login()` and `logout()`, which as the names suggest are used to log the user in and out, respectively.

As the default implementation of the `Identity` bean is named, it may be referenced via an EL expression, or be used as the target of an EL action. Take the following JSF code snippet for example:

```
<h:commandButton action="#{identity.login}" value="Log in"/>
```

This JSF command button would typically be used in a login form (which would also contain inputs for the user's username and password) that allows the user to log into the application.

Note

The bean type of the `Identity` bean is `org.jboss.seam.security.Identity`. This interface is what you should inject if you need to access the `Identity` bean from your own beans. The default implementation is `org.jboss.seam.security.IdentityImpl`.

The other important bean to know about right now is the `Credentials` bean. Its' purpose is to hold the user's credentials (such as their username and password) before the user logs in. The default implementation of the `Credentials` bean is also a session-scoped, named bean (just like the `Identity` bean).

The `Credentials` bean has two properties, `username` and `credential` that are used to hold the current user's username and credential (e.g. a password) values. The default implementation of the `Credentials` bean provides an additional convenience property called `password`, which may be used in lieu of the `credential` property when a simple password is required.

Note

The bean type of the `Credential` bean is `org.jboss.seam.security.Credentials`. The default implementation for this bean type is `org.jboss.seam.security.CredentialsImpl`. Also, as credentials may come in many forms (such as passwords, biometric data such as that from a fingerprint reader, etc) the `credential` property of the `Credentials` bean must be able to support each variation, not just passwords. To allow for this, any credential that implements the `org.picketlink.idm.api.Credential` interface is a valid value for the `credential` property.

2.2. Built-in Authenticators

The Seam Security module provides the following built-in `Authenticator` implementations:

- `org.jboss.seam.security.jaas.JaasAuthenticator` - used to authenticate against a JAAS configuration defined by the container.
- `org.jboss.seam.security.management.IdmAuthenticator` - used to authenticate against an Identity Store using the Identity Management API. See the Identity Management chapter for details on how to configure this authenticator.
- `org.jboss.seam.security.external.openid.OpenIdAuthenticator` (provided by the external module) - used to authenticate against an external OpenID provider, such as Google, Yahoo, etc. See the External Authentication chapter for details on how to configure this authenticator.

2.3. Which Authenticator will Seam use?

The `Identity` bean has an `authenticatorClass` property, which if set will be used to determine which `Authenticator` bean implementation to invoke during the authentication process. This property may be set by configuring it with a predefined authenticator type, for example by using Solder XML Config. The following XML configuration example shows how you would configure the `Identity` bean to use the `com.acme.MyCustomerAuthenticator` bean for authentication:

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:s="urn:java:ee"
  xmlns:security="urn:java:org.jboss.seam.security"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee   http://jboss.org/schema/cdi/
beans_1_0.xsd">

  <security:IdentityImpl>
    <s:modifies/>
                                  <security:authenticatorClass>com.acme.MyCustomAuthenticator</
security:authenticatorClass>
  </security:IdentityImpl>
```

```
</beans>
```

Alternatively, if you wish to be able to select the `Authenticator` to authenticate with by specifying the name of the `Authenticator` implementation (i.e. for those annotated with the `@Named` annotation), the `authenticatorName` property may be set instead. This might be useful if you wish to offer your users the choice of how they would like to authenticate, whether it be through a local user database, an external OpenID provider, or some other method.

The following example shows how you might configure the `authenticatorName` property with the Seam Config module:

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:s="urn:java:ee"
  xmlns:security="urn:java:org.jboss.seam.security"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee    http://jboss.org/schema/cdi/
beans_1_0.xsd">
  <security:IdentityImpl>
    <s:modifies/>
    <security:authenticatorName>openIdAuthenticator</security:authenticatorName>
  </security:IdentityImpl>
</beans>
```

If neither the `authenticatorClass` or `authenticatorName` properties are set, then the authentication process with automatically use a custom `Authenticator` implementation, if the developer has provided one (and only one) within their application.

If neither property is set, and the user has not provided a custom `Authenticator`, then the authentication process will fall back to the Identity Management API to attempt to authenticate the user.

2.4. Writing a custom Authenticator

All `Authenticator` implementations must implement the `org.jboss.seam.security.Authenticator` interface. This interface defines the following methods:

```
public interface Authenticator {
 void authenticate();
 void postAuthenticate();
 User getUser();
 AuthenticationStatus getStatus();
}
```

The `authenticate()` method is invoked during the authentication process and is responsible for performing the work necessary to validate whether the current user is who they claim to be.

The `postAuthenticate()` method is invoked after the authentication process has already completed, and may be used to perform any post-authentication business logic, such as setting session variables, logging, auditing, etc.

The `getUser()` method should return an instance of `org.picketlink.idm.api.User`, which is generally determined during the authentication process.

The `getStatus()` method must return the current status of authentication, represented by the `AuthenticationStatus` enum. Possible values are `SUCCESS`, `FAILURE` and `DEFERRED`. The `DEFERRED` value should be used for special circumstances, such as asynchronous authentication as a result of authenticating against a third party as is the case with OpenID, etc.

The easiest way to get started writing your own custom authenticator is to extend the `org.jboss.seam.security.BaseAuthenticator` abstract class. This class implements the `getUser()` and `getStatus()` methods for you, and provides `setUser()` and `setStatus()` methods for setting both the user and status values.

Warning

An `Authenticator` implementation cannot be a stateless session bean.

To access the user's credentials from within the `authenticate()` method, you can inject the `Credentials` bean like so:

```
@Inject Credentials credentials;
```

Once the credentials are injected, the `authenticate()` method is responsible for checking that the provided credentials are valid. Here is a complete example:

```
public class SimpleAuthenticator extends BaseAuthenticator implements Authenticator {
  @Inject Credentials credentials;

  @Override
  public void authenticate() {
    if ("demo".equals(credentials.getUsername()) &&
        credentials.getCredential() instanceof PasswordCredential &&
        "demo".equals(((PasswordCredential) credentials.getCredential()).getValue())) {
      setStatus(AuthenticationStatus.SUCCESS);
```

```
         setUser(new SimpleUser("demo"));
      }
   }
}
```

Note

The above code was taken from the simple authentication example, included in the Seam Security distribution.

In the above code, the `authenticate()` method checks that the user has provided a username of *demo* and a password of *demo*. If so, the authentication is deemed as successful and the status is set to `AuthenticationStatus.SUCCESS`, and a new `SimpleUser` instance is created to represent the authenticated user.

Warning

The `Authenticator` implementation *must* return a non-null value when `getUser()` is invoked if authentication is successful. Failure to return a non-null value will result in an `AuthenticationException` being thrown.

Security - Authorization

3.1. Configuration

Before using any of Seam's authorization features, you must enable the `SecurityInterceptor` by adding the following code to your application's `beans.xml`:

```
<interceptors>
  <class>org.jboss.seam.security.SecurityInterceptor</class>
</interceptors>
```

3.2. Basic Concepts

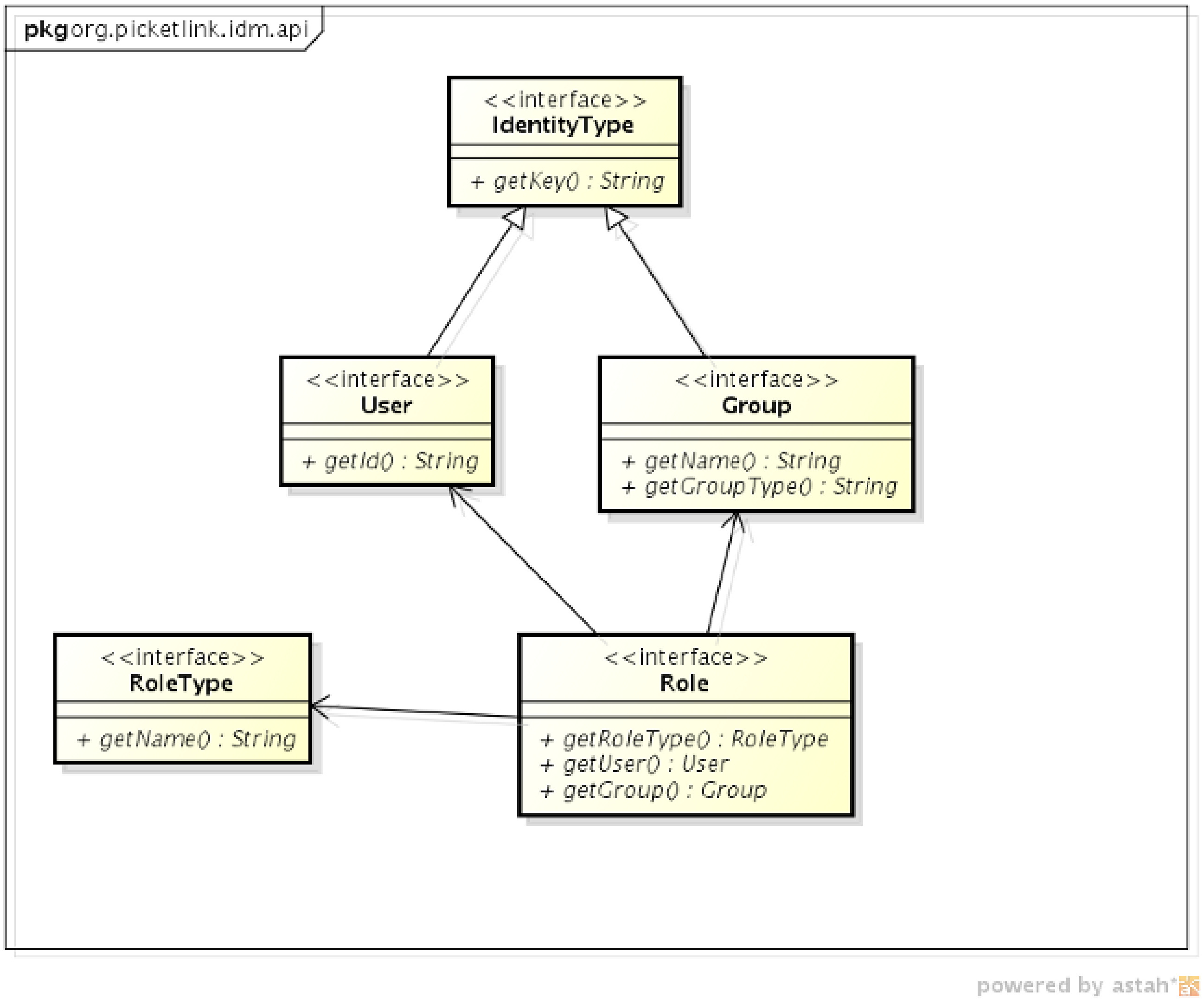
Seam Security provides a number of facilities for restricting access to certain parts of your application. As mentioned previously, the security API is centered around the `Identity` bean, which is a session-scoped bean used to represent the *identity* of the current user.

To be able to restrict the sensitive parts of your code, you may inject the `Identity` bean into your class:

```
@Inject Identity identity;
```

Once you have injected the `Identity` bean, you may invoke its methods to perform various types of authorization. The following sections will examine each of these in more detail.

The security model in Seam Security is based upon the PicketLink API. Let's briefly examine a few of the core interfaces provided by PicketLink that are used in Seam.



3.2.1. IdentityType

This is the common base interface for both `User` and `Group`. The `getKey()` method should return a unique identifying value for the identity type.

3.2.2. User

Represents a user. The `getId()` method should return a unique value for each user.

3.2.3. Group

Represents a group. The `getName()` method should return the name of the group, while the `getGroupType()` method should return the group type.

3.2.4. Role

Represents a role, which is a direct one-to-one typed relationship between a User and a Group. The `getRoleType()` method should return the role type. The `getUser()` method should return the User for which the role is assigned, and the `getGroup()` method should return the Group that the user is associated with.

3.2.5. RoleType

Represents a role type. The `getName()` method should return the name of the role type. Some examples of role types might be `admin`, `superuser`, `manager`, etc.

3.3. Role and Group-based authorization

This is the simplest type of authorization, used to define coarse-grained privileges for users assigned to a certain role or belonging to a certain group. Users may belong to zero or more roles and groups, and inversely, roles and groups may contain zero or more members.

Note

The concept of a *role* in Seam Security is based upon the model defined by PicketLink. I.e, a role is a direct relationship between a user and a group, which consists of three aspects - a member, a role name and a group (see the class diagram above). For example, user *Bob* (the member) may be an *admin* (the role name) user in the *HEAD OFFICE* group.

The `Identity` bean provides the following two methods for checking role membership:

```
boolean hasRole(String role, String group, String groupType);
void checkRole(String role, String group, String groupType);
```

These two methods are similar in function, and both accept the same parameter values. Their behaviour differs when an authorization check fails. The `hasRole()` returns a value of `false` when the current user is not a member of the specified role. The `checkRole()` method on the other hand, will throw an `AuthorizationException`. Which of the two methods you use will depend on your requirements.

The following code listing contains a usage example for the `hasRole()` method:

```
  if (identity.hasRole("manager", "Head Office", "OFFICE")) {
    report.addManagementSummary();
  }
```

Groups can be used to define a collection of users that meet some common criteria. For example, an application might use groups to define users in different geographical locations, their role in the company, their department or division or some other criteria which may be significant from a security point of view. As can be seen in the above class diagram, groups consist of a unique combination of group name and group type. Some examples of group types may be "OFFICE", "DEPARTMENT", "SECURITY_LEVEL", etc. An individual user may belong to many different groups.

The `Identity` bean provides the following methods for checking group membership:

```
boolean inGroup(String name, String groupType);
void checkGroup(String group, String groupType);
```

These methods are similar in behaviour to the role-specific methods above. The `inGroup()` method returns a value of `false` when the current user isn't in the specified group, and the `checkGroup()` method will throw an exception.

3.4. Typesafe authorization

Seam Security provides a way to secure your bean classes and methods by annotating them with a *typesafe security binding*. Each security binding must have a matching authorizer method, which is responsible for performing the business logic required to determine whether a user has the necessary privileges to invoke a bean method. Creating and applying a security binding is quite simple, and is described in the following steps.

3.4.1. Creating a typesafe security binding

A typesafe security binding is an annotation, meta-annotated with the `SecurityBindingType` annotation:

```
import org.jboss.seam.security.annotations.SecurityBindingType;

@SecurityBindingType
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface Admin { }
```

The security binding annotation may also define member values, which are taken into account when matching the annotated bean class or method with an authorizer method. All member values are taken into consideration, except for those annotated with `@Nonbinding`, in much the same way as a qualifier binding type.

```
import javax.enterprise.util.Nonbinding;
import org.jboss.seam.security.annotations.SecurityBindingType;
```

```
@SecurityBindingType
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface Foo {
   String bar();
   @Nonbinding String other() default "";
}
```

3.4.2. Creating an authorizer method

The next step after creating the security binding type is to create a matching authorizer method. This method must contain the business logic required to perform the required authorization check, and return a `boolean` value indicating whether the authorization check passed or failed.

An authorizer method must be annotated with the `@Secures` annotation, and the security binding types for which it is performing the authorization check. An authorizer method may declare zero or more method parameters. Any parameters defined by the authorizer method are treated as injection points, and are automatically injected by the Seam Security extension. The following example demonstrates an authorizer method that injects the `Identity` bean, which is then used to perform the authorization check.

```
import org.jboss.seam.security.annotations.Secures;

public class Restrictions {
  public @Secures @Admin boolean isAdmin(Identity identity) {
    return identity.hasRole("admin", "USERS", "GROUP");
  }
}
```

Note

Authorizer methods will generally make use of the security API to perform their security check, however this is not a hard restriction.

3.4.3. Applying the binding to your business methods

Once the security binding annotation and the matching authorizer method have been created, the security binding type may be applied to a bean class or method. If applied at the class level, every method of the bean class will have the security restriction applied. Methods annotated with a security binding type also inherit any security bindings on their declaring class. Both bean classes and methods may be annotated with multiple security bindings.

```
public @ConversationScoped class UserAction {
  public @Admin void deleteUser(String userId) {
    // code
  }
}
```

If a security check fails when invoking a method annotated with a security binding type, an `AuthorizationException` is thrown. Solder can be used to handle this exception gracefully, for example by redirecting them to an error page or displaying an error message. Here's an example of an exception handler that creates a JSF error message:

```
@HandlesExceptions
public class ExceptionHandler {
   @Inject FacesContext facesContext;
                                          public             void             handleAuthorizationException(@Handles
 CaughtException<AuthorizationException> evt) {
      facesContext.addMessage(null, new FacesMessage(FacesMessage.SEVERITY_ERROR,
            "You do not have the necessary permissions to perform that operation", ""));
      evt.handled();
   }
}
```

3.4.4. Built-in security binding annotations

Seam Security provides one security binding annotation out of the box, `@LoggedIn`. This annotation may be applied to a bean to restrict its methods to only those users that are currently authenticated.

```
import org.jboss.seam.security.annotations.LoggedIn;

public @LoggedIn class CustomerAction {
  public void createCustomer() {
    // code
  }
}
```

Security - Identity Management

4.1. Overview

Identity Management is a feature that allows you to manage the users, groups and roles in your application. The Identity Management features in Seam Security are provided by *PicketLink IDM* [http://www.jboss.org/picketlink/IDM]. The best place to find more information about PicketLink IDM is the reference documentation, available *here* [http://anonsvn.jboss.org/repos/picketlink/idm/downloads/docs/1.0.0.GA/ReferenceGuide/en-US/html_single/index.html].

PicketLink provides two identity store implementations to allow you to use Hibernate or LDAP to store identity-related data (please refer to the PicketLink IDM documentation for details on configuring these). Seam Security provides an additional implementation called `JpaIdentityStore`, which allows you to store your identity data using JPA.

In a Seam-based application it probably makes more sense to use the standards-based `JpaIdentityStore` rather than `HibernateIdentityStore`, as you will most likely be running in an Java EE container that supports JPA. `JpaIdentityStore` is an implementation of the PicketLink `IdentityStore` interface, provided by Seam Security. This identity store allows you to store your identity model inside a relational database, accessible via JPA. It provides an immense amount of flexibility in the way you define your identity model, and in most cases should be compatible with existing database schemas.

Note

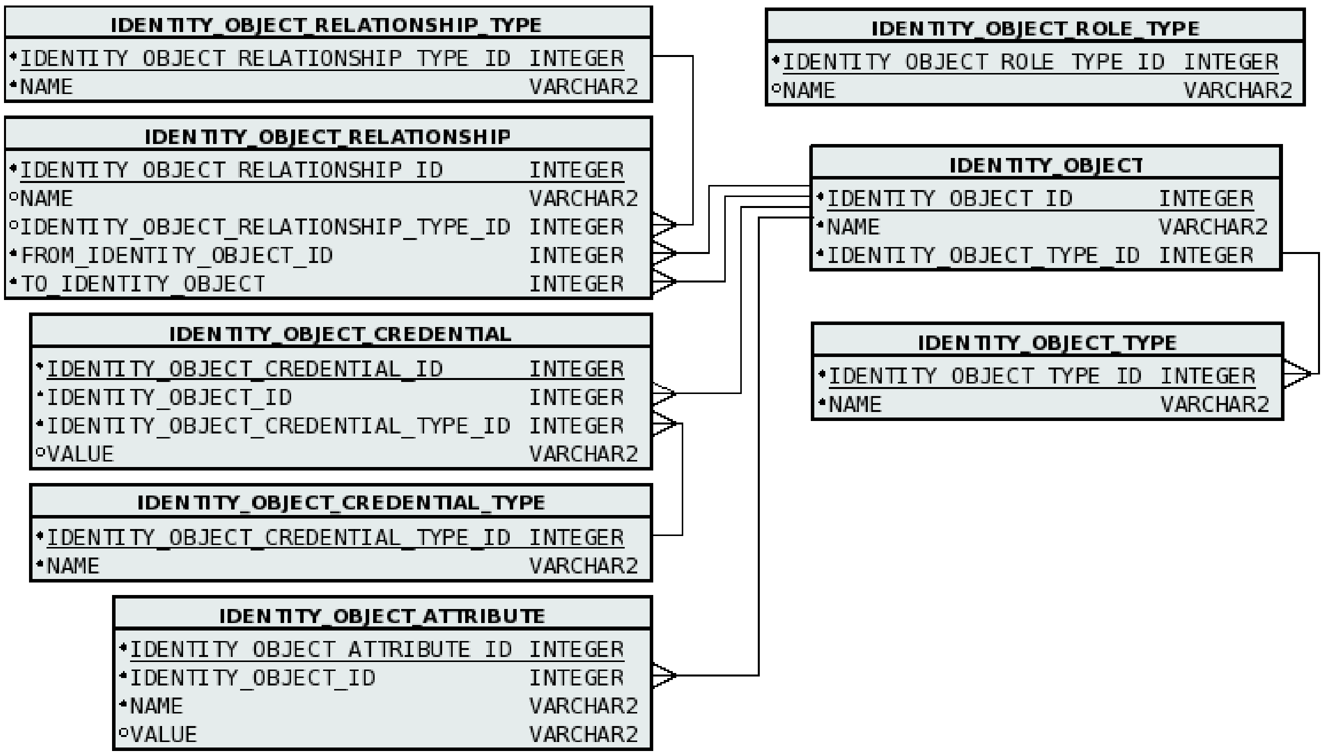
See the idmconsole example application (included in the Seam distribution) for a demonstration of Seam's Identity Management features.

4.2. Configuring Seam to use Identity Management with JPA

Like all authentication providers in Seam, Identity Management is supported via a concrete `Authenticator` implementation called `IdmAuthenticator`. To use Identity Management in your own application, you don't need to do anything! Simply don't configure any authenticator, and as long as you have an identity store configured (see the next section), the Identity Management API will be used to authenticate automatically.

4.2.1. Recommended database schema

While `JpaIdentityStore` should be compatible with a large variety of database schemas, the following diagram displays the recommended database schema to use:


IDENTITY_OBJECT_RELATIONSHIP_TYPE
•IDENTITY_OBJECT_RELATIONSHIP_TYPE_ID INTEGER
•NAME VARCHAR2
IDENTITY_OBJECT_RELATIONSHIP
•IDENTITY_OBJECT_RELATIONSHIP_ID INTEGER
○NAME VARCHAR2
○IDENTITY_OBJECT_RELATIONSHIP_TYPE_ID INTEGER
•FROM_IDENTITY_OBJECT_ID INTEGER
•TO_IDENTITY_OBJECT INTEGER
IDENTITY_OBJECT_CREDENTIAL
•IDENTITY_OBJECT_CREDENTIAL_ID INTEGER
•IDENTITY_OBJECT_ID INTEGER
•IDENTITY_OBJECT_CREDENTIAL_TYPE_ID INTEGER
○VALUE VARCHAR2
IDENTITY_OBJECT_CREDENTIAL_TYPE
•IDENTITY_OBJECT_CREDENTIAL_TYPE_ID INTEGER
•NAME VARCHAR2
IDENTITY_OBJECT_ATTRIBUTE
•IDENTITY_OBJECT_ATTRIBUTE_ID INTEGER
•IDENTITY_OBJECT_ID INTEGER
•NAME VARCHAR2
○VALUE VARCHAR2
IDENTITY_OBJECT_ROLE_TYPE
•IDENTITY_OBJECT_ROLE_TYPE_ID INTEGER
○NAME VARCHAR2
IDENTITY_OBJECT
•IDENTITY_OBJECT_ID INTEGER
•NAME VARCHAR2
•IDENTITY_OBJECT_TYPE_ID INTEGER
IDENTITY_OBJECT_TYPE
•IDENTITY_OBJECT_TYPE_ID INTEGER
•NAME VARCHAR2

4.2.2. The `@IdentityEntity` and `@IdentityProperty` annotations

Seam Security provides two annotations for configuring your entity beans for use with `JpaIdentityStore`. The first, `@IdentityEntity` is a class annotation used to mark an entity bean so that `JpaIdentityStore` knows it contains identity-related data. It has a single member of type `EntityType`, that tells `JpaIdentityStore` what type of identity data it contains. Possible values are:

- IDENTITY_OBJECT
- IDENTITY_CREDENTIAL
- IDENTITY_RELATIONSHIP
- IDENTITY_ATTRIBUTE
- IDENTITY_ROLE_NAME

The second one, `IdentityProperty`, is a field or method annotation which is used to configure which properties of the bean contain identity values. This annotation declares two values, `value` and `attributeName`:

```
package org.jboss.seam.security.annotations.management;

public @interface IdentityProperty {
  PropertyType value();
  String attributeName() default "";
}
```

The `value()` member is of type `PropertyType`, which is an enum that defines the following values:

```
public enum PropertyType {
  NAME, TYPE, VALUE, RELATIONSHIP_FROM, RELATIONSHIP_TO, CREDENTIAL,
  CREDENTIAL_TYPE, ATTRIBUTE }
```

By placing the `IdentityProperty` annotation on various fields of your entity beans, `JpaIdentityStore` can determine how identity-related data must be stored within your database tables.

In the following sections we'll look at how each of the main entities are configured.

4.2.3. Identity Object

Let's start by looking at identity object. In the recommended database schema, the `IDENTITY_OBJECT` table is responsible for storing objects such as users and groups. This table may be represented by the following entity bean:

```
@Entity
@IdentityEntity(IDENTITY_OBJECT)
public class IdentityObject implements Serializable {
   @Id @GeneratedValue private Long id;

   @IdentityProperty(PropertyType.NAME)
   private String name;

   @ManyToOne @IdentityProperty(PropertyType.TYPE)
   @JoinColumn(name = "IDENTITY_OBJECT_TYPE_ID")
   private IdentityObjectType type;

   // snip getter and setter methods
}
```

In the above code both the `name` and `type` fields are annotated with `@IdentityProperty`. This tells `JpaIdentityStore` that these two fields are significant in terms of identity management-related state. By annotating the `name` field with `@IdentityProperty(PropertyType.NAME)`, `JpaIdentityStore` knows that this field is used to store the *name* of the identity object. Likewise, the `@IdentityProperty(PropertyType.TYPE)` annotation on the `type` field indicates that the value of this field is used to represent the *type* of identity object.

The `IdentityObjectType` entity is simply a lookup table containing the names of the valid identity types. The field representing the actual name of the type itself should be annotated with `@IdentityProperty(PropertyType.NAME)`:

```
@Entity
public class IdentityObjectType implements Serializable {

  @Id @GeneratedValue private Long id;
  @IdentityProperty(PropertyType.NAME) private String name;

  // snip getter and setter methods
}
```

4.2.4. Credential

The credentials table is used to store user credentials, such as passwords. Here's an example of an entity bean configured to store identity object credentials:

```
@Entity
@IdentityEntity(IDENTITY_CREDENTIAL)
public class IdentityObjectCredential implements Serializable {
  @Id @GeneratedValue private Long id;

  @ManyToOne @JoinColumn(name = "IDENTITY_OBJECT_ID")
  private IdentityObject identityObject;

  @ManyToOne @IdentityProperty(PropertyType.TYPE)
  @JoinColumn(name = "CREDENTIAL_TYPE_ID")
  private IdentityObjectCredentialType type;

  @IdentityProperty(PropertyType.VALUE)
  private String value;

  // snip getter and setter methods
}
```

The `IdentityObjectCredentialType` entity is used to store a list of valid credential types. Like `IdentityObjectType`, it is a simple lookup table with the field representing the credential type name annotated with `@IdentityProperty(PropertyType.NAME)`:

```
@Entity
```

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

   @IdentityProperty(PropertyType.NAME)
   private String name;

   // snip getter and setter methods
}
```

4.2.5. Identity Object Relationship

The relationship table stores associations between identity objects. Here's an example of an entity bean that has been configured to store identity object relationships:

```
@Entity
@IdentityEntity(IDENTITY_RELATIONSHIP)
public class IdentityObjectRelationship implements Serializable
{
   @Id @GeneratedValue private Long id;

   @IdentityProperty(PropertyType.NAME)
   private String name;

   @ManyToOne @IdentityProperty(PropertyType.TYPE) @JoinColumn(name = "RELATIONSHIP_TYPE_ID")
   private IdentityObjectRelationshipType relationshipType;

   @ManyToOne @IdentityProperty(PropertyType.RELATIONSHIP_FROM) @JoinColumn(name = "FROM_IDENTI
   private IdentityObject from;

   @ManyToOne @IdentityProperty(PropertyType.RELATIONSHIP_TO) @JoinColumn(name = "TO_IDENTITY_ID"
   private IdentityObject to;

   // snip getter and setter methods
}
```

The `name` property is annotated with `@IdentityProperty(PropertyType.NAME)` to indicate that this field contains the name value for named relationships. An example of a named relationship is a role, which uses the `name` property to store the role type name.

The `relationshipType` property is annotated with `@IdentityProperty(PropertyType.TYPE)` to indicate that this field represents the type of relationship. This is typically a value in a lookup table.

The `from` property is annotated with `@IdentityProperty(PropertyType.RELATIONSHIP_FROM)` to indicate that this field represents the `IdentityObject` on the *from* side of the relationship.

The `to` property is annotated with `@IdentityProperty(PropertyType.RELATIONSHIP_TO)` to indicate that this field represents the `IdentityObject` on the *to* side of the relationship.

The `IdentityObjectRelationshipType` entity is a lookup table containing the valid relationship types. The `@IdentityProperty(PropertyType.NAME)` annotation is used to indicate the field containing the relationship type names:

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

  @IdentityProperty(PropertyType.NAME)
  private String name;

  // snip getter and setter methods
}
```

4.2.6. Attributes

The attribute table is used to store any additional information that is to be associated with identity objects. Here's an example of an entity bean used to store attributes:

```
@Entity
@IdentityEntity(IDENTITY_ATTRIBUTE)
public class IdentityObjectAttribute implements Serializable {

  @Id @GeneratedValue private Integer attributeId;

  @ManyToOne
  @JoinColumn(name = "IDENTITY_OBJECT_ID")
  private IdentityObject identityObject;

  @IdentityProperty(PropertyType.NAME)
  private String name;

  @IdentityProperty(PropertyType.VALUE)
  private String value;

  // snip getter and setter methods
}
```

The `name` field is annotated with `@IdentityProperty(PropertyType.NAME)` to indicate that this field contains the attribute name. The `value` field is annotated with `@IdentityProperty(PropertyType.VALUE)` to indicate that this field contains the attribute value.

4.3. Managing Users, Groups and Roles

The Identity Management features are provided by a number of manager objects, which can be access from an `IdentitySession`. The `IdentitySession` may be injected directly into your beans like so:

```
import org.picketlink.idm.api.IdentitySession;

 public @Model class IdentityAction {
   @Inject IdentitySession identitySession;

   // code goes here...
 }
```

Once you have the `IdentitySession` object, you can use it to perform various identity management operations. You should refer to the PicketLink documentation for a complete description of the available features, however the following sections contain a brief overview.

4.3.1. Managing Users and Groups

Users and groups are managed by a `PersistenceManager`, which can be obtained by calling `getPersistenceManager()` on the `IdentitySession` object:

```
PersistenceManager pm = identitySession.getPersistenceManager();
```

Once you have the `PersistenceManager` object, you can create `User` objects with the `createUser()` method:

```
User user = pm.createUser("john");
```

Similarly, you can create `Group` objects with the `createGroup()` method:

```
Group headOffice = pm.createGroup("Head Office", "OFFICE");
```

You can also remove users and groups by calling the `removeUser()` or `removeGroup()` method.

4.3.2. Managing Relationships

Relationships are used to associate `User` objects with `Group` objects. Relationships can be managed with the `RelationshipManager` object, which can be obtained by calling `getRelationshipManager()` on the `IdentitySession`:

```
RelationshipManager rm = identitySession.getRelationshipManager();
```

Relationships are created by invoking the `associateUser()` method, and passing in the group and user objects that should be associated:

```
rm.associateUser(headOffice, user);
```

4.3.3. Managing Roles

Roles are managed via the `RoleManager` object, which can be obtained by invoke the `getRoleManager()` method on the `IdentitySession` object:

```
RoleManager roleManager = identitySession.getRoleManager();
```

Roles are an association between a user and a group, however they are slightly more complex than a simple group membership as the association also has a *role type*. The role type is generally used to describe a particular function of the user within the group. Role types are represented by the `RoleType` object, and can be created with the `createRoleType()` method:

```
RoleType manager = roleManager.createRoleType("manager");
```

Roles can be assigned to users by invoking the `createRole()` method, and passing in the `RoleType`, `User` and `Group`:

```
Role r = roleManager.createRole(manager, user, headOffice);
```

Security - External Authentication

5.1. Introduction

The external authentication module is an optional add-on to the core Seam Security module, which provides a number of features that enable your application to authenticate against third party identity services, via a number of supported protocols.

Warning

The features described in this chapter are a *preview* only. The APIs described may change in a subsequent version of Seam, and may not be backwards-compatible with previous versions.

Currently this module supports authentication via OpenID, and other protocols (such as SAML and OAuth) are currently under development for the next version of Seam.

5.1.1. Configuration

If your project is Maven-based, then add the following dependency to your project:

```
<dependency>
  <groupId>org.jboss.seam.security</groupId>
  <artifactId>seam-security-external</artifactId>
</dependency>
```

If you are not using Maven, you must add the `seam-security-external.jar` library to your project, which can be found in the Seam Security downloadable distribution.

5.2. OpenID

OpenID allows the users of your application to authenticate without requiring them to create an account. When using OpenID, your user is temporarily redirected to the web site of their OpenID provider so that they can enter their password, after which they are redirected back to your application. The OpenID authentication process is safe - at no time is the user's password seen by any site besides their OpenID provider.

5.2.1. Overview

The external authentication module provides support for OpenID based on *OpenID4Java* [http://code.google.com/p/openid4java/], an open source OpenID library (licensed under the Apache v2 license) with both Relying Party and Identity Provider capabilities. This feature allows your application to authenticate its users against an external OpenID provider, such as Google or Yahoo, or to turn your application into an OpenID provider itself.

Note

To see the OpenID features in action, take a look at the `openid-rp` example included in the Seam Security distribution.

5.2.2. Enabling OpenID for your application

To use OpenID in your own application, you must configure Seam Security to use `OpenIdAuthenticator`, an `Authenticator` implementation that performs authentication against an OpenID provider. This authenticator is a named, session-scoped bean, with the following declaration:

```
public @Named("openIdAuthenticator") @SessionScoped class OpenIdAuthenticator
```

5.2.2.1. Using OpenID as your only authentication method

If your application only uses OpenID to provide authentication services, then it is recommended that `OpenIdAuthenticator` is selected by configuring the `authenticatorClass` property of the `Identity` bean. The following code sample demonstrates how this might be done by using Solder:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:s="urn:java:ee"
  xmlns:security="urn:java:org.jboss.seam.security"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://jboss.org/schema/cdi/beans_1_0.xsd">

  <security:Identity>
    <s:modifies/>
    <security:authenticatorClass>org.jboss.seam.security.external.openid.OpenIdAuthenticator</security:authenticatorClass>
  </security:Identity>
```

5.2.2.2. Using OpenID as one of many possible authentication methods

If your application gives the user a choice of which authentication method to use, then it is not possible to pre-configure which `Authenticator` implementation is used to authenticate. In these circumstances, it is recommended that you configure the authenticator by specifying a value for the `authenticatorName` property of the `Identity` bean. This can be done by binding a view-layer control such as a radio group directly to this property, to allow the user to select the method of authentication they wish to use. See the following JSF code as an example:

```
<h:outputLabel value="Authenticate using:"/>
<h:selectOneRadio id="authenticator" value="#{identity.authenticatorName}">
  <f:selectItem itemLabel="OpenID" itemValue="openIdAuthenticator" />
  <f:selectItem itemLabel="Custom" itemValue="customAuthenticator" />
</h:selectOneRadio>
```

5.2.3. Choosing which OpenID provider to use

Seam provides built-in support for a number of well-known OpenID providers. The `OpenIdAuthenticator` bean may be configured to select which OpenID provider will be used to process an authentication request. Each concrete provider implements the following interface:

```
public interface OpenIdProvider {
   String getCode();
   String getName();
   String getUrl();
}
```

The following table lists the providers that come pre-packaged in Seam:

Provider	Code	Name	URL
CustomOpenIdProvider	custom	Google	
GoogleOpenIdProvider	google	Google	https://www.google.com/accounts/o8/id
MyOpenIdProvider	myopenid	MyOpenID	https://myopenid.com
YahooOpenIdProvider	yahoo	Yahoo	https://me.yahoo.com

To select one of the built-in providers to use for an authentication request, the `providerCode` property of the `OpenIdAuthenticator` bean should be set to one of the Code values from the above table. The `OpenIdAuthenticator` bean provides a convenience method called `getProviders()` that returns a list of all known providers. This may be used in conjunction with a radio group to allow the user to select which OpenID provider they wish to authenticate with - see the following JSF snippet for an example:

```
<h:selectOneRadio value="#{openIdAuthenticator.providerCode}">
<f:selectItems value="#{openIdAuthenticator.providers}" var="p" itemValue="#{p.code}" itemLabel="#{p.name}"/
>
</h:selectOneRadio>
```

5.2.3.1. Using a custom OpenID provider

If you would like to allow your users to specify an OpenID provider that is not supported out of the box by Seam, then the `CustomOpenIdProvider` may be used. As it is a `@Named` bean, it can be accessed directly from the view layer via EL. The following JSF code shows how you might allow the user to specify their own OpenID provider:

```
<h:outputLabel value="If you have selected the Custom OpenID provider, please provide a URL:"/
>
<h:inputText value="#{customOpenIdProvider.url}"/>
```

5.2.4. Managing the OpenID authentication process

Your application must provide an implementation of the `OpenIdRelyingPartySpi` interface to process OpenID callback events. This interface declares the following methods:

```
public interface OpenIdRelyingPartySpi {
   void loginSucceeded(OpenIdPrincipal principal, ResponseHolder responseHolder);
   void loginFailed(String message, ResponseHolder responseHolder);
```

The implementation is responsible for processing the response of the OpenID authentication, and is typically used to redirect the user to an appropriate page depending on whether authentication was successful or not.

There are two API calls that *must* be made in the case of a successful authentication. The first one should notify the `OpenIdAuthenticator` that the authentication attempt was successful, and pass it the `OpenIdPrincipal` object:

Warning

If the following two API calls are omitted, unpredictable results may occur!

```
openIdAuthenticator.success(principal);
```

Secondly, a `DeferredAuthenticationEvent` must be fired to signify that a deferred authentication attempt has been completed:

```
deferredAuthentication.fire(new DeferredAuthenticationEvent());
```

After making these two API calls, the implementation may perform whatever additional logic is required. The following code shows a complete example:

```
import java.io.IOException;

import javax.enterprise.event.Event;
import javax.inject.Inject;
import javax.servlet.ServletContext;

import org.jboss.seam.security.events.DeferredAuthenticationEvent;
import org.jboss.seam.security.external.api.ResponseHolder;
import org.jboss.seam.security.external.openid.OpenIdAuthenticator;
import org.jboss.seam.security.external.openid.api.OpenIdPrincipal;
import org.jboss.seam.security.external.spi.OpenIdRelyingPartySpi;

public class OpenIdRelyingPartySpiImpl implements OpenIdRelyingPartySpi {
  @Inject private ServletContext servletContext;
  @Inject OpenIdAuthenticator openIdAuthenticator;
  @Inject Event<DeferredAuthenticationEvent> deferredAuthentication;

  public void loginSucceeded(OpenIdPrincipal principal, ResponseHolder responseHolder) {
    try {
      openIdAuthenticator.success(principal);
      deferredAuthentication.fire(new DeferredAuthenticationEvent());

              responseHolder.getResponse().sendRedirect(servletContext.getContextPath() + "/
UserInfo.jsf");
    } catch (IOException e) {
        throw new RuntimeException(e);
      }
   }

   public void loginFailed(String message, ResponseHolder responseHolder) {
      try {
               responseHolder.getResponse().sendRedirect(servletContext.getContextPath() + "/
AuthenticationFailed.jsf");
      } catch (IOException e) {
        throw new RuntimeException(e);
      }
   }
}
```

Security - Events

6.1. Introduction

A number of CDI events are fired during the course of many security-related operations, allowing additional business logic to be executed in response to certain security events. This is useful if you would like to generate additional logging or auditing, or produce messages to display to the user.

6.2. Event list

The following table contains the list of event classes that may be fired by Seam Security, along with a description of when the event is fired. All event classes are contained in the `org.jboss.seam.security.events` package.

Event	Description
AlreadyLoggedInEvent	Fired when a user who is already logged in attempts to log in again
AuthorizationCheckEvent	Fired when an authorization check is performed, such as `Identity.hasPermission()`.
CredentialsUpdatedEvent	Fired whenever a user's credentials (such as their username or password) are updated.
DeferredAuthenticationEvent	Fired when a deferred authentication occurs. For example, at the end of the OpenID authentication process when the OpenID provider redirects the user back to the application.
LoggedInEvent	Fired when the user is successfully logged in.
LoginFailedEvent	Fired when an authentication attempt by the user fails.
NotAuthorizedEvent	Fired when the user is not authorized to invoke a particular operation.
NotLoggedInEvent	Fired when the user attempts to invoke a privileged operation before they have authenticated.
PreAuthenticateEvent	Fired just before a user is authenticated
PostAuthenticateEvent	Fired after a user has authenticated successfully.
PreLoggedOutEvent	Fired just before a user is logged out.
PostLoggedOutEvent	Fired after a user has logged out.

Event	Description
PrePersistUserEvent	Fired just before a new user is persisted (when using Identity Management).
PrePersistUserRoleEvent	Fired just before a new user role is persisted (when using Identity Management).
QuietLoginEvent	Fired when a user is quietly authenticated.
SessionInvalidatedEvent	Fired when a user's session is invalidated.
UserAuthenticatedEvent	Fired when a user is authenticated.
UserCreatedEvent	

6.3. Usage Example

The following code listing shows the `SecurityEventMessages` class, from the Seam Security implementation library. This class (which is disabled by default due to the `@Veto` annotation) uses the Messages API from Seam International to generate user-facing messages in response to certain security events.

```
package org.jboss.seam.security;

import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.event.Observes;

import org.jboss.seam.international.status.Messages;
import org.jboss.seam.security.events.AlreadyLoggedInEvent;
import org.jboss.seam.security.events.LoggedInEvent;
import org.jboss.seam.security.events.LoginFailedEvent;
import org.jboss.seam.security.events.NotLoggedInEvent;
import org.jboss.seam.security.events.PostAuthenticateEvent;
import org.jboss.solder.core.Requires;
import org.jboss.solder.core.Veto;

public @ApplicationScoped @Veto @Requires("org.jboss.seam.international.status.Messages")
class SecurityEventMessages {
   private static final String DEFAULT_LOGIN_FAILED_MESSAGE = "Login failed - please check
 your username and password before trying again.";
    private static final String DEFAULT_LOGIN_SUCCESSFUL_MESSAGE = "Welcome, {0}.";
     private static final String DEFAULT_ALREADY_LOGGED_IN_MESSAGE = "You're already
 logged in. Please log out first if you wish to log in again.";
    private static final String DEFAULT_NOT_LOGGED_IN_MESSAGE = "Please log in first.";

    public void postAuthenticate(@Observes PostAuthenticateEvent event, Messages messages,
 Identity identity) {
        messages.info(DEFAULT_LOGIN_SUCCESSFUL_MESSAGE, identity.getUser().getId());
    }

    public void addLoginFailedMessage(@Observes LoginFailedEvent event, Messages messages) {
        messages.error(DEFAULT_LOGIN_FAILED_MESSAGE);
    }

    public void addLoginSuccessMessage(@Observes LoggedInEvent event, Messages messages, Credentials credentials) {
        messages.info(DEFAULT_LOGIN_SUCCESSFUL_MESSAGE, credentials.getUsername());
    }

    public void addAlreadyLoggedInMessage(@Observes AlreadyLoggedInEvent event, Messages messages) {
        messages.error(DEFAULT_ALREADY_LOGGED_IN_MESSAGE);
    }

    public void addNotLoggedInMessage(@Observes NotLoggedInEvent event, Messages messages) {
        messages.error(DEFAULT_NOT_LOGGED_IN_MESSAGE);
    }
}
```