Seam - Contextual Components

A Framework for Java EE 5

Version: 1.1.1.GA

Table of Contents

Introduction to JBoss Seam

Seam is an application framework for Java EE 5. It is inspired by the following principles:

Integrate JSF with EJB 3.0

JSF and EJB 3.0 are two of the best new features of Java EE 5. EJB3 is a brand new component model for server side business and persistence logic. Meanwhile, JSF is a great component model for the presentation tier. Unfortunately, neither component model is able to solve all problems in computing by itself. Indeed, JSF and EJB3 work best used together. But the Java EE 5 specification provides no standard way to integrate the two component models. Fortunately, the creators of both models foresaw this situation and provided standard extension points to allow extension and integration of other solutions.

Seam unifies the component models of JSF and EJB3, eliminating glue code, and letting the developer think about the business problem.

Integrated AJAX

Seam supports two open source JSF-based AJAX solutions: ICEfaces and Ajax4JSF. These solutions let you add AJAX capability to your user interface without the need to write any JavaScript code.

Seam also provides a built-in JavaScript remoting layer for EJB3 components. AJAX clients can easily call server-side components and subscribe to JMS topics, without the need for an intermediate action layer.

Neither of these approaches would work well, were it not for Seam's built-in concurrency and state management, which ensures that many concurrent fine-grained, asynchronous AJAX requests are handled safely and efficiently on the server side.

Integrate Business Process as a First Class Construct

Optionally, Seam integrates transparent business process management via jBPM. You won't believe how easy it is to implement complex workflows using jBPM and Seam.

Seam even allows definition of presentation tier conversation flow by the same means.

JSF provides an incredibly rich event model for the presentation tier. Seam enhances this model by exposing jBPM's business process related events via exactly the same event handling mechanism, providing a uniform event model for Seam's uniform component model.

One Kind of "Stuff"

Seam provides a uniform component model. A Seam component may be stateful, with the state associated to any one of a number of contexts, ranging from the long-running business process to a single web request.

There is no distinction between presentation tier components and business logic components in Seam. It is possible to write Seam applications where "everything" is an EJB. This may come as a surprise if you are used to thinking of EJBs as coarse-grained, heavyweight objects that are a pain in the backside to create! However, EJB 3.0 completely changes the nature of EJB from the point of view of the developer. An EJB is a fine-grained object - nothing more complex than an annotated JavaBean. Seam even encourages you to use session beans as JSF action listeners!

Unlike plain Java EE or J2EE components, Seam components may *simultaneously* access state associated with the web request and state held in transactional resources (without the need to propagate web request state manually via method parameters). You might object that the application layering imposed upon you by the old J2EE platform was a Good Thing. Well, nothing stops you creating an equivalent layered architecture using Seam - the difference is that *you* get to architect your own application and decide what the

layers are and how they work together.

Declarative State Management

We are all used to the concept of declarative transaction management and J2EE declarative security from EJB 2.x. EJB 3.0 even introduces declarative persistence context management. These are three examples of a broader problem of managing state that is associated with a particular *context*, while ensuring that all needed cleanup occurs when the context ends. Seam takes the concept of declarative state management much further and applies it to *application state*. Traditionally, J2EE applications almost always implement state management manually, by getting and setting servlet session and request attributes. This approach to state management is the source of many bugs and memory leaks when applications fail to clean up session attributes, or when session data associated with different workflows collides in a multi-window application. Seam has the potential to almost entirely eliminate this class of bugs.

Declarative application state management is made possible by the richness of the *context model* defined by Seam. Seam extends the context model defined by the servlet spec—request, session, application—with two new contexts—conversation and business process—that are more meaningful from the point of view of the business logic.

Bijection

The notion of *Inversion of Control* or *dependency injection* exists in both JSF and EJB3, as well as in numerous so-called "lighweight containers". Most of these containers emphasize injection of components that implement *stateless services*. Even when injection of stateful components is supported (such as in JSF), it is virtually useless for handling application state because the scope of the stateful component cannot be defined with sufficient flexibility.

Bijection differs from IoC in that it is *dynamic*, *contextual*, and *bidirectional*. You can think of it as a mechanism for aliasing contextual variables (names in the various contexts bound to the current thread) to attributes of the component. Bijection allows auto-assembly of stateful components by the container. It even allows a component to safely and easily manipulate the value of a context variable, just by assigning to an attribute of the component.

Workspace Management

Optionally, Seam applications may take advantage of *workspace management*, allowing users to freely switch between different conversations (workspaces) in a single browser window. Seam provides not only correct multi-window operation, but also multi-window-like operation in a single window!

Annotated POJOs Everywhere

EJB 3.0 embraces annotations and "configuration by exception" as the easiest way to provide information to the container in a declarative form. Unfortunately, JSF is still heavily dependent on verbose XML configuration files. Seam extends the annotations provided by EJB 3.0 with a set of annotations for declarative state management and declarative context demarcation. This lets you eliminate the noisy JSF managed bean declarations and reduce the required XML to just that information which truly belongs in XML (the JSF navigation rules).

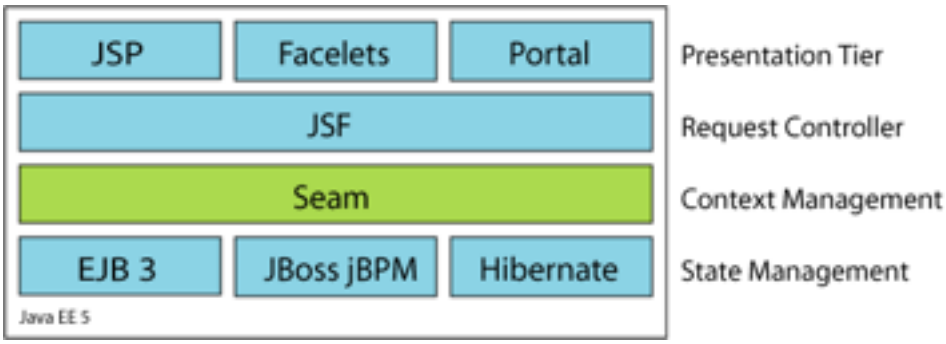
Testability as a Core Feature

Seam components, being POJOs, are by nature unit testable. But for complex applications, unit testing alone is insufficient. Integration testing has traditionally been a messy and difficult task for Java web applications. Therefore, Seam provides for testability of Seam applications as a core feature of the framework. You can easily write JUnit or TestNG tests that reproduce a whole interaction with a user, exercising all components of the system apart from the view (the JSP or Facelets page). You can run these tests directly inside your IDE, where Seam will automatically deploy EJB components into the JBoss Embeddable EJB3 container.

Get started now!

Seam works in any application server that supports EJB 3.0. You can even use Seam in a servlet container like Tomcat, or in any J2EE application server, by leveraging the new JBoss Embeddable EJB3 container.

However, we realize that not everyone is ready to make the switch to EJB 3.0. So, in the interim, you can use Seam as a framework for applications that use JSF for presentation, Hibernate (or plain JDBC) for persistence and JavaBeans for application logic. Then, when you're ready to make the switch to EJB 3.0, migration will be straightforward.



It turns out that the combination of Seam, JSF and EJB3 is *the* simplest way to write a complex web application in Java. You won't believe how little code is required!

Chapter 1. Seam Tutorial

1.1. Try the examples

In this tutorial, we'll assume that you have downloaded JBoss AS 4.0.5 and installed the EJB 3.0 profile (using the JBoss AS installer). You should also have a copy of Seam downloaded and extracted to a work directory.

The directory structure of each example in Seam follows this pattern:

- Web pages, images and stylesheets may be found in `examples/registration/view`
- Resources such as deployment descriptors and data import scripts may be found in `examples/registration/resources`
- Java source code may be found in `examples/registration/src`
- The Ant build script is `examples/registration/build.xml`

1.1.1. Running the examples on JBoss AS

First, make sure you have Ant correctly installed, with `$ANT_HOME` and `$JAVA_HOME` set correctly. Next, make sure you set the location of your JBoss AS 4.0.5 installation in the `build.properties` file in the root folder of your Seam installation. If you haven't already done so, start JBoss AS now by typing `bin/run.sh` or `bin/run.bat` in the root directory of your JBoss installation.

Now, build and deploy the example by typing `ant deploy` in the `examples/registration` directory.

Try it out by accessing `http://localhost:8080/seam-registration/` with your web browser.

1.1.2. Running the examples on Tomcat

First, make sure you have Ant correctly installed, with `$ANT_HOME` and `$JAVA_HOME` set correctly. Next, make sure you set the location of your Tomcat 5.5 installation in the `build.properties` file in the root folder of your Seam installation.

Now, build and deploy the example by typing `ant deploy.tomcat` in the `examples/registration` directory.

Finally, start Tomcat.

Try it out by accessing `http://localhost:8080/jboss-seam-registration/` with your web browser.

When you deploy the example to Tomcat, any EJB3 components will run inside the JBoss Embeddable EJB3 container, a complete standalone EJB3 container environment.

1.1.3. Running the example tests

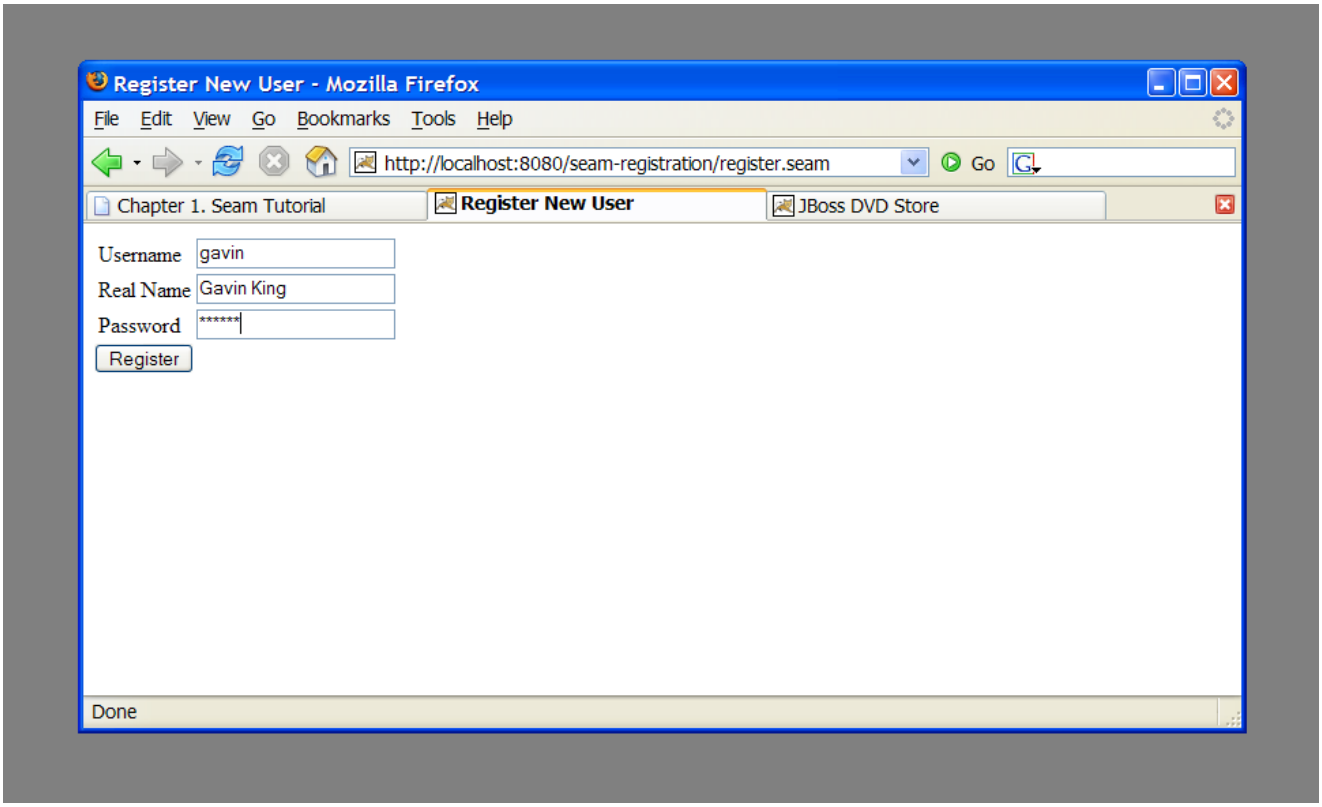
Most of the examples come with a suite of TestNG integration tests. The easiest way to run the tests is to run `ant testexample` inside the `examples/registration` directory. It is also possible to run the tests inside your IDE using the TestNG plugin.

1.2. Your first Seam application: the registration example

The registration example is a fairly trivial application that lets a new user store his username, real name and password in the database. The example isn't intended to show off all of the cool functionality of Seam. However, it demonstrates the use of an EJB3 session bean as a JSF action listener, and basic configuration of Seam.

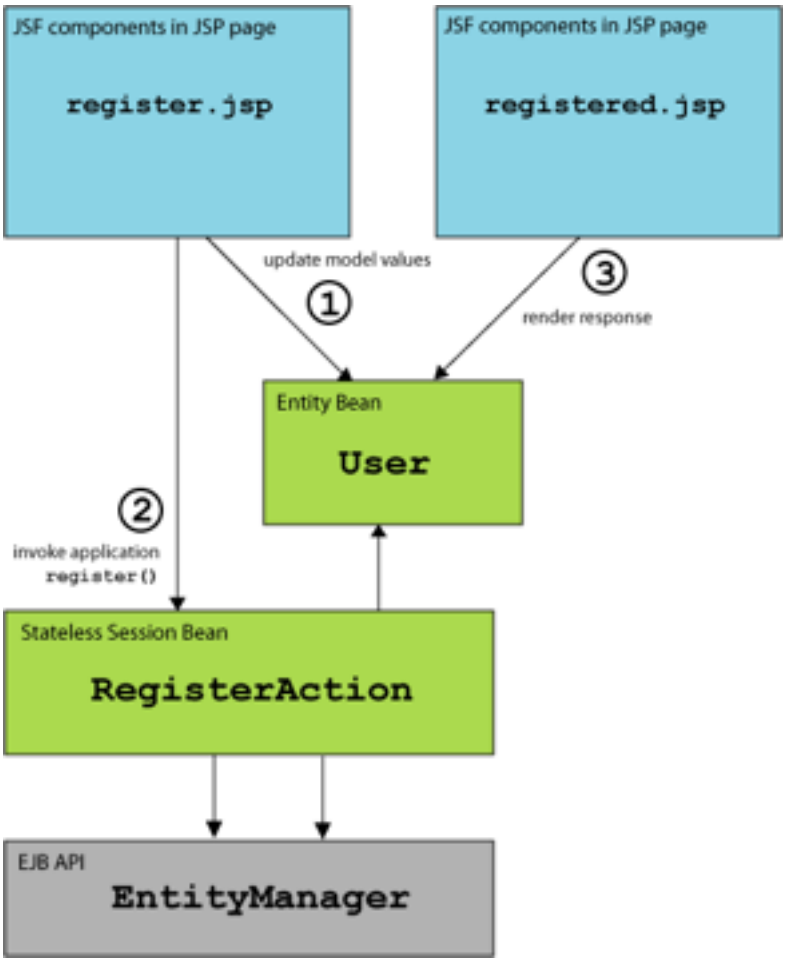
We'll go slowly, since we realize you might not yet be familiar with EJB 3.0.

The start page displays a very basic form with three input fields. Try filling them in and then submitting the form. This will save a user object in the database.



1.2.1. Understanding the code

The example is implemented with two JSP pages, one entity bean and one stateless session bean.



Let's take a look at the code, starting from the "bottom".

1.2.1.1. The entity bean: `User.java`

We need an EJB entity bean for user data. This class defines *persistence* and *validation* declaratively, via annotations. It also needs some extra annotations that define the class as a Seam component.

Example 1.1.

```
@Entity                                                                    (1)
@Name("user")                                                              (2)
@Scope(SESSION)                                                            (3)
@Table(name="users")                                                       (4)
public class User implements Serializable
{
   private static final long serialVersionUID = 1881413500711441951L;

   private String username;                                                (5)
   private String password;
   private String name;

   public User(String name, String password, String username)
   {
      this.name = name;
      this.password = password;
      this.username = username;
   }

   public User() {}                                                        (6)

   @NotNull @Length(min=5, max=15)                                         (7)
   public String getPassword()
```

```
   {
      return password;
   }

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

   @NotNull
   public String getName()
   {
      return name;
   }

   public void setName(String name)
   {
      this.name = name;
   }

   @Id @NotNull @Length(min=5, max=15)                                    (8)
   public String getUsername()
   {
      return username;
   }

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

}
```

(1) The EJB3 standard `@Entity` annotation indicates that the `User` class is an entity bean.

(2) A Seam component needs a *component name* specified by the `@Name` annotation. This name must be unique within the Seam application. When JSF asks Seam to resolve a context variable with a name that is the same as a Seam component name, and the context variable is currently undefined (null), Seam will instantiate that component, and bind the new instance to the context variable. In this case, Seam will instantiate a `User` the first time JSF encounters a variable named `user`.

(3) Whenever Seam instantiates a component, it binds the new instance to a context variable in the component's *default context*. The default context is specified using the `@Scope` annotation. The `User` bean is a session scoped component.

(4) The EJB standard `@Table` annotation indicates that the `User` class is mapped to the `users` table.

(5) `name`, `password` and `username` are the persistent attributes of the entity bean. All of our persistent attributes define accessor methods. These are needed when this component is used by JSF in the render response and update model values phases.

(6) An empty constructor is both required by both the EJB specification and by Seam.

(7) The `@NotNull` and `@Length` annotations are part of the Hibernate Validator framework. Seam integrates Hibernate Validator and lets you use it for data validation (even if you are not using Hibernate for persistence).

(8) The EJB standard `@Id` annotation indicates the primary key attribute of the entity bean.

The most important things to notice in this example are the `@Name` and `@Scope` annotations. These annotations establish that this class is a Seam component.

We'll see below that the properties of our `User` class are bound to directly to JSF components and are populated by JSF during the update model values phase. We don't need any tedious glue code to copy data back and forth between the JSP pages and the entity bean domain model.

However, entity beans shouldn't do transaction management or database access. So we can't use this component as a JSF action listener. For that we need a session bean.

1.2.1.2. The stateless session bean class: `RegisterAction.java`

Most Seam application use session beans as JSF action listeners (you can use JavaBeans instead if you like).

We have exactly one JSF action in our application, and one session bean method attached to it. In this case, we'll use a stateless session bean, since all the state associated with our action is held by the `User` bean.

This is the only really interesting code in the example!

Example 1.2.

```
@Stateless                                                                    (1)
@Name("register")
public class RegisterAction implements Register
{

   @In                                                                        (2)
   private User user;

   @PersistenceContext                                                        (3)
   private EntityManager em;

   @Logger                                                                    (4)
   private Log log;

   public String register()                                                   (5)
   {
      List existing = em.createQuery("select username from User where username=:username")
         .setParameter("username", user.getUsername())
         .getResultList();

      if (existing.size()==0)
      {
         em.persist(user);
         log.info("Registered new user #{user.username}");                    (6)
         return "/registered.jsp";                                            (7)
      }
      else
      {
         FacesMessages.instance().add("User #{user.username} already exists"); (8)
         return null;
      }
   }

}
```

(1) The EJB standard `@Stateless` annotation marks this class as stateless session bean.

(2) The `@In` annotation marks an attribute of the bean as injected by Seam. In this case, the attribute is injected from a context variable named `user` (the instance variable name).

(3) The EJB standard `@PersistenceContext` annotation is used to inject the EJB3 entity manager.

(4) The Seam `@Logger` annotation is used to inject the component's `Log` instance.

(5) The action listener method uses the standard EJB3 `EntityManager` API to interact with the database, and returns the JSF outcome. Note that, since this is a sesson bean, a transaction is automatically begun when the `register()` method is called, and committed when it completes.

(6) The `Log` API lets us easily display templated log messages.

(7) JSF action listener methods return a string-valued outcome that determines what page will be displayed next. A null outcome (or a void action listener method) redisplays the previous page. In plain JSF, it is normal to always use a JSF *navigation rule* to determine the JSF view id from the outcome. For complex application this indirection is useful and a good practice. However, for very simple examples like this one, Seam lets you use the JSF view id as the outcome, eliminating the requirement for a navigation rule. *Note*

that when you use a view id as an outcome, Seam always performs a browser redirect.

(8) Seam provides a number of *built-in components* to help solve common problems. The `FacesMessages` component makes it easy to display templated error or success messages. Built-in Seam components may be obtained by injection, or by calling an `instance()` method.

Note that we did not explicitly specify a `@Scope` this time. Each Seam component type has a default scope if not explicitly specified. For stateless session beans, the default scope is the stateless context. Actually, *all* stateless session beans belong in the stateless context.

Our session bean action listener performs the business and persistence logic for our mini-application. In more complex applications, we might need to layer the code and refactor persistence logic into a dedicated data access component. That's perfectly trivial to do. But notice that Seam does not force you into any particular strategy for application layering.

Furthermore, notice that our session bean has simultaneous access to context associated with the web request (the form values in the `User` object, for example), and state held in transactional resources (the `EntityManager` object). This is a break from traditional J2EE architectures. Again, if you are more comfortable with the traditional J2EE layering, you can certainly implement that in a Seam application. But for many applications, it's simply not very useful.

1.2.1.3. The session bean local interface: `Register.java`

Naturally, our session bean needs a local interface.

Example 1.3.

```
@Local
public interface Register
{
   public String register();
}
```

That's the end of the Java code. Now onto the deployment descriptors.

1.2.1.4. The Seam component deployment descriptor: `components.xml`

If you've used many Java frameworks before, you'll be used to having to declate all your component classes in some kind of XML file that gradually grows more and more unmanageable as your project matures. You'll be relieved to know that Seam does not require that application components be accompanied by XML. Most Seam applications require a very small amount of XML that does not grow very much as the project gets bigger.

Nevertheless, it is often useful to be able to provide for *some* external configuration of *some* components (particularly the components built in to Seam). You have a couple of options here, but the most flexible option is to provide this configuration in a file called `components.xml`, located in the `WEB-INF` directory. We'll use the `components.xml` file to tell Seam how to find our EJB components in JNDI:

Example 1.4.

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core">
    <core:init jndi-pattern="@jndiPattern@"/>
</components>
```

This code configures a property named `jndiPattern` of a built-in Seam component named `org.jboss.seam.core.init`.

1.2.1.5. The web deployment description: `web.xml`

The presentation layer for our mini-application will be deployed in a WAR. So we'll need a web deployment descriptor.

Example 1.5.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
    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">

    <!-- Seam -->

    <listener>
        <listener-class>org.jboss.seam.servlet.SeamListener</listener-class>
    </listener>

    <!-- MyFaces -->

    <listener>
        <listener-class>
            org.apache.myfaces.webapp.StartupServletContextListener
        </listener-class>
    </listener>

    <context-param>
        <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
        <param-value>client</param-value>
    </context-param>

    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <!-- Faces Servlet Mapping -->
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.seam</url-pattern>
    </servlet-mapping>

</web-app>
```

This `web.xml` file configures Seam and MyFaces. The configuration you see here is pretty much identical in all Seam applications.

1.2.1.6. The JSF configration: `faces-config.xml`

All Seam applications use JSF views as the presentation layer. So we'll need `faces-config.xml`.

Example 1.6.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE faces-config
PUBLIC "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
                           "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
<faces-config>

    <!-- A phase listener is needed by all Seam applications -->

    <lifecycle>
        <phase-listener>org.jboss.seam.jsf.SeamPhaseListener</phase-listener>
    </lifecycle>

</faces-config>
```

The `faces-config.xml` file integrates Seam into JSF. Note that we don't need any JSF managed bean declarations! The managed beans are the Seam components. In Seam applications, the `faces-config.xml` is used much less often than in plain JSF.

In fact, once you have all the basic descriptors set up, the *only* XML you need to write as you add new functionality to a Seam application is the navigation rules, and possibly jBPM process definitions. Seam takes the view that *process flow* and *configuration data* are the only things that truly belong in XML.

In this simple example, we don't even need a navigation rule, since we decided to embed the view id in our action code.

1.2.1.7. The EJB deployment descriptor: `ejb-jar.xml`

The `ejb-jar.xml` file integrates Seam with EJB3, by attaching the `SeamInterceptor` to all session beans in the archive.

```
<ejb-jar 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/ejb-j
         version="3.0">

   <interceptors>
     <interceptor>
       <interceptor-class>org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
     </interceptor>
   </interceptors>

   <assembly-descriptor>
      <interceptor-binding>
         <ejb-name>*</ejb-name>
         <interceptor-class>org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
      </interceptor-binding>
   </assembly-descriptor>

</ejb-jar>
```

1.2.1.8. The EJB persistence deployment descriptor: `persistence.xml`

The `persistence.xml` file tells the EJB persistence provider where to find the datasource, and contains some vendor-specific settings. In this case, enables automatic schema export at startup time.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/per
             version="1.0">
    <persistence-unit name="userDatabase">
      <provider>org.hibernate.ejb.HibernatePersistence</provider>
      <jta-data-source>java:/DefaultDS</jta-data-source>
```

```
      <properties>
         <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
      </properties>
    </persistence-unit>
</persistence>
```

1.2.1.9. The view: `register.jsp` and `registered.jsp`

The view pages for a Seam application could be implemented using any technology that supports JSF. In this example we use JSP, since it is familiar to most developers and since we have minimal requirements here anyway. (But if you take our advice, you'll use Facelets for your own applications.)

Example 1.7.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://jboss.com/products/seam/taglib" prefix="s" %>
<html>
 <head>
  <title>Register New User</title>
 </head>
 <body>
  <f:view>
   <h:form>
     <table border="0">
       <s:validateAll>
         <tr>
           <td>Username</td>
           <td><h:inputText value="#{user.username}"/></td>
         </tr>
         <tr>
           <td>Real Name</td>
           <td><h:inputText value="#{user.name}"/></td>
         </tr>
         <tr>
           <td>Password</td>
           <td><h:inputSecret value="#{user.password}"/></td>
         </tr>
       </s:validateAll>
     </table>
     <h:messages/>
     <h:commandButton type="submit" value="Register" action="#{register.register}"/>
   </h:form>
  </f:view>
 </body>
</html>
```

The only thing here that is specific to Seam is the `<s:validateAll>` tag. This JSF component tells JSF to validate all the contained input fields against the Hibernate Validator annotations specified on the entity bean.

Example 1.8.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<html>
 <head>
  <title>Successfully Registered New User</title>
 </head>
 <body>
  <f:view>
    Welcome, <h:outputText value="#{user.name}"/>,
```

```
    you are successfully registered as <h:outputText value="#{user.username}"/>.
  </f:view>
 </body>
</html>
```

This is a boring old JSP pages using standard JSF components. There is nothing specific to Seam here.

1.2.1.10. The EAR deployment descriptor: `application.xml`

Finally, since our application is deployed as an EAR, we need a deployment descriptor there, too.

Example 1.9.

```
<?xml version="1.0" encoding="UTF-8"?>
<application 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/a
             version="5">

    <display-name>Seam Registration</display-name>

    <module>
        <web>
            <web-uri>jboss-seam-registration.war</web-uri>
            <context-root>/seam-registration</context-root>
        </web>
    </module>
    <module>
        <ejb>jboss-seam-registration.jar</ejb>
    </module>
    <module>
        <java>jboss-seam.jar</java>
    </module>
    <module>
        <java>el-api.jar</java>
    </module>
    <module>
        <java>el-ri.jar</java>
    </module>

</application>
```

This deployment descriptor links modules in the enterprise archive and binds the web application to the context root `/seam-registration`.

We've now seen *all* the files in the entire application!

1.2.2. How it works

When the form is submitted, JSF asks Seam to resolve the variable named `user`. Since there is no value already bound to that name (in any Seam context), Seam instantiates the `user` component, and returns the resulting `User` entity bean instance to JSF after storing it in the Seam session context.

The form input values are now validated against the Hibernate Validator constraints specified on the `User` entity. If the constraints are violated, JSF redisplays the page. Otherwise, JSF binds the form input values to properties of the `User` entity bean.

Next, JSF asks Seam to resolve the variable named `register`. Seam finds the `RegisterAction` stateless session

bean in the stateless context and returns it. JSF invokes the `register()` action listener method.

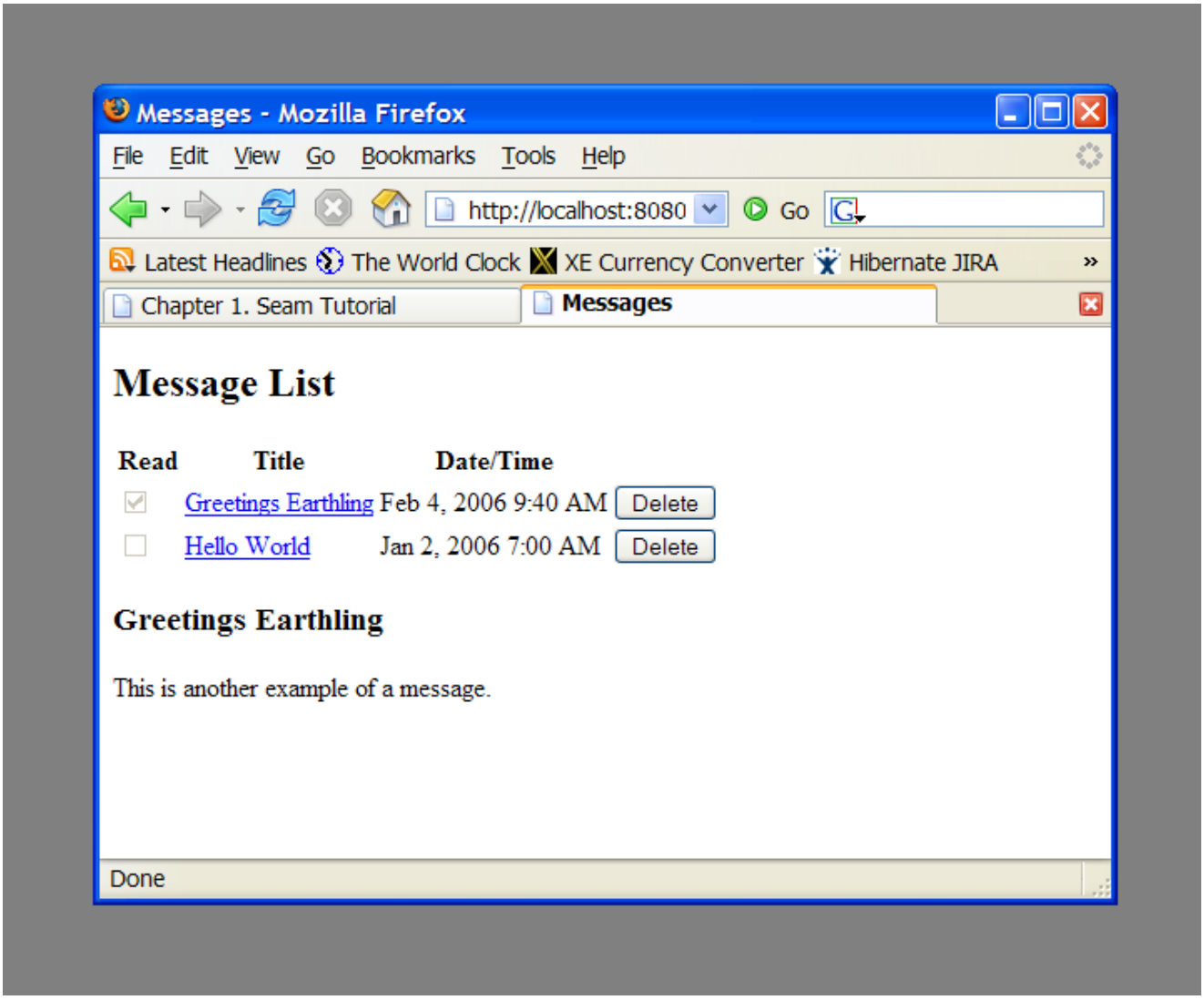
Seam intercepts the method call and injects the `User` entity from the Seam session context, before continuing the invocation.

The `register()` method checks if a user with the entered username already exists. If so, an error message is queued with the `FacesMessages` component, and a null outcome is returned, causing a page redisplay. The `FacesMessages` component interpolates the JSF expression embedded in the message string and adds a JSF `FacesMessage` to the view.

If no user with that username exists, the `"/registered.jsp"` outcome triggers a browser redirect to the `registered.jsp` page. When JSF comes to render the page, it asks Seam to resolve the variable named `user` and uses property values of the returned `User` entity from Seam's session scope.

1.3. Clickable lists in Seam: the messages example

Clickable lists of database search results are such an important part of any online application that Seam provides special functionality on top of JSF to make it easier to query data using EJB-QL or HQL and display it as a clickable list using a JSF `<h:dataTable>`. The messages example demonstrates this functionality.



1.3.1. Understanding the code

The message list example has one entity bean, `Message`, one session bean, `MessageListBean` and one JSP.

1.3.1.1. The entity bean: `Message.java`

The `Message` entity defines the title, text, date and time of a message, and a flag indicating whether the message has been read:

Example 1.10.

```
@Entity
@Name("message")
@Scope(EVENT)
public class Message implements Serializable
{
   private Long id;
   private String title;
   private String text;
   private boolean read;
   private Date datetime;

   @Id @GeneratedValue
   public Long getId() {
      return id;
   }
   public void setId(Long id) {
      this.id = id;
   }

   @NotNull @Length(max=100)
   public String getTitle() {
      return title;
   }
   public void setTitle(String title) {
      this.title = title;
   }

   @NotNull @Lob
   public String getText() {
      return text;
   }
   public void setText(String text) {
      this.text = text;
   }

   @NotNull
   public boolean isRead() {
      return read;
   }
   public void setRead(boolean read) {
      this.read = read;
   }

   @NotNull
   @Basic @Temporal(TemporalType.TIMESTAMP)
   public Date getDatetime() {
      return datetime;
   }
   public void setDatetime(Date datetime) {
      this.datetime = datetime;
   }

}
```

1.3.1.2. The stateful session bean: `MessageManagerBean.java`

Just like in the previous example, we have a session bean, `MessageManagerBean`, which defines the action

listener methods for the two buttons on our form. One of the buttons selects a message from the list, and displays that message. The other button deletes a message. So far, this is not so different to the previous example.

But `MessageManagerBean` is also responsible for fetching the list of messages the first time we navigate to the message list page. There are various ways the user could navigate to the page, and not all of them are preceded by a JSF action—the user might have bookmarked the page, for example. So the job of fetching the message list takes place in a Seam *factory method*, instead of in an action listener method.

We want to cache the list of messages in memory between server requests, so we will make this a stateful session bean.

Example 1.11.

```
@Stateful
@Scope(SESSION)
@Name("messageManager")
public class MessageManagerBean implements Serializable, MessageManager
{

   @DataModel                                                                    (1)
   private List<Message> messageList;

   @DataModelSelection                                                           (2)
   @Out(required=false)                                                          (3)
   private Message message;

   @PersistenceContext(type=EXTENDED)                                            (4)
   private EntityManager em;

   @Factory("messageList")                                                       (5)
   public void findMessages()
   {
      messageList = em.createQuery("from Message msg order by msg.datetime desc").getResultList();
   }

   public void select()                                                          (6)
   {
      message.setRead(true);
   }

   public void delete()                                                          (7)
   {
      messageList.remove(message);
      em.remove(message);
      message=null;
   }

   @Remove @Destroy                                                              (8)
   public void destroy() {}

}
```

(1) The `@DataModel` annotation exposes an attibute of type `java.util.List` to the JSF page as an instance of `javax.faces.model.DataModel`. This allows us to use the list in a JSF `<h:dataTable>` with clickable links for each row. In this case, the `DataModel` is made available in a session context variable named `messageList`.

(2) The `@DataModelSelection` annotation tells Seam to inject the `List` element that corresponded to the clicked link.

(3) The `@Out` annotation then exposes the selected value directly to the page. So ever time a row of the clickable list is selected, the `Message` is injected to the attribute of the stateful bean, and the subsequently *outjected* to the event context variable named `message`.

(4) This stateful bean has an EJB3 *extended persistence context*. The messages retrieved in the query remain

in the managed state as long as the bean exists, so any subsequent method calls to the stateful bean can update them without needing to make any explicit call to the `EntityManager`.

(5) The first time we navigate to the JSP page, there will be no value in the `messageList` context variable. The `@Factory` annotation tells Seam to create an instance of `MessageManagerBean` and invoke the `findMessages()` method to initialize the value. We call `findMessages()` a *factory method* for `messages`.

(6) The `select()` action listener method marks the selected `Message` as read, and updates it in the database.

(7) The `delete()` action listener method removes the selected `Message` from the database.

(8) All stateful session bean Seam components *must* have a method marked `@Remove @Destroy` to ensure that Seam will remove the stateful bean when the Seam context ends, and clean up any server-side state.

Note that this is a session-scoped Seam component. It is associated with the user login session, and all requests from a login session share the same instance of the component. (In Seam applications, we usually use session-scoped components sparingly.)

1.3.1.3. The session bean local interface: `MessageManager.java`

All session beans have a business interface, of course.

```
@Local
public interface MessageManager
{
   public void findMessages();
   public void select();
   public void delete();
   public void destroy();
}
```

From now on, we won't show local interfaces in our code examples.

Let's skip over `components.xml`, `persistence.xml`, `web.xml`, `ejb-jar.xml`, `faces-config.xml` and `application.xml` since they are much the same as the previous example, and go straight to the JSP.

1.3.1.4. The view: `messages.jsp`

The JSP page is a straightforward use of the JSF `<h:dataTable>` component. Again, nothing specific to Seam.

Example 1.12.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<html>
 <head>
  <title>Messages</title>
 </head>
 <body>
  <f:view>
   <h:form>
     <h2>Message List</h2>
     <h:outputText value="No messages to display" rendered="#{messageList.rowCount==0}"/>
     <h:dataTable var="msg" value="#{messageList}" rendered="#{messageList.rowCount>0}">
        <h:column>
           <f:facet name="header">
              <h:outputText value="Read"/>
           </f:facet>
           <h:selectBooleanCheckbox value="#{msg.read}" disabled="true"/>
        </h:column>
        <h:column>
           <f:facet name="header">
              <h:outputText value="Title"/>
           </f:facet>
```

```
            <h:commandLink value="#{msg.title}" action="#{messageManager.select}"/>
         </h:column>
         <h:column>
            <f:facet name="header">
               <h:outputText value="Date/Time"/>
            </f:facet>
            <h:outputText value="#{msg.datetime}">
               <f:convertDateTime type="both" dateStyle="medium" timeStyle="short"/>
            </h:outputText>
         </h:column>
         <h:column>
            <h:commandButton value="Delete" action="#{messageManager.delete}"/>
         </h:column>
      </h:dataTable>
      <h3><h:outputText value="#{message.title}"/></h3>
      <div><h:outputText value="#{message.text}"/></div>
   </h:form>
  </f:view>
 </body>
</html>
```

1.3.2. How it works

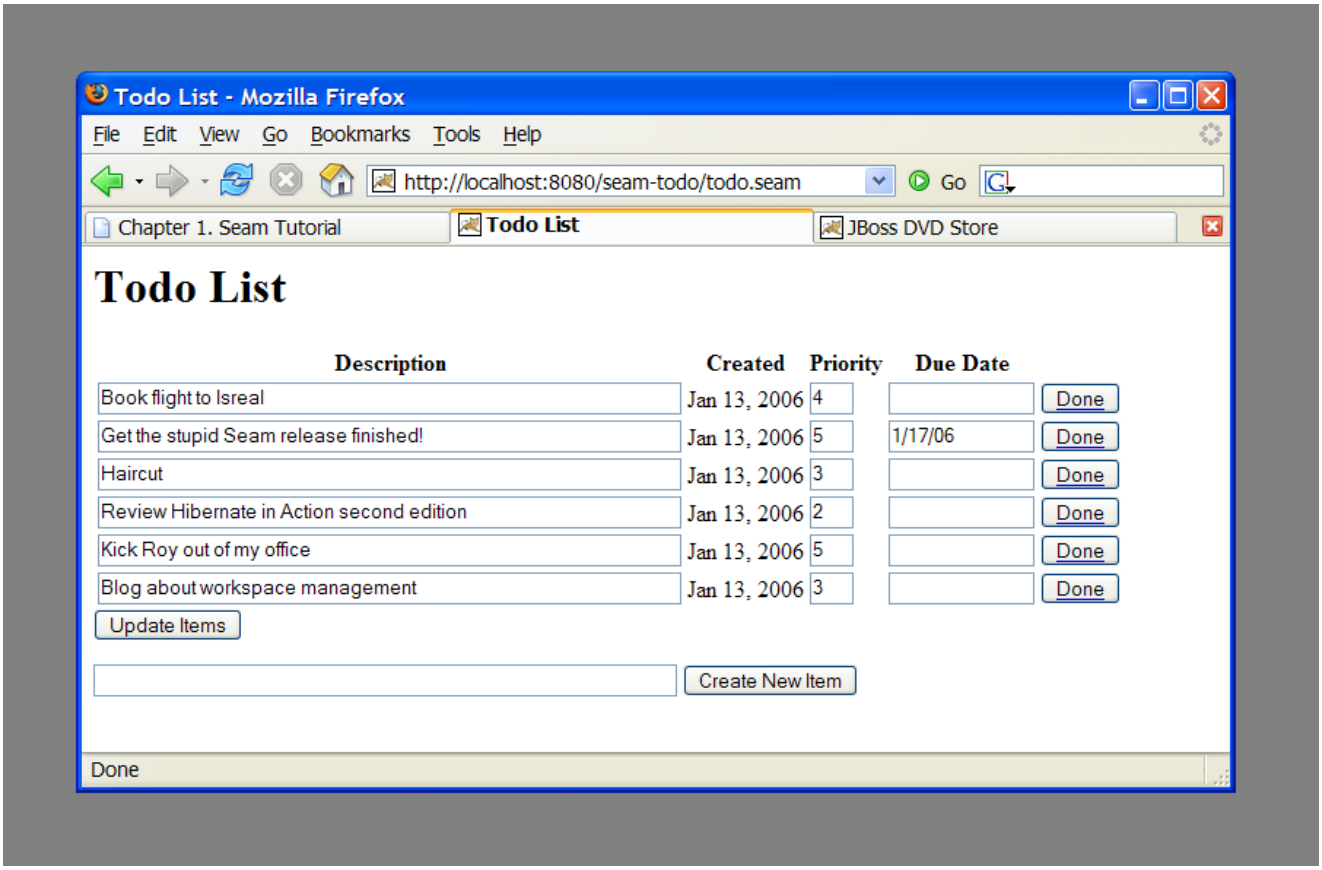
The first time we navigate to the `messages.jsp` page, whether by a JSF postback (faces request) or a direct browser GET request (non-faces request), the page will try to resolve the `messageList` context variable. Since this context variable is not initialized, Seam will call the factory method `findMessages()`, which performs a query against the database and results in a `DataModel` being outjected. This `DataModel` provides the row data needed for rendering the `<h:dataTable>`.

When the user clicks the `<h:commandLink>`, JSF calls the `select()` action listener. Seam intercepts this call and injects the selected row data into the `message` attribute of the `messageManager` component. The action listener fires, marking the selected `Message` as read. At the end of the call, Seam outjects the selected `Message` to the context variable named `message`. Next, the EJB container commits the transaction, and the change to the `Message` is flushed to the database. Finally, the page is re-rendered, redisplaying the message list, and displaying the selected message below it.

If the user clicks the `<h:commandButton>`, JSF calls the `delete()` action listener. Seam intercepts this call and injects the selected row data into the `message` attribute of the `messageList` component. The action listener fires, removing the selected `Message` from the list, and also calling `remove()` on the `EntityManager`. At the end of the call, Seam refreshes the `messageList` context variable and clears the context variable named `message`. The EJB container commits the transaction, and deletes the `Message` from the database. Finally, the page is re-rendered, redisplaying the message list.

1.4. Seam and jBPM: the todo list example

jBPM provides sophisticated functionality for workflow and task management. To get a small taste of how jBPM integrates with Seam, we'll show you a simple "todo list" application. Since managing lists of tasks is such core functionality for jBPM, there is hardly any Java code at all in this example.



1.4.1. Understanding the code

The center of this example is the jBPM process definition. There are also two JSPs and two trivial JavaBeans (There was no reason to use session beans, since they do not access the database, or have any other transactional behavior). Let's start with the process definition:

Example 1.13.

```
<process-definition name="todo">

   <start-state name="start">                                          (1)
      <transition to="todo"/>
   </start-state>

   <task-node name="todo">                                             (2)
      <task name="todo" description="#{todoList.description}">         (3)
         <assignment actor-id="#{actor.id}"/>                          (4)
      </task>
      <transition to="done"/>
   </task-node>

   <end-state name="done"/>                                            (5)

</process-definition>
```

(1) The `<start-state>` node represents the logical start of the process. When the process starts, it immediately transitions to the `todo` node.

(2) The `<task-node>` node represents a *wait state*, where business process execution pauses, waiting for one or more tasks to be performed.

(3) The `<task>` element defines a task to be performed by a user. Since there is only one task defined on this node, when it is complete, execution resumes, and we transition to the end state. The task gets its descrip-

tion from a Seam component named `todoList` (one of the JavaBeans).

(4) Tasks need to be assigned to a user or group of users when they are created. In this case, the task is assigned to the current user, which we get from a built-in Seam component named `actor`. Any Seam component may be used to perform task assignment.

(5) The `<end-state>` node defines the logical end of the business process. When execution reaches this node, the process instance is destroyed.

If we view this process definition using the process definition editor provided by JBossIDE, this is what it looks like:

This document defines our *business process* as a graph of nodes. This is the most trivial possible business process: there is one *task* to be performed, and when that task is complete, the business process ends.

The first JavaBean handles the login screen `login.jsp`. Its job is just to initialize the jBPM actor id using the `actor` component. (In a real application, it would also need to authenticate the user.)

Example 1.14.

```
@Name("login")
public class Login {

   @In
   private Actor actor;

   private String user;

   public String getUser() {
      return user;
   }

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

```
   public String login()
   {
      actor.setId(user);
      return "/todo.jsp";
   }
}
```

Here we see the use of `@In` to inject the built-in `Actor` component.

The JSP itself is trivial:

Example 1.15.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<html>
<head>
<title>Login</title>
</head>
<body>
<h1>Login</h1>
<f:view>
    <h:form>
      <div>
        <h:inputText value="#{login.user}"/>
        <h:commandButton value="Login" action="#{login.login}"/>
      </div>
    </h:form>
</f:view>
</body>
</html>
```

The second JavaBean is responsible for starting business process instances, and ending tasks.

Example 1.16.

```
@Name("todoList")
public class TodoList {

   private String description;

   public String getDescription()                                          (1)
   {
      return description;
   }

   public void setDescription(String description) {
      this.description = description;
   }

   @CreateProcess(definition="todo")                                       (2)
   public void createTodo() {}

   @StartTask @EndTask                                                     (3)
   public void done() {}

}
```

(1) The description property accepts user input form the JSP page, and exposes it to the process definition, allowing the task description to be set.

(2) The Seam `@CreateProcess` annotation creates a new jBPM process instance for the named process definition.

(3) The Seam `@StartTask` annotation starts work on a task. The `@EndTask` ends the task, and allows the business process execution to resume.

In a more realistic example, `@StartTask` and `@EndTask` would not appear on the same method, because there is usually work to be done using the application in order to complete the task.

Finally, the meat of the application is in `todo.jsp`:

Example 1.17.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://jboss.com/products/seam/taglib" prefix="s" %>
<html>
<head>
<title>Todo List</title>
</head>
<body>
<h1>Todo List</h1>
<f:view>
   <h:form id="list">
      <div>
         <h:outputText value="There are no todo items." rendered="#{empty taskInstanceList}"/>
         <h:dataTable value="#{taskInstanceList}" var="task" rendered="#{not empty taskInstanceList}">
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Description"/>
                </f:facet>
                <h:inputText value="#{task.description}"/>
            </h:column>
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Created"/>
                </f:facet>
                <h:outputText value="#{task.taskMgmtInstance.processInstance.start}">
                    <f:convertDateTime type="date"/>
                </h:outputText>
            </h:column>
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Priority"/>
                </f:facet>
                <h:inputText value="#{task.priority}" style="width: 30"/>
            </h:column>
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Due Date"/>
                </f:facet>
                <h:inputText value="#{task.dueDate}" style="width: 100">
                    <f:convertDateTime type="date" dateStyle="short"/>
                </h:inputText>
            </h:column>
            <h:column>
                <s:button value="Done" action="#{todoList.done}" taskInstance="#{task}"/>
            </h:column>
         </h:dataTable>
      </div>
      <div>
      <h:messages/>
      </div>
      <div>
         <h:commandButton value="Update Items" action="update"/>
      </div>
   </h:form>
```

```
   <h:form id="new">
      <div>
         <h:inputText value="#{todoList.description}"/>
         <h:commandButton value="Create New Item" action="#{todoList.createTodo}"/>
      </div>
   </h:form>
</f:view>
</body>
</html>
```

Let's take this one piece at a time.

The page renders a list of tasks, which it gets from a built-in Seam component named `taskInstanceList`. The list is defined inside a JSF form.

```
<h:form id="list">
   <div>
      <h:outputText value="There are no todo items." rendered="#{empty taskInstanceList}"/>
      <h:dataTable value="#{taskInstanceList}" var="task" rendered="#{not empty taskInstanceList}">
         ...
      </h:dataTable>
   </div>
</h:form>
```

Each element of the list is an instance of the jBPM class `TaskInstance`. The following code simply displays the interesting properties of each task in the list. For the description, priority and due date, we use input controls, to allow the user to update these values.

```
<h:column>
    <f:facet name="header">
       <h:outputText value="Description"/>
    </f:facet>
    <h:inputText value="#{task.description}"/>
</h:column>
<h:column>
    <f:facet name="header">
        <h:outputText value="Created"/>
    </f:facet>
    <h:outputText value="#{task.taskMgmtInstance.processInstance.start}">
        <f:convertDateTime type="date"/>
    </h:outputText>
</h:column>
<h:column>
    <f:facet name="header">
        <h:outputText value="Priority"/>
    </f:facet>
    <h:inputText value="#{task.priority}" style="width: 30"/>
</h:column>
<h:column>
    <f:facet name="header">
        <h:outputText value="Due Date"/>
    </f:facet>
    <h:inputText value="#{task.dueDate}" style="width: 100">
        <f:convertDateTime type="date" dateStyle="short"/>
    </h:inputText>
</h:column>
```

This button ends the task by calling the action method annotated `@StartTask @EndTask`. It passes the task id to Seam as a request parameter:

```
<h:column>
    <s:button value="Done" action="#{todoList.done}" taskInstance="#{task}"/>
</h:column>
```

(Note that this is using a Seam `<s:button>` JSF control from the `seam-ui.jar` package.)

This button is used to update the properties of the tasks. When the form is submitted, Seam and jBPM will make any changes to the tasks persistent. There is no need for any action listener method:

```
<h:commandButton value="Update Items" action="update"/>
```

A second form on the page is used to create new items, by calling the action method annotated `@CreateProcess`.

```
<h:form id="new">
    <div>
        <h:inputText value="#{todoList.description}"/>
        <h:commandButton value="Create New Item" action="#{todoList.createTodo}"/>
    </div>
</h:form>
```

There are several other files needed for the example, but they are just standard jBPM and Seam configuration and not very interesting.

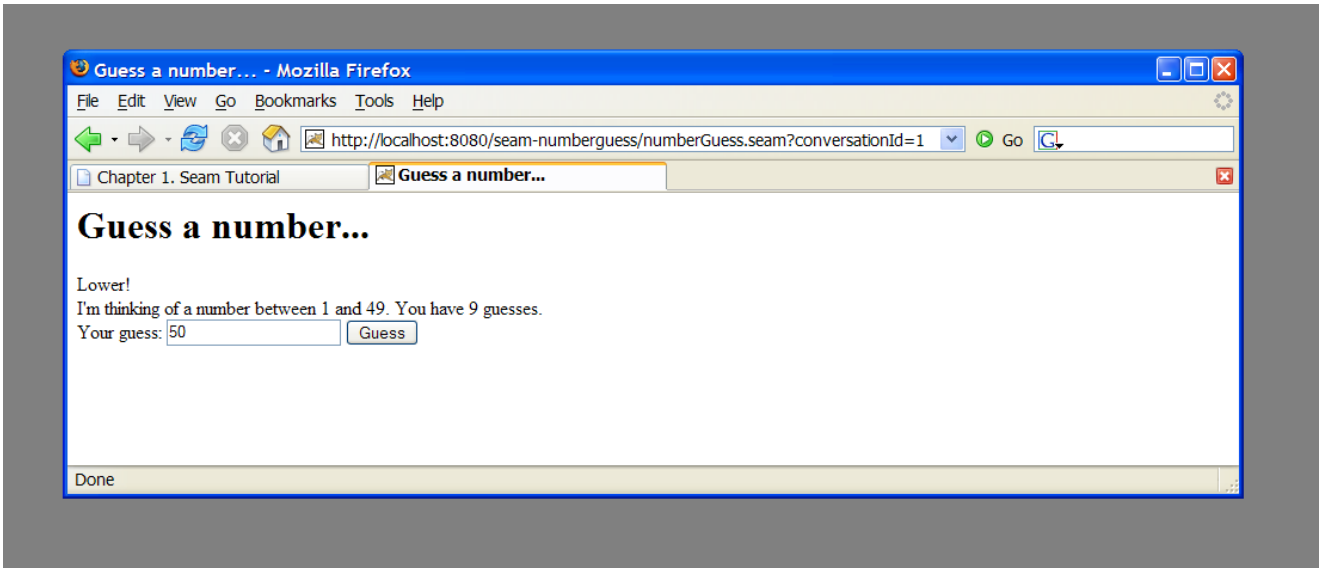
1.4.2. How it works

TODO

1.5. Seam pageflow: the numberguess example

For Seam applications with relatively freeform (ad hoc) navigation, JSF/Seam navigation rules are a perfectly good way to define the page flow. For applications with a more constrained style of navigation, especially for user interfaces which are more stateful, navigation rules make it difficult to really understand the flow of the system. To understand the flow, you need to piece it together from the view pages, the actions and the navigation rules.

Seam allows you to use a jPDL process definition to define pageflow. The simple number guessing example shows how this is done.



1.5.1. Understanding the code

The example is implemented using one JavaBean, three JSP pages and a jPDL pageflow definition. Let's begin with the pageflow:

Example 1.18.

```
<pageflow-definition name="numberGuess">

   <start-page name="displayGuess" view-id="/numberGuess.jsp">
      <redirect/>
      <transition name="guess" to="evaluateGuess">
          <action expression="#{numberGuess.guess}" />
      </transition>                                                          (1)
   </start-page>                                                             (2)
                                                                             (3)
   <decision name="evaluateGuess" expression="#{numberGuess.correctGuess}">
      <transition name="true" to="win"/>
      <transition name="false" to="evaluateRemainingGuesses"/>
   </decision>                                                               (4)

   <decision name="evaluateRemainingGuesses" expression="#{numberGuess.lastGuess}">
      <transition name="true" to="lose"/>
      <transition name="false" to="displayGuess"/>
   </decision>

   <page name="win" view-id="/win.jsp">
      <redirect/>
      <end-conversation />
   </page>

   <page name="lose" view-id="/lose.jsp">
      <redirect/>
      <end-conversation />
   </page>

</pageflow-definition>
```

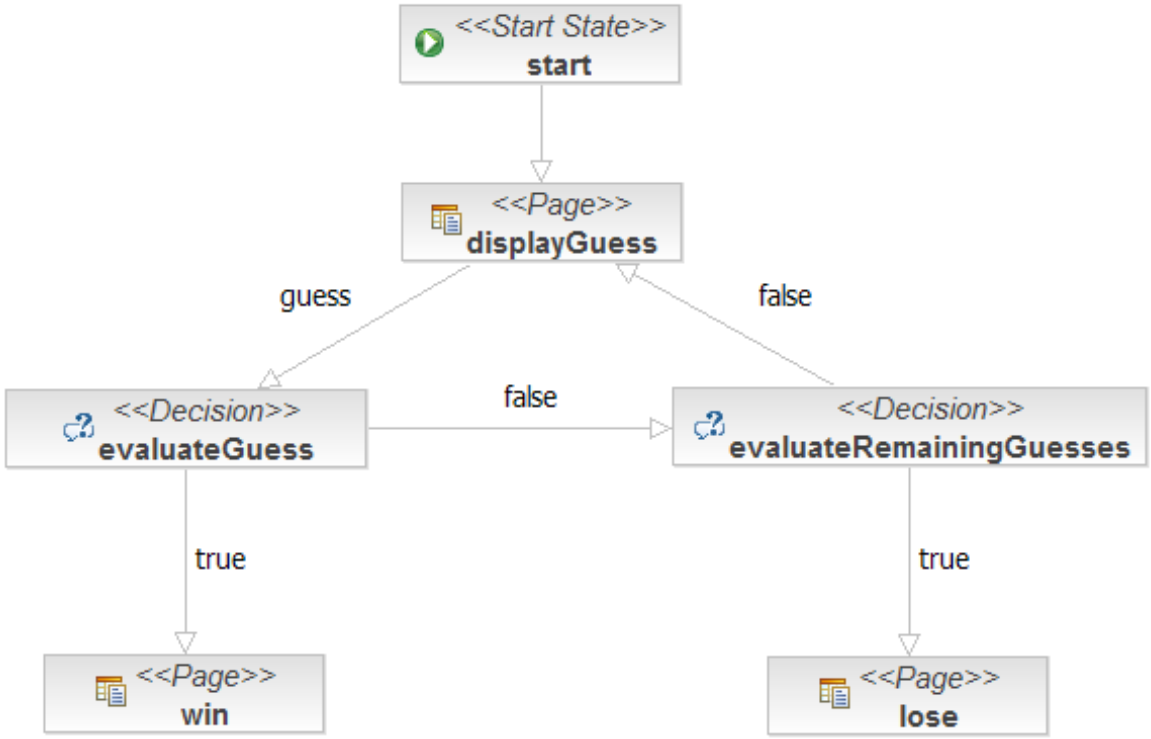
(1) The `<page>` element defines a wait state where the system displays a particular JSF view and waits for user input. The `view-id` is the same JSF view id used in plain JSF navigation rules. The `redirect` attribute tells Seam to use post-then-redirect when navigating to the page. (This results in friendly browser URLs.)

(2) The `<transition>` element names a JSF outcome. The transition is triggered when a JSF action results in that outcome. Execution will then proceed to the next node of the pageflow graph, after invocation of any jBPM transition actions.

(3) A transition `<action>` is just like a JSF action, except that it occurs when a jBPM transition occurs. The transition action can invoke any Seam component.

(4) A `<decision>` node branches the pageflow, and determines the next node to execute by evaluating a JSF EL expression.

Here is what the pageflow looks like in the JBossIDE pageflow editor:



Now that we have seen the pageflow, it is very, very easy to understand the rest of the application!

Here is the main page of the application, `numberGuess.jsp`:

Example 1.19.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<html>
<head>
<title>Guess a number...</title>
</head>
<body>
<h1>Guess a number...</h1>
<f:view>
    <h:form>
        <h:outputText value="Higher!" rendered="#{numberGuess.randomNumber>numberGuess.currentGuess}"
        <h:outputText value="Lower!" rendered="#{numberGuess.randomNumber<numberGuess.currentGuess}" /
        <br />
        I'm thinking of a number between <h:outputText value="#{numberGuess.smallest}" /> and
        <h:outputText value="#{numberGuess.biggest}" />. You have
        <h:outputText value="#{numberGuess.remainingGuesses}" /> guesses.
        <br />
        Your guess:
        <h:inputText value="#{numberGuess.currentGuess}" id="guess" required="true">
            <f:validateLongRange
                maximum="#{numberGuess.biggest}"
                minimum="#{numberGuess.smallest}"/>
        </h:inputText>
        <h:commandButton type="submit" value="Guess" action="guess" />
        <br/>
        <h:message for="guess" style="color: red"/>
    </h:form>
</f:view>
</body>
</html>
```

Notice how the command button names the `guess` transition instead of calling an action directly.

The `win.jsp` page is predictable:

Example 1.20.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<html>
<head>
<title>You won!</title>
</head>
<body>
<h1>You won!</h1>
<f:view>
    Yes, the answer was <h:outputText value="#{numberGuess.currentGuess}" />.
    It took you <h:outputText value="#{numberGuess.guessCount}" /> guesses.
    Would you like to <a href="numberGuess.seam">play again</a>?
  </f:view>
</body>
</html>
```

As is `lose.jsp` (which I can't be bothered copy/pasting). Finally, the JavaBean Seam component:

Example 1.21.

```
@Name("numberGuess")
@Scope(ScopeType.CONVERSATION)
public class NumberGuess {

   private int randomNumber;
   private Integer currentGuess;
   private int biggest;
   private int smallest;
   private int guessCount;
   private int maxGuesses;

   @Create                                                                    (1)
   @Begin(pageflow="numberGuess")                                             (2)
   public void begin()
   {
      randomNumber = new Random().nextInt(100);
      guessCount = 0;
      biggest = 100;
      smallest = 1;
   }

   public void setCurrentGuess(Integer guess)
   {
      this.currentGuess = guess;
   }

   public Integer getCurrentGuess()
   {
      return currentGuess;
   }

   public void guess()
   {
      if (currentGuess>randomNumber)
      {
         biggest = currentGuess - 1;
      }
      if (currentGuess<randomNumber)
```

```
      {
         smallest = currentGuess + 1;
      }
      guessCount ++;
   }

   public boolean isCorrectGuess()
   {
      return currentGuess==randomNumber;
   }

   public int getBiggest()
   {
      return biggest;
   }

   public int getSmallest()
   {
      return smallest;
   }

   public int getGuessCount()
   {
      return guessCount;
   }

   public boolean isLastGuess()
   {
      return guessCount==maxGuesses;
   }

   public int getRemainingGuesses() {
      return maxGuesses-guessCount;
   }

   public void setMaxGuesses(int maxGuesses) {
      this.maxGuesses = maxGuesses;
   }

   public int getMaxGuesses() {
      return maxGuesses;
   }

   public int getRandomNumber() {
      return randomNumber;
   }
}
```

(1) The first time a JSP page asks for a `numberGuess` component, Seam will create a new one for it, and the `@Create` method will be invoked, allowing the component to initialize itself.

(2) The `@Begin` annotation starts a Seam *conversation* (much more about that later), and specifies the page-flow definition to use for the conversation's page flow.

As you can see, this Seam component is pure business logic! It doesn't need to know anything at all about the user interaction flow. This makes the component potentially more reuseable.

1.5.2. How it works

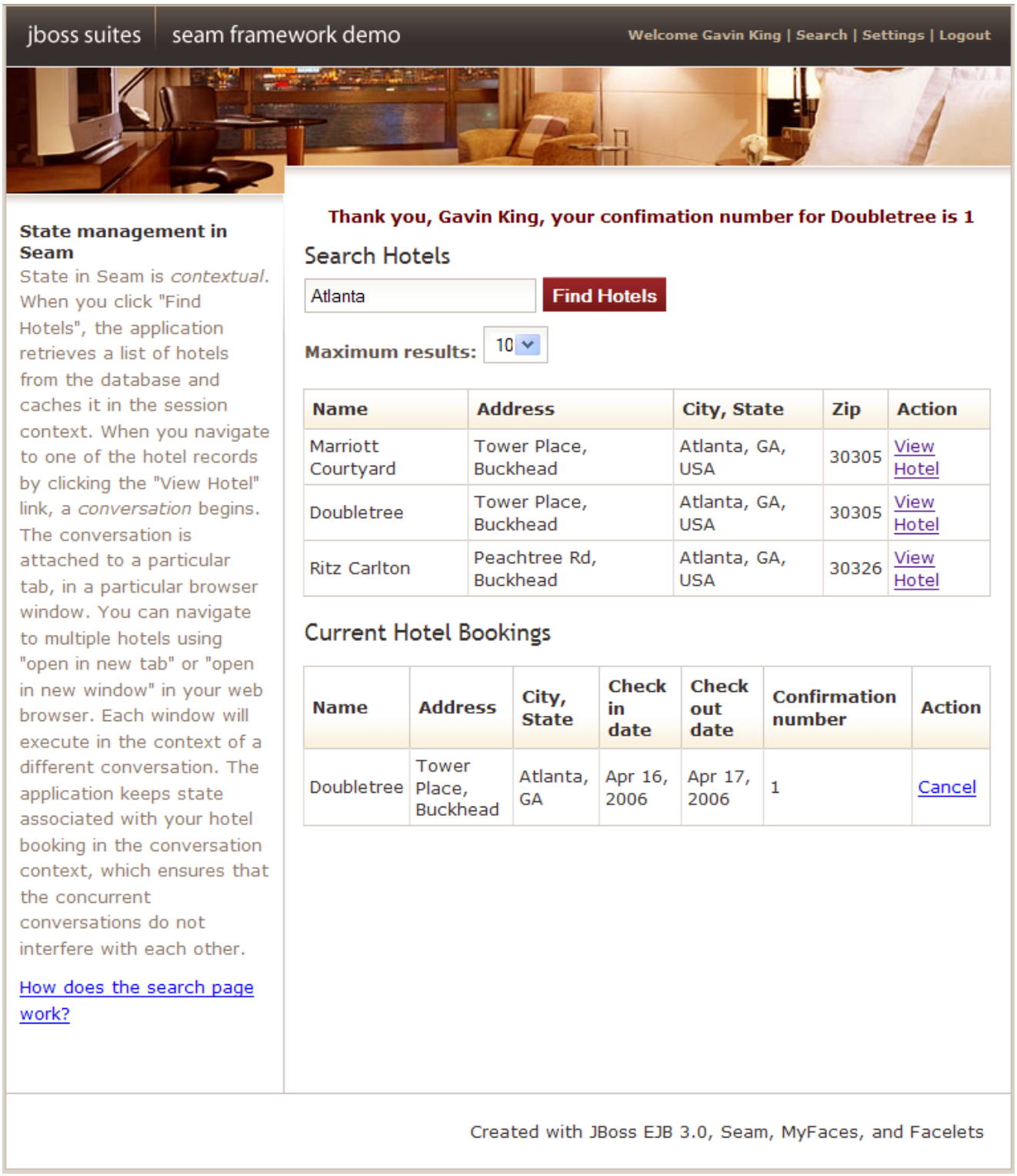
TODO

1.6. A complete Seam application: the Hotel Booking example

1.6.1. Introduction

The booking application is a complete hotel room reservation system incorporating the following features:

- User registration
- Login
- Logout
- Set password
- Hotel search
- Hotel selection
- Room reservation
- Reservation confirmation
- Existing reservation list



The booking application uses JSF, EJB 3.0 and Seam, together with Facelets for the view. There is also a port of this application to JSF, Facelets, Seam, JavaBeans and Hibernate3.

One of the things you'll notice if you play with this application for long enough is that it is extremely *robust*. You can play with back buttons and browser refresh and opening multiple windows and entering nonsensical data as much as you like and you will find it very difficult to make the application crash. You might think that we spent weeks testing and fixing bugs to achive this. Actually, this is not the case. Seam was designed to make it very straightforward to build robust web applications and a lot of robustness that you are probably used to having to code yourself comes naturally and automatically with Seam.

As you browse the sourcecode of the example application, and learn how the application works, observe how the declarative state management and integrated validation has been used to achieve this robustness.

1.6.2. Overview of the booking example

The project structure is identical to the previous one, to install and deploy this application, please refer to Section 1.1, “Try the examples”. Once you've successfully started the application, you can access it by pointing your browser to `http://localhost:8080/seam-booking/`

Just ten classes (plus six session beans local interfaces and 1 annotation interface) where used to implement this application. Six session bean action listeners contain all the business logic for the listed features.

- `BookingListAction` retrieves existing bookings for the currently logged in user.
- `ChangePasswordAction` updates the password of the currently logged in user.
- `HotelBookingAction` implements the core functionality of the application: hotel room searching, selection, booking and booking confirmation. This functionality is implemented as a *conversation*, so this is the most interesting class in the application.
- `LoginAction` validates the login details and retrieves the logged in user.
- `LogoutAction` ends the login session.
- `RegisterAction` registers a new system user.

Three entity beans implement the application's persistent domain model.

- `Hotel` is an entity bean that represent a hotel
- `Booking` is an entity bean that represents an existing booking
- `User` is an entity bean to represents a user who can make hotel bookings

Finally, the `LoggedIn` annotation and the `LoggedInInterceptor` are used to protect actions that require a logged in user.

1.6.3. Understanding Seam conversations

We encourage you browse the sourcecode at your pleasure. In this tutorial we'll concentrate upon one particular piece of functionality: hotel search, selection, booking and confirmation. From the point of view of the user, everything from selecting a hotel to confirming a booking is one continuous unit of work, a *conversation*. Searching, however, is *not* part of the conversation. The user can select multiple hotels from the same search results page, in different browser tabs.

Most web application architectures have no first class construct to represent a conversation. This causes enormous problems managing state associated with the conversation. Usually, Java web applications use a combination of two techniques: first, some state is thrown into the `HttpSession`; second, persistable state is flushed to the database after every request, and reconstructed from the database at the beginning of each new request.

Since the database is the least scalable tier, this often results in an utterly unacceptable lack of scalability. Added latency is also a problem, due to the extra traffic to and from the database on every request. To reduce this redundant traffic, Java applications often introduce a data (second-level) cache that keeps commonly accessed data between requests. This cache is necessarily inefficient, because invalidation is based upon an LRU policy instead of being based upon when the user has finished working with the data. Furthermore, because the cache is shared between many concurrent transactions, we've introduced a whole raft of problem's associated with

keeping the cached state consistent with the database.

Now consider the state held in the `HttpSession`. By very careful programming, we might be able to control the size of the session data. This is a lot more difficult than it sounds, since web browsers permit ad hoc non-linear navigation. But suppose we suddenly discover a system requirement that says that a user is allowed to have *mutiple concurrent conversations*, halfway through the development of the system (this has happened to me). Developing mechanisms to isolate session state associated with different concurrent conversations, and incorporating failsafes to ensure that conversation state is destroyed when the user aborts one of the conversations by closing a browser window or tab is not for the faint hearted (I've implemented this stuff twice so far, once for a client application, once for Seam, but I'm famously psychotic).

Now there is a better way.

Seam introduces the *conversation context* as a first class construct. You can safely keep conversational state in this context, and be assured that it will have a well-defined lifecycle. Even better, you won't need to be continually pushing data back and forth between the application server and the database, since the conversation context is a natural cache of data that the user is currently working with.

Usually, the components we keep in the conversation context are stateful session beans. (We can also keep entity beans and JavaBeans in the conversation context.) There is an ancient canard in the Java community that stateful session beans are a scalability killer. This may have been true in 1998 when WebFoobar 1.0 was released. It is no longer true today. Application servers like JBoss 4.0 have extremely sophisticated mechanisms for stateful session bean state replication. (For example, the JBoss EJB3 container performs fine-grained replication, replicating only those bean attribute values which actually changed.) Note that all the traditional technical arguments for why stateful beans are inefficient apply equally to the `HttpSession`, so the practice of shifting state from business tier stateful session bean components to the web session to try and improve performance is unbelievably misguided. It is certainly possible to write unscalable applications using stateful session beans, by using stateful beans incorrectly, or by using them for the wrong thing. But that doesn't mean you should *never* use them. Anyway, Seam guides you toward a safe usage model. Welcome to 2005.

OK, I'll stop ranting now, and get back to the tutorial.

The booking example application shows how stateful components with different scopes can collaborate together to achieve complex behaviors. The main page of the booking application allows the user to search for hotels. The search results are kept in the Seam session scope. When the user navigates to one of these hotels, a conversation begins, and a conversation scoped component calls back to the session scoped component to retrieve the selected hotel.

The booking example also demonstrates the use of Ajax4JSF to implement rich client behavior without the use of handwritten JavaScript.

The search functionality is implemented using a session-scope stateful session bean, similar to the one we saw in the message list example above.

Example 1.22.

```
@Stateful                                                                    (1)
@Name("hotelSearch")
@Scope(ScopeType.SESSION)
@LoggedIn                                                                    (2)
@Synchronized
public class HotelSearchingAction implements HotelSearching
{

   @PersistenceContext
```

```
   private EntityManager em;

   private String searchString;
   private int pageSize = 10;
   private int page;

   @DataModel                                                                    (3)
   private List<Hotel> hotels;

   public String find()
   {
      page = 0;
      queryHotels();
      return "main";
   }

   public String nextPage()
   {
      page++;
      queryHotels();
      return "main";
   }

   private void queryHotels()
   {
      String searchPattern = searchString==null ? "%" : '%' + searchString.toLowerCase().replace('*',
      hotels = em.createQuery("select h from Hotel h where lower(h.name) like :search or lower(h.city
            .setParameter("search", searchPattern)
            .setMaxResults(pageSize)
            .setFirstResult( page * pageSize )
            .getResultList();
   }

   public boolean isNextPageAvailable()
   {
      return hotels!=null && hotels.size()==pageSize;
   }

   public int getPageSize() {
      return pageSize;
   }

   public void setPageSize(int pageSize) {
      this.pageSize = pageSize;
   }

   public String getSearchString()
   {
      return searchString;
   }

   public void setSearchString(String searchString)
   {
      this.searchString = searchString;
   }

   @Destroy @Remove                                                              (4)
   public void destroy() {}

}
```

(1) The EJB standard `@Stateful` annotation identifies this class as a stateful session bean. Stateful session beans are scoped to the conversation context by default.

(2) The `@LoggedIn` annotation applies a custom Seam interceptor to the component. This works because `@LoggedIn` is marked with an `@Interceptor` meta-annotation.

(3) The `@DataModel` annotation exposes a `List` as a JSF `ListDataModel`. This makes it easy to implement clickable lists for search screens. In this case, the list of hotels is exposed to the page as a `ListDataModel`

in the conversation variable named `hotels`.

(4) The EJB standard `@Remove` annotation specifies that a stateful session bean should be removed and its state destroyed after invocation of the annotated method. In Seam, all stateful session beans should define a method marked `@Destroy @Remove`. This is the EJB remove method that will be called when Seam destroys the session context. Actually, the `@Destroy` annotation is of more general usefulness, since it can be used for any kind of cleanup that should happen when any Seam context ends. If you don't have an `@Destroy @Remove` method, state will leak and you will suffer performance problems.

The main page of the application is a Facelets page. Let's look at the fragment which relates to searching for hotels:

Example 1.23.

```
<div class="section">
<h:form>

  <span class="errors">
    <h:messages globalOnly="true"/>
  </span>

  <h1>Search Hotels</h1>
  <fieldset>
     <h:inputText value="#{hotelSearch.searchString}" style="width: 165px;">
        <a:support event="onkeyup" actionListener="#{hotelSearch.find}"        (1)
                   reRender="searchResults" />
     </h:inputText>
      
     <a:commandButton value="Find Hotels" action="#{hotelSearch.find}"
                      styleClass="button" reRender="searchResults"/>
      
     <a:status>                                                                (2)
        <f:facet name="start">
           <h:graphicImage value="/img/spinner.gif"/>
        </f:facet>
     </a:status>
     <br/>
     <h:outputLabel for="pageSize">Maximum results:</h:outputLabel> 
     <h:selectOneMenu value="#{hotelSearch.pageSize}" id="pageSize">
        <f:selectItem itemLabel="5" itemValue="5"/>
        <f:selectItem itemLabel="10" itemValue="10"/>
        <f:selectItem itemLabel="20" itemValue="20"/>
     </h:selectOneMenu>
  </fieldset>

</h:form>
</div>

<a:outputPanel id="searchResults">                                            (3)
  <div class="section">
  <h:outputText value="No Hotels Found"
                rendered="#{hotels != null and hotels.rowCount==0}"/>
  <h:dataTable value="#{hotels}" var="hot" rendered="#{hotels.rowCount>0}">
    <h:column>
      <f:facet name="header">Name</f:facet>
      #{hot.name}
    </h:column>
    <h:column>
      <f:facet name="header">Address</f:facet>
      #{hot.address}
    </h:column>
    <h:column>
      <f:facet name="header">City, State</f:facet>
      #{hot.city}, #{hot.state}, #{hot.country}
    </h:column>
    <h:column>
```

```
      <f:facet name="header">Zip</f:facet>
      #{hot.zip}
    </h:column>
    <h:column>
      <f:facet name="header">Action</f:facet>
      <s:link value="View Hotel" action="#{hotelBooking.selectHotel(hot)}"/>   (4)
    </h:column>
  </h:dataTable>
  <s:link value="More results" action="#{hotelSearch.nextPage}"
          rendered="#{hotelSearch.nextPageAvailable}"/>
  </div>
</a:outputPanel>
```

(1) The Ajax4JSF `<a:support>` tag allows a JSF action event listener to be called by asynchronous `XMLHttpRequest` when a JavaScript event like `onkeyup` occurs. Even better, the `reRender` attribute lets us render a fragment of the JSF page and perform a partial page update when the asynchronous response is received.

(2) The Ajax4JSF `<a:status>` tag lets us display a cheesy annimated image while we wait for asynchronous requests to return.

(3) The Ajax4JSF `<a:outputPanel>` tag defines a region of the page which can be re-rendered by an asynchronous request.

(4) The Seam `<s:link>` tag lets us attach a JSF action listener to an ordinary (non-JavaScript) HTML link. The advantage of this over the standard JSF `<h:commandLink>` is that it preserves the operation of "open in new window" and "open in new tab". Also notice that we use a method binding with a parameter: `#{hotelBooking.selectHotel(hot)}`. This is not possible in the standard Unified EL, but Seam provides an extension to the EL that lets you use parameters on any method binding expression.

This page displays the search results dynamically as we type, and lets us choose a hotel and pass it to the `selectHotel()` method of the `HotelBookingAction`, which is where the *really* interesting stuff is going to happen.

Now lets see how the booking example application uses a conversation-scoped stateful session bean to achieve a natural cache of persistent data related to the conversation. The following code example is pretty long. But if you think of it as a list of scripted actions that implement the various steps of the conversation, it's understandable. Read the class from top to bottom, as if it were a story.

Example 1.24.

```
@Stateful
@Name("hotelBooking")
@LoggedIn
public class HotelBookingAction implements HotelBooking
{

   @PersistenceContext(type=EXTENDED)                                    (1)
   private EntityManager em;

   @In                                                                   (2)
   private User user;

   @In(required=false) @Out
   private Hotel hotel;

   @In(required=false)
   @Out(required=false)
   private Booking booking;

   @In
   private FacesMessages facesMessages;
```

```
   @In
   private Events events;

   @Logger
   private Log log;

   @Begin                                                                    (3)
   public String selectHotel(Hotel selectedHotel)
   {
      hotel = em.merge(selectedHotel);
      return "hotel";
   }

   public String bookHotel()
   {
      booking = new Booking(hotel, user);
      Calendar calendar = Calendar.getInstance();
      booking.setCheckinDate( calendar.getTime() );
      calendar.add(Calendar.DAY_OF_MONTH, 1);
      booking.setCheckoutDate( calendar.getTime() );

      return "book";
   }

   public String setBookingDetails()
   {
      if (booking==null || hotel==null) return "main";
      if ( !booking.getCheckinDate().before( booking.getCheckoutDate() ) )
      {
         facesMessages.add("Check out date must be later than check in date");
         return null;
      }
      else
      {
         return "confirm";
      }
   }

   @End                                                                      (4)
   public String confirm()
   {
      if (booking==null || hotel==null) return "main";
      em.persist(booking);
      facesMessages.add("Thank you, #{user.name}, your confimation number for #{hotel.name} is #{book
      log.info("New booking: #{booking.id} for #{user.username}");
      events.raiseEvent("bookingConfirmed");
      return "confirmed";
   }

   @End
   public String cancel()
   {
      return "main";
   }

   @Destroy @Remove                                                          (5)
   public void destroy() {}

}
```

(1) This bean uses an EJB3 *extended persistence context*, so that any entity instances remain managed for the whole lifecycle of the stateful session bean.

(2) The `@Out` annotation declares that an attribute value is *outjected* to a context variable after method invocations. In this case, the context variable named `hotel` will be set to the value of the `hotel` instance variable after every action listener invocation completes.

(3) The `@Begin` annotation specifies that the annotated method begins a *long-running conversation*, so the current conversation context will not be destroyed at the end of the request. Instead, it will be reassociated

with every request from the current window, and destroyed either by timeout due to conversation inactivity or invocation of a matching `@End` method.

(4) The `@End` annotation specifies that the annotated method ends the current long-running conversation, so the current conversation context will be destroyed at the end of the request.

(5) This EJB remove method will be called when Seam destroys the conversation context. Don't ever forget to define this method!

`HotelBookingAction` contains all the action listener methods that implement selection, booking and booking confirmation, and holds state related to this work in its instance variables. We think you'll agree that this code is much cleaner and simpler than getting and setting `HttpSession` attributes.

Even better, a user can have multiple isolated conversations per login session. Try it! Log in, run a search, and navigate to different hotel pages in multiple browser tabs. You'll be able to work on creating two different hotel reservations at the same time. If you leave any one conversation inactive for long enough, Seam will eventually time out that conversation and destroy its state. If, after ending a conversation, you backbutton to a page of that conversation and try to perform an action, Seam will detect that the conversation was already ended, and redirect you to the search page.

1.6.4. The Seam UI control library

If you check inside the WAR file for the booking application, you'll find `seam-ui.jar` in the `WEB-INF/lib` directory. This package contains a number of JSF custom controls that integrate with Seam. The booking application uses the `<s:link>` control for navigation from the search screen to the hotel page:

```
<s:link value="View Hotel" action="#{hotelBooking.selectHotel}"/>
```

The use of `<s:link>` here allows us to attach an action listener to a HTML link without breaking the browser's "open in new window" feature. The standard JSF `<h:commandLink>` does not work with "open in new window". We'll see later that `<s:link>` also offers a number of other useful features, including conversation propagation rules.

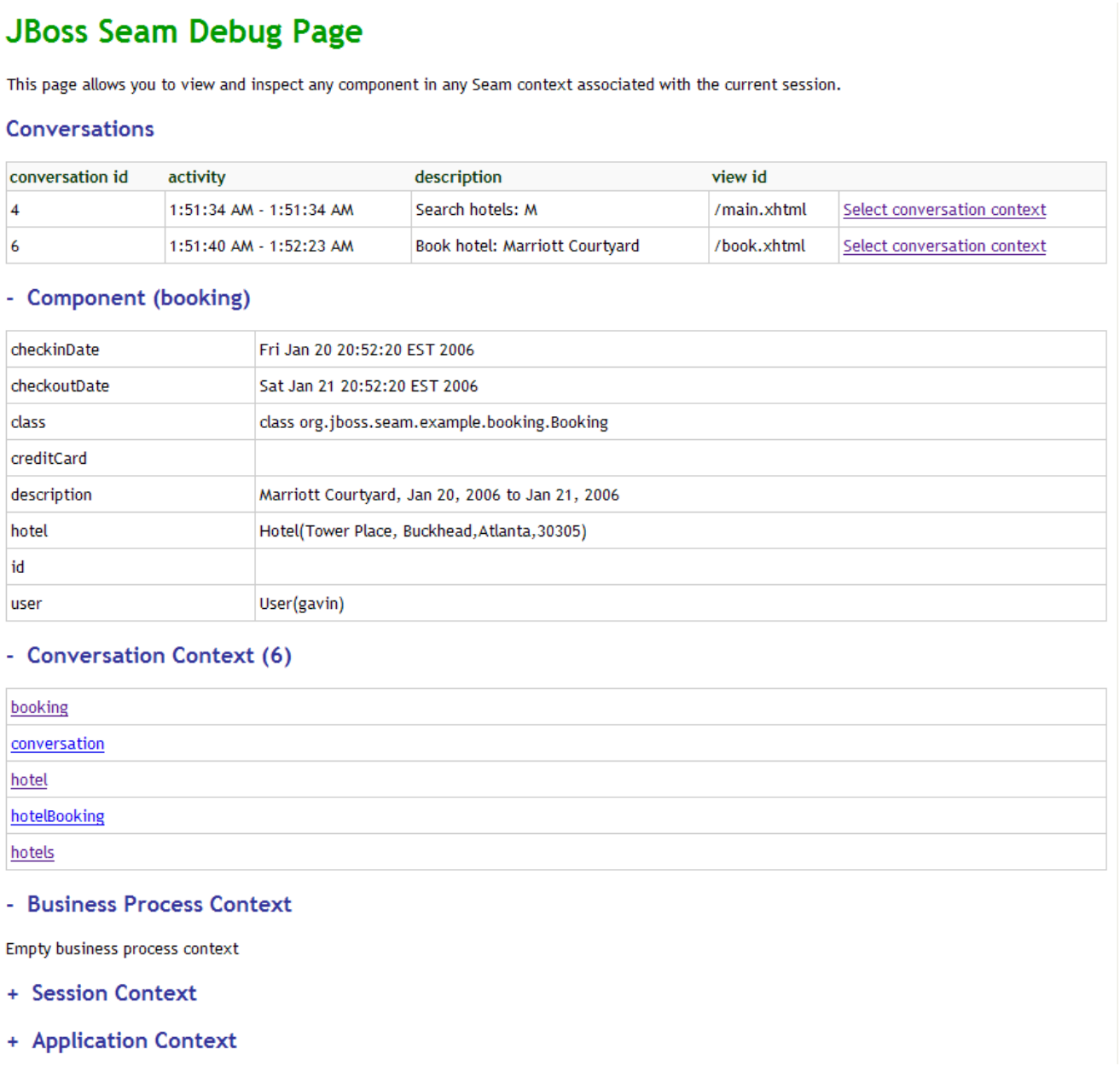
The booking application uses some other Seam and Ajax4JSF controls, especially on the `/book.xhtml` page. We won't get into the details of those controls here, but if you want to understand this code, please refer to the chapter covering Seam's functionality for JSF form validation.

1.6.5. The Seam Debug Page

The WAR also includes `seam-debug.jar`. If this jar is deployed in `WEB-INF/lib`, along with the Facelets, and if you set the following Seam property in `web.xml` or `seam.properties`:

```
<context-param>
    <param-name>org.jboss.seam.core.init.debug</param-name>
    <param-value>true</param-value>
</context-param>
```

Then the Seam debug page will be available. This page lets you browse and inspect the Seam components in any of the Seam contexts associated with your current login session. Just point your browser at `http://localhost:8080/seam-booking/debug.seam`.



JBoss Seam Debug Page

This page allows you to view and inspect any component in any Seam context associated with the current session.

Conversations

conversation id	activity	description	view id	
4	1:51:34 AM - 1:51:34 AM	Search hotels: M	/main.xhtml	Select conversation context
6	1:51:40 AM - 1:52:23 AM	Book hotel: Marriott Courtyard	/book.xhtml	Select conversation context

- Component (booking)

checkinDate	Fri Jan 20 20:52:20 EST 2006
checkoutDate	Sat Jan 21 20:52:20 EST 2006
class	class org.jboss.seam.example.booking.Booking
creditCard	
description	Marriott Courtyard, Jan 20, 2006 to Jan 21, 2006
hotel	Hotel(Tower Place, Buckhead,Atlanta,30305)
id	
user	User(gavin)

- Conversation Context (6)

booking
conversation
hotel
hotelBooking
hotels

- Business Process Context

Empty business process context

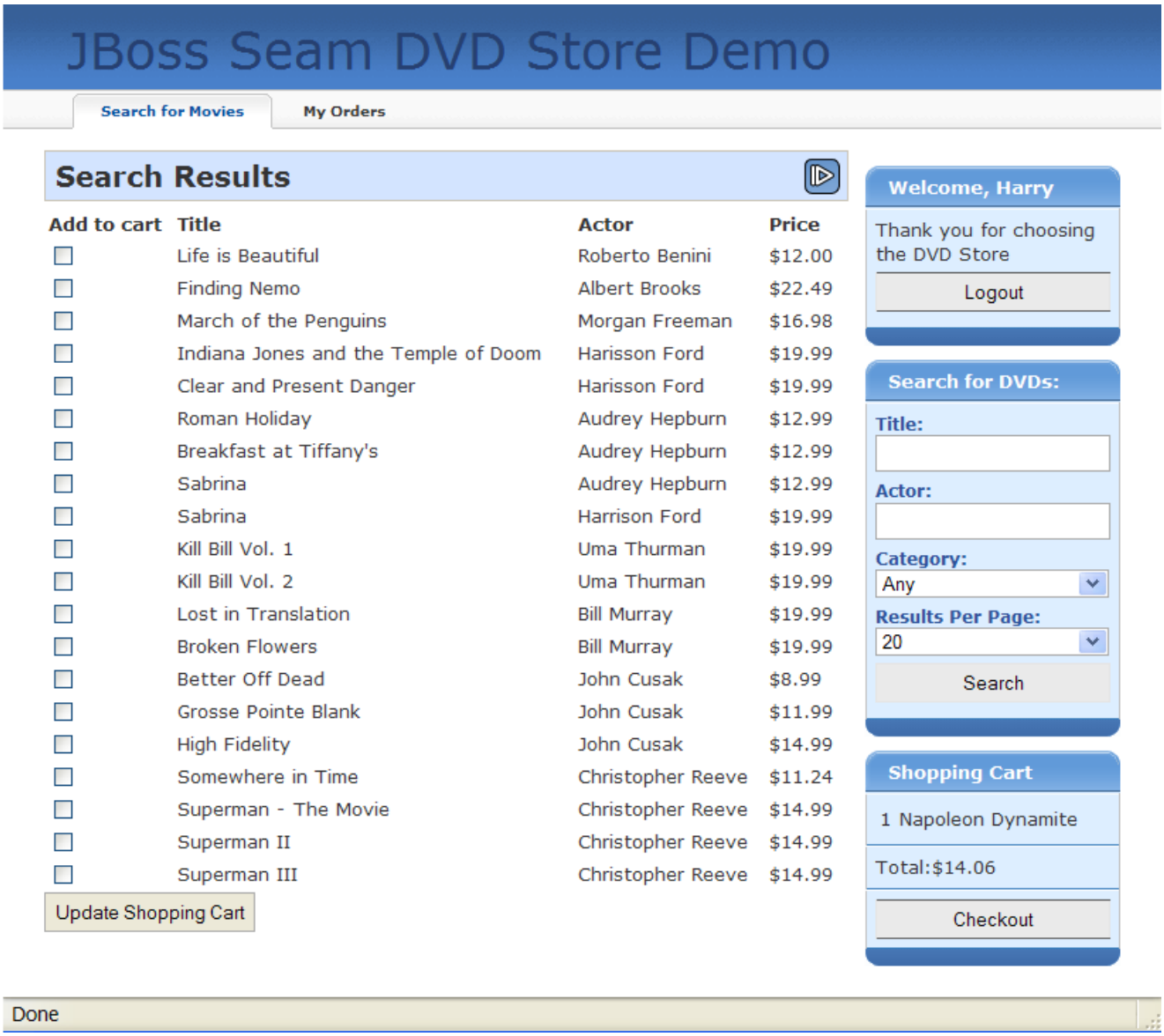
+ Session Context

+ Application Context

1.7. A complete application featuring Seam and jBPM: the DVD Store example

The DVD Store demo application shows the practical usage of jBPM for both task management and pageflow.

The user screens take advantage of a jPDL pageflow to implement searching and shopping cart functionality.



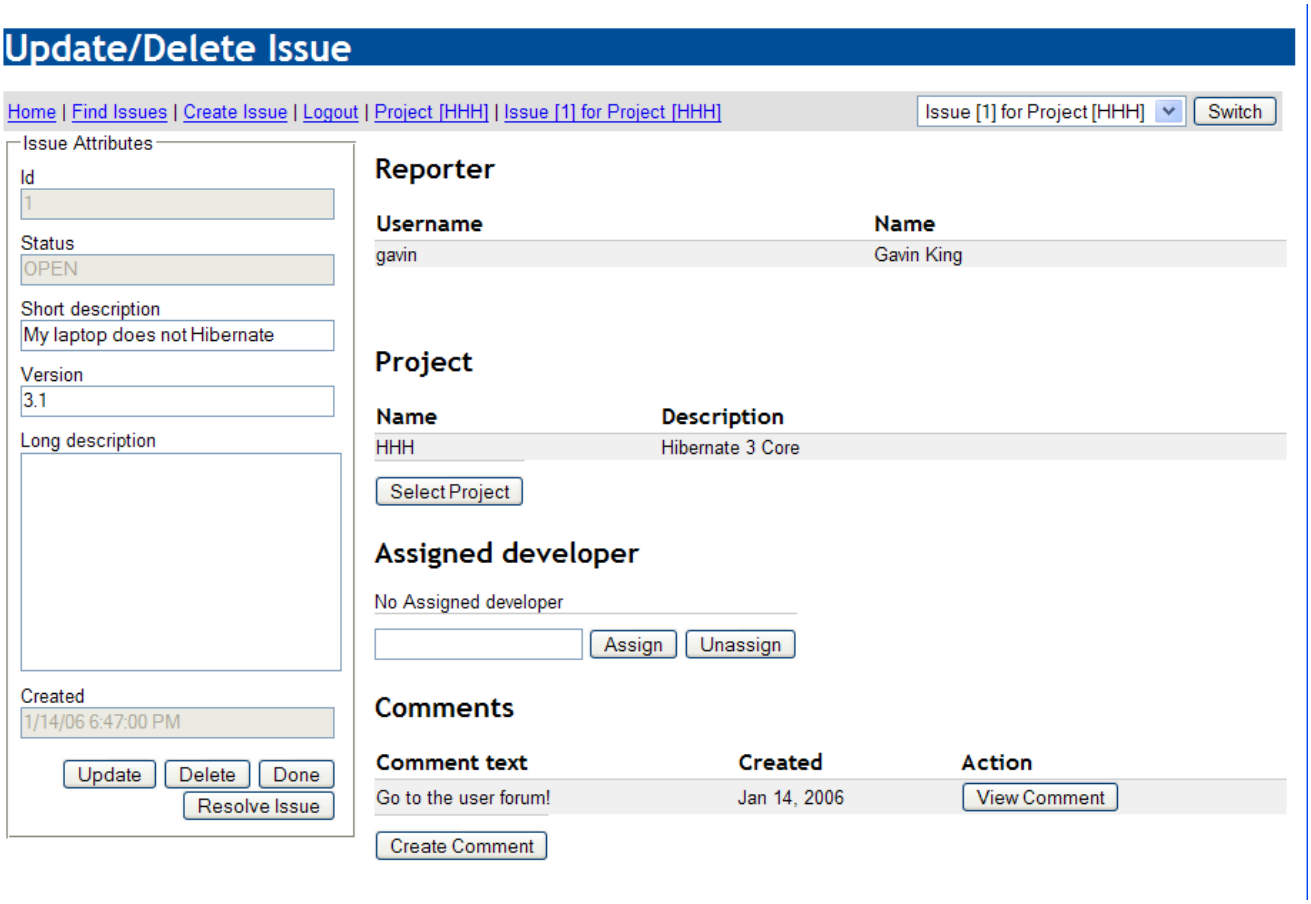
The administration screens take use jBPM to manage the approval and shipping cycle for orders. The business process may even be changed dynamically, by selecting a different process definition!

TODO

Look in the `dvdstore` directory.

1.8. A complete application featuring Seam workspace management: the Issue Tracker example

The Issue Tracker demo shows off Seam's workspace management functionality: the conversation switcher, conversation list and breadcrumbs.



TODO

Look in the `issues` directory.

1.9. An example of Seam with Hibernate: the Hibernate Booking example

The Hibernate Booking demo is a straight port of the Booking demo to an alternative architecture that uses Hibernate for persistence and JavaBeans instead of session beans.

TODO

Look in the `hibernate` directory.

1.10. A RESTful Seam application: the Blog example

Seam makes it very easy to implement applications which keep state on the server-side. However, server-side state is not always appropriate, especially in for functionality that serves up *content*. For this kind of problem we often need to let the user bookmark pages and have a relatively stateless server, so that any page can be accessed at any time, via the bookmark. The Blog example shows how to a implement RESTful application using Seam. Every page of the application can be bookmarked, including the search results page.

The Blog example demonstrates the use of "pull"-style MVC, where instead of using action listener methods to retrieve data and prepare the data for the view, the view pulls data from components as it is being rendered.

1.10.1. Using "pull"-style MVC

This snippet from the `index.xhtml` facelets page displays a list of recent blog entries:

Example 1.25.

```
<h:dataTable value="#{blog.recentBlogEntries}" var="blogEntry" rows="3">
   <h:column>
      <div class="blogEntry">
         <h3>#{blogEntry.title}</h3>
         <div>
            <h:outputText escape="false"
                  value="#{blogEntry.excerpt==null ? blogEntry.body : blogEntry.excerpt}"/>
         </div>
         <p>
            <h:outputLink value="entry.seam" rendered="#{blogEntry.excerpt!=null}">
               <f:param name="blogEntryId" value="#{blogEntry.id}"/>
               Read more...
            </h:outputLink>
         </p>
         <p>
            [Posted on
            <h:outputText value="#{blogEntry.date}">
               <f:convertDateTime timeZone="#{blog.timeZone}" locale="#{blog.locale}" type="both"/>
```

```
            </h:outputText>]
             
            <h:outputLink value="entry.seam">[Link]
               <f:param name="blogEntryId" value="#{blogEntry.id}"/>
            </h:outputLink>
         </p>
      </div>
   </h:column>
</h:dataTable>
```

If we navigate to this page from a bookmark, how does the data used by the `<h:dataTable>` actually get initialized? Well, what happens is that the `Blog` is retrieved lazily—"pulled"—when needed, by a Seam component named `blog`. This is the opposite flow of control to what is usual in traditional web action-based frameworks like Struts.

Example 1.26.

```
@Name("blog")
@Scope(ScopeType.STATELESS)
public class BlogService
{

   @In                                                                    (1)
   private EntityManager entityManager;

   @Unwrap                                                                (2)
   public Blog getBlog()
   {
      return (Blog) entityManager.createQuery("from Blog b left join fetch b.blogEntries")
            .setHint("org.hibernate.cacheable", true)
            .getSingleResult();
   }

}
```

(1) This component uses a *seam-managed persistence context*. Unlike the other examples we've seen, this persistence context is managed by Seam, instead of by the EJB3 container. The persistence context spans the entire web request, allowing us to avoid any exceptions that occur when accessing unfetched associations in the view.

(2) The `@Unwrap` annotation tells Seam to provide the return value of the method—the `Blog`—instead of the actual `BlogService` component to clients. This is the Seam *manager component pattern*.

This is good so far, but what about bookmarking the result of form submissions, such as a search results page?

1.10.2. Bookmarkable search results page

The blog example has a tiny form in the top right of each page that allows the user to search for blog entries. This is defined in a file, `menu.xhtml`, included by the facelets template, `template.xhtml`:

Example 1.27.

```
<div id="search">
   <h:form>
      <h:inputText value="#{searchAction.searchPattern}"/>
      <h:commandButton value="Search" action="/search.xhtml"/>
   </h:form>
```

```
</div>
```

To implement a bookmarkable search results page, we need to perform a browser redirect after processing the search form submission. Because we used the JSF view id as the action outcome, Seam automatically redirects to the view id when the form is submitted. Alternatively, we could have defined a navigation rule like this:

Example 1.28.

```
<navigation-rule>
   <navigation-case>
      <from-outcome>searchResults</from-outcome>
      <to-view-id>/search.xhtml</to-view-id>
      <redirect/>
   </navigation-case>
</navigation-rule>
```

Then the form would have looked like this:

Example 1.29.

```
<div id="search">
   <h:form>
      <h:inputText value="#{searchAction.searchPattern}"/>
      <h:commandButton value="Search" action="searchResults"/>
   </h:form>
</div>
```

But when we redirect, we need to include the values submitted with the form as request parameters, to get a bookmarkable URL like `http://localhost:8080/seam-blog/search.seam?searchPattern=seam`. JSF does not provide an easy way to do this, but Seam does. We use a Seam *page parameter*, defined in `WEB-INF/pages.xml`:

Example 1.30.

```
<pages>
   <page view-id="/search.xhtml">
      <param name="searchPattern" value="#{searchService.searchPattern}"/>
   </page>
   ...
</pages>
```

This tells Seam to include the value of `#{searchService.searchPattern}` as a request parameter named `searchPattern` when redirecting to the page, and then re-apply the value of that parameter to the model before rendering the page.

The redirect takes us to the `search.xhtml` page:

Example 1.31.

```
<h:dataTable value="#{searchResults}" var="blogEntry">
   <h:column>
```

```
      <div>
         <h:outputLink value="entry.seam">
            <f:param name="blogEntryId" value="#{blogEntry.id}"/>
            #{blogEntry.title}
         </h:outputLink>
         posted on
         <h:outputText value="#{blogEntry.date}">
            <f:convertDateTime timeZone="#{blog.timeZone}" locale="#{blog.locale}" type="both"/>
         </h:outputText>
      </div>
   </h:column>
</h:dataTable>
```

Which again uses "pull"-style MVC to retrieve the actual search results:

Example 1.32.

```
@Name("searchService")
public class SearchService
{

   @In
   private EntityManager entityManager;

   private String searchPattern;

   @Factory("searchResults")
   public List<BlogEntry> getSearchResults()
   {
      if (searchPattern==null)
      {
         return null;
      }
      else
      {
         return entityManager.createQuery("select be from BlogEntry be where lower(be.title) like :sea
               .setParameter( "searchPattern", getSqlSearchPattern() )
               .setMaxResults(100)
               .getResultList();
      }
   }

   private String getSqlSearchPattern()
   {
      return searchPattern==null ? "" : '%' + searchPattern.toLowerCase().replace('*', '%').replace('
   }

   public String getSearchPattern()
   {
      return searchPattern;
   }

   public void setSearchPattern(String searchPattern)
   {
      this.searchPattern = searchPattern;
   }

}
```

1.10.3. Using "push"-style MVC in a RESTful application

Very occasionally, it makes more sense to use push-style MVC for processing RESTful pages, and so Seam

provides the notion of a *page action*. The Blog example uses a page action for the blog entry page, `entry.xhtml`. Note that this is a little bit contrived, it would have been easier to use pull-style MVC here as well.

The `entryAction` component works much like an action class in a traditional push-MVC action-oriented framework like Struts:

Example 1.33.

```
@Name("entryAction")
@Scope(STATELESS)
public class EntryAction
{
   @In(create=true)
   private Blog blog;

   @Out
   private BlogEntry blogEntry;

   public void loadBlogEntry(String id) throws EntryNotFoundException
   {
      blogEntry = blog.getBlogEntry(id);
      if (blogEntry==null) throw new EntryNotFoundException(id);
   }

}
```

Page actions are also declared in `pages.xml`:

Example 1.34.

```
<pages>
   ...

   <page view-id="/entry.xhtml" action="#{entryAction.loadBlogEntry(blogEntry.id)}">
      <param name="blogEntryId" value="#{blogEntry.id}"/>
   </page>

   <page view-id="/post.xhtml" action="#{loginAction.challenge}"/>

   <page view-id="*" action="#{blog.hitCount.hit}"/>

</pages>
```

Notice that the example is using page actions for some other functionality—the login challenge, and the pageview counter. Also notice the use of a parameter in the page action method binding. This is not a standard feature of JSF EL, but Seam lets you use it, not just for page actions, but also in JSF method bindings.

When the `entry.xhtml` page is requested, Seam first binds the page parameter `blogEntryId` to the model, then runs the page action, which retrieves the needed data—the `blogEntry`—and places it in the Seam event context. Finally, the following is rendered:

Example 1.35.

```
<div class="blogEntry">
   <h3>#{blogEntry.title}</h3>
```

```
   <div>
      <h:outputText escape="false" value="#{blogEntry.body}"/>
   </div>
   <p>
      [Posted on 
      <h:outputText value="#{blogEntry.date}">
         <f:convertDateTime timezone="#{blog.timeZone}" locale="#{blog.locale}" type="both"/>
      </h:outputText>]
   </p>
</div>
```

If the blog entry is not found in the database, the `EntryNotFoundException` exception is thrown. We want this exception to result in a 404 error, not a 505, so we annotate the exception class:

Example 1.36.

```
@ApplicationException(rollback=true)
@HttpError(errorCode=HttpServletResponse.SC_NOT_FOUND)
public class EntryNotFoundException extends Exception
{
   EntryNotFoundException(String id)
   {
      super("entry not found: " + id);
   }
}
```

An alternative implementation of the example does not use the parameter in the method binding:

Example 1.37.

```
@Name("entryAction")
@Scope(STATELESS)
public class EntryAction
{
   @In(create=true)
   private Blog blog;

   @In @Out
   private BlogEntry blogEntry;

   public void loadBlogEntry() throws EntryNotFoundException
   {
      blogEntry = blog.getBlogEntry( blogEntry.getId() );
      if (blogEntry==null) throw new EntryNotFoundException(id);
   }

}
```

```
<pages>
   ...

   <page view-id="/entry.xhtml" action="#{entryAction.loadBlogEntry}">
      <param name="blogEntryId" value="#{blogEntry.id}"/>
   </page>

   ...
</pages>
```

It is a matter of taste which implementation you prefer.

Chapter 2. Getting started with Seam, using seam-gen

The Seam distribution includes a command line utility that makes it really easy to set up an Eclipse project, generate some simple Seam skeleton code, and reverse engineer an application from a pre-existing database.

This is the easy way to get your feet wet with Seam, and gives you some ammunition for next time you find yourself trapped in an elevator with one of those tedious Ruby guys ranting about how great and wonderful his new toy is for building totally trivial applications that put things in databases.

In this release, seam-gen only works for people who want to use Seam with EJB 3.0 in JBoss AS. Future versions will support other deployment environments.

You *can* use seam-gen without Eclipse, but in this tutorial, we want to show you how to use it in conjunction with Eclipse for debugging and integration testing. If you don't want to install Eclipse, you can still follow along with this tutorial—all steps can be peformed from the command line.

Seam-gen is basically just a big ugly Ant script wrapped around Hibernate Tools, together with some templates. Which means it is easy to customize if you need to.

2.1. Before you start

Make sure you have JDK 5 or JDK 6, JBoss AS 4.0.5 and Ant 1.6, along with recent versions of Eclipse, the JBoss IDE plugin for Eclipse and the TestNG plugin for Eclipse correctly installed before starting. Add your JBoss installation to the JBoss Server View in Eclipse. Start JBoss in debug mode. Finally, start a command prompt in the directory where you unzipped the Seam distribution.

2.2. Setting up a new Eclipse project

The first thing we need to do is configure seam-gen for your environment: JBoss AS installation directory, Eclipse workspace, and database connection. It's easy, just type:

```
cd jboss-seam-1.1.x
seam setup
```

And you will be prompted for the needed information:

```
C:\Projects\jboss-seam>seam setup
Buildfile: C:\Projects\jboss-seam\seam-gen\build.xml

setup:
    [echo] Welcome to seam-gen :-)
    [input] Enter your Java project workspace [C:/Projects]

    [input] Enter your JBoss home directory [C:/Program Files/jboss-4.0.5.GA]

    [input] Enter the project name [myproject]
helloworld
    [input] Enter the Java package name for your session beans [com.mydomain.helloworld]
org.jboss.helloworld
    [input] Enter the Java package name for your entity beans [org.jboss.helloworld]

    [input] Enter the Java package name for your test cases [org.jboss.helloworld.test]

    [input] What kind of database are you using? [hsql] (hsql,mysql,oracle,postgres,mssql,db2,sybase,
```

```
mysql
    [input] Enter the Hibernate dialect for your database [org.hibernate.dialect.MySQLDialect]

    [input] Enter the filesystem path to the JDBC driver jar [lib/hsqldb.jar]
../../mysql-connector.jar
    [input] Enter JDBC driver class for your database [com.mysql.jdbc.Driver]

    [input] Enter the JDBC URL for your database [jdbc:mysql:///test]

    [input] Enter database username [sa]
gavin
    [input] Enter database password []

    [input] Are you working with tables that already exist in the database? [n] (y,n,)
y
[propertyfile] Creating new property file: C:\Projects\jboss-seam\seam-gen\build.properties
     [echo] Installing JDBC driver jar to JBoss server
     [echo] Type 'seam new-project' to create the new project

BUILD SUCCESSFUL
Total time: 1 minute 17 seconds
C:\Projects\jboss-seam>
```

The tool provides sensible defaults, which you can accept by just pressing enter at the prompt.

The settings are stored in `seam-gen/build.properties`, but you can also modify them simply by running `seam setup` a second time.

Now we can create a new project in our Eclipse workspace directory, by typing:

```
seam new-project
```

```
C:\Projects\jboss-seam>seam new-project
Buildfile: C:\Projects\jboss-seam\seam-gen\build.xml

validate-workspace:

validate-project:

copy-lib:
     [echo] Copying project jars ...
     [copy] Copying 32 files to C:\Projects\helloworld\lib
     [copy] Copying 9 files to C:\Projects\helloworld\embedded-ejb

file-copy-wtp:

file-copy:
     [echo] Copying project resources ...
     [copy] Copying 12 files to C:\Projects\helloworld\resources
     [copy] Copying 1 file to C:\Projects\helloworld\resources
     [copy] Copying 5 files to C:\Projects\helloworld\view
     [copy] Copying 5 files to C:\Projects\helloworld
    [mkdir] Created dir: C:\Projects\helloworld\src

new-project:
     [echo] A new Seam project was created in the C:/Projects directory
     [echo] Add the project from inside Eclipse (or type 'seam explode') and go to http://localhost:
8080/helloworld

BUILD SUCCESSFUL
Total time: 7 seconds
C:\Projects\jboss-seam>
```

This copies the Seam jars, dependent jars and the JDBC driver jar to a new Eclipse project, and generates all needed resources and configuration files, a facelets template file and stylesheet, along with Eclipse metadata and an Ant build script. The Eclipse project will be automatically deployed to an exploded directory structure in

JBoss AS as soon as you add the project using `New -> Project... -> Java Project -> Next`, typing the `Project name` (`myproject` in this case), selecting your Java SE 5 or Java SE 6 JRE and then clicking `Finish`. Do not select `Create new project from existing source`. Alternatively, you can deploy the project from outside Eclipse by typing `seam explode`.

Go to `http://localhost:8080/helloworld` to see a welcome page. This is a facelets page, `view/home.xhtml`, using the template `view/layout/template.xhtml`. You can edit this page, or the template, in eclipse, and see the results *immediately*, by clicking refresh in your browser.

Don't get scared by the XML configuration documents that were generated into the project directory. They are mostly standard Java EE stuff, the stuff you need to create once and then never look at again, and they are 90% the same between all Seam projects. (They are so easy to write that even seam-gen can do it.)

2.3. Creating a new action

If you're used to traditional action-style web frameworks, you're probably wondering how you can create a simple webpage with a stateless action method in Java. If you type:

```
seam new-action
```

Seam will prompt for some information, and generate a new facelets page and Seam component for your project.

```
C:\Projects\jboss-seam>seam new-action ping
Buildfile: C:\Projects\jboss-seam\seam-gen\build.xml

validate-workspace:

validate-project:

action-input:
    [input] Enter the Seam component name
ping
    [input] Enter the local interface name [Ping]

    [input] Enter the bean class name [PingBean]

    [input] Enter the action method name [ping]

    [input] Enter the page name [ping]


setup-filters:

new-action:
     [echo] Creating a new stateless session bean component with an action method
     [copy] Copying 1 file to C:\Projects\hello\src\com\hello
     [copy] Copying 1 file to C:\Projects\hello\src\com\hello
     [copy] Copying 1 file to C:\Projects\hello\src\com\hello\test
     [copy] Copying 1 file to C:\Projects\hello\src\com\hello\test
     [copy] Copying 1 file to C:\Projects\hello\view
     [echo] Type 'seam restart' and go to http://localhost:8080/helloworld/ping.seam

BUILD SUCCESSFUL
Total time: 13 seconds
C:\Projects\jboss-seam>
```

Because we've added a new Seam component, we need to restart the exploded directory deployment. You can do this by typing `seam restart`, or by running the `restart` target in the generated project `build.xml` file from inside Eclipse. *You do not need to restart JBoss each time you change the application.*

Now go to `http://localhost:8080/helloworld/ping.seam` and click the button. You can see the code behind this action by looking in the project `src` directory. Put a breakpoint in the `ping()` method, and click the button again. Finally, locate the `PingTest.xml` file in the test package and run the integration tests using the TestNG plugin.

2.4. Creating a form with an action

The next step is to create a form. Type:

```
seam new-form
```

```
C:\Projects\jboss-seam>seam new-form
Buildfile: C:\Projects\jboss-seam\seam-gen\build.xml

validate-workspace:

validate-project:

action-input:
    [input] Enter the Seam component name
hello
    [input] Enter the local interface name [Hello]

    [input] Enter the bean class name [HelloBean]

    [input] Enter the action method name [hello]

    [input] Enter the page name [hello]


setup-filters:

new-form:
     [echo] Creating a new stateful session bean component with an action method
     [copy] Copying 1 file to C:\Projects\hello\src\com\hello
     [copy] Copying 1 file to C:\Projects\hello\src\com\hello
     [copy] Copying 1 file to C:\Projects\hello\src\com\hello\test
     [copy] Copying 1 file to C:\Projects\hello\view
     [copy] Copying 1 file to C:\Projects\hello\src\com\hello\test
     [echo] Type 'seam restart' and go to http://localhost:8080/hello/hello.seam

BUILD SUCCESSFUL
Total time: 5 seconds
C:\Projects\jboss-seam>
```

Restart the application again, and go to `http://localhost:8080/helloworld/hello.seam`. Then take a look at the generated code. Run the test. Try adding some new fields to the form and Seam component (remember to restart the deploment each time you change the Java code).

2.5. Generating an application from an existing database

Manually create some tables in your database. (If you need to switch to a different database, just run `seam setup` again.) Now type:

```
seam generate-entities
```

Restart the deployment, and go to `http://localhost:8080/helloworld`. You can browse the database, edit existing objects, and create new objects. If you look at the generated code, you'll probably be amazed how simple

it is! Seam was designed so that data access code is easy to write by hand, even for people who don't want to cheat using seam-gen.

2.6. Deploying the application as an EAR

Finally, we want to be able to deploy the application using standard Java EE 5 packaging. First, we need to remove the exploded directory by running `seam unexplode`. To deploy the EAR, we can type `seam deploy` at the command prompt, or run the `deploy` target of the generated project build script. You can undeploy using `seam undeploy` or the `undeploy` target.

Chapter 3. The contextual component model

The two core concepts in Seam are the notion of a *context* and the notion of a *component*. Components are stateful objects, usually EJBs, and an instance of a component is associated with a context, and given a name in that context. *Bijection* provides a mechanism for aliasing internal component names (instance variables) to contextual names, allowing component trees to be dynamically assembled, and reassembled by Seam.

Let's start by describing the contexts built in to Seam.

3.1. Seam contexts

Seam contexts are created and destroyed by the framework. The application does not control context demarcation via explicit Java API calls. Context are usually implicit. In some cases, however, contexts are demarcated via annotations.

The basic Seam contexts are:

- Stateless context
- Event (or request) context
- Page context
- Conversation context
- Session context
- Business process context
- Application context

You will recognize some of these contexts from servlet and related specifications. However, two of them might be new to you: *conversation context*, and *business process context*. One reason state management in web applications is so fragile and error-prone is that the three built-in contexts (request, session and application) are not especially meaningful from the point of view of the business logic. A user login session, for example, is a fairly arbitrary construct in terms of the actual application work flow. Therefore, most Seam components are scoped to the conversation or business process contexts, since they are the contexts which are most meaningful in terms of the application.

Let's look at each context in turn.

3.1.1. Stateless context

Components which are truly stateless (stateless session beans, primarily) always live in the stateless context (this is really a non-context). Stateless components are not very interesting, and are arguably not very object-oriented. Nevertheless, they are important and often useful.

3.1.2. Event context

The event context is the "narrowest" stateful context, and is a generalization of the notion of the web request context to cover other kinds of events. Nevertheless, the event context associated with the lifecycle of a JSF re-

quest is the most important example of an event context, and the one you will work with most often. Components associated with the event context are destroyed at the end of the request, but their state is available and well-defined for at least the lifecycle of the request.

When you invoke a Seam component via RMI, or Seam Remoting, the event context is created and destroyed just for the invocation.

3.1.3. Page context

The page context allows you to associate state with a particular instance of a rendered page. You can initialize state in your event listener, or while actually rendering the page, and then have access to it from any event that originates from that page. This is especially useful for functionality like clickable lists, where the list is backed by changing data on the server side. The state is actually serialized to the client, so this construct is extremely robust with respect to multi-window operation and the back button.

3.1.4. Conversation context

The conversation context is a truly central concept in Seam. A *conversation* is a unit of work from the point of view of the user. It might span several interactions with the user, several requests, and several database transactions. But to the user, a conversation solves a single problem. For example, "book hotel", "approve contract", "create order" are all conversations. You might like to think of a conversation implementing a single "use case" or "user story", but the relationship is not necessarily quite exact.

A conversation holds state associated with "what the user is doing now, in this window". A single user may have multiple conversations in progress at any point in time, usually in multiple windows. The conversation context allows us to ensure that state from the different conversations does not collide and cause bugs.

It might take you some time to get used to thinking of applications in terms of conversations. But once you get used to it, we think you'll love the notion, and never be able to not think in terms of conversations again!

Some conversations last for just a single request. Conversations that span multiple requests must be demarcated using annotations provided by Seam.

Some conversations are also *tasks*. A task is a conversation that is significant in terms of a long-running business process, and has the potential to trigger a business process state transition when it is successfully completed. Seam provides a special set of annotations for task demarcation.

Conversations may be *nested*, with one conversation taking place "inside" a wider conversation. This is an advanced feature.

Usually, conversation state is actually held by Seam in the servlet session between requests. Seam implements configurable *conversation timeout*, automatically destroying inactive conversations, and thus ensuring that the state held by a single user login session does not grow without bound if the user abandons conversations.

Seam serializes processing of concurrent requests that take place in the same long-running conversation context, in the same process.

Alternatively, Seam may be configured to keep conversational state in the client browser.

3.1.5. Session context

A session context holds state associated with the user login session. While there are some cases where it is use-

ful to share state between several conversations, we generally frown on the use of session context for holding components other than global information about the logged in user.

In a JSR-168 portal environment, the session context represents the portlet session.

3.1.6. Business process context

The business process context holds state associated with the long running business process. This state is managed and made persistent by the BPM engine (JBoss jBPM). The business process spans multiple interactions with multiple users, so this state is shared between multiple users, but in a well-defined manner. The current task determines the current business process instance, and the lifecycle of the business process is defined externally using a *process definition language*, so there are no special annotations for business process demarcation.

3.1.7. Application context

The application context is the familiar servlet context from the servlet spec. Application context is mainly useful for holding static information such as configuration data, reference data or metamodels. For example, Seam stores its own configuration and metamodel in the application context.

3.1.8. Context variables

A context defines a namespace, a set of *context variables*. These work much the same as session or request attributes in the servlet spec. You may bind any value you like to a context variable, but usually we bind Seam component instances to context variables.

So, within a context, a component instance is identified by the context variable name (this is usually, but not always, the same as the component name). You may programatically access a named component instance in a particular scope via the `Contexts` class, which provides access to several thread-bound instances of the `Context` interface:

```
User user = (User) Contexts.getSessionContext().get("user");
```

You may also set or change the value associated with a name:

```
Contexts.getSessionContext().set("user", user);
```

Usually, however, we obtain components from a context via injection, and put component instances into a context via outjection.

3.1.9. Context search priority

Sometimes, as above, component instances are obtained from a particular known scope. Other times, all stateful scopes are searched, in *priority order*. The order is as follows:

- Event context
- Page context
- Conversation context

- Session context
- Business process context
- Application context

You can perform a priority search by calling `Contexts.lookupInStatefulContexts()`. Whenever you access a component by name from a JSF page, a priority search occurs.

3.1.10. Concurrency model

Neither the servlet nor EJB specifications define any facilities for managing concurrent requests originating from the same client. The servlet container simply lets all threads run concurrently and leaves enforcing thread-safeness to application code. The EJB container allows stateless components to be accessed concurrently, and throws an exception if multiple threads access a stateful session bean.

This behavior might have been okay in old-style web applications which were based around fine-grained, synchronous requests. But for modern applications which make heavy use of many fine-grained, asynchronous (AJAX) requests, concurrency is a fact of life, and must be supported by the programming model. Seam weaves a concurrency management layer into its context model.

The Seam session and application contexts are multithreaded. Seam will allow concurrent requests in a context to be processed concurrently. The event and page contexts are by nature single threaded. The business process context is strictly speaking multi-threaded, but in practice concurrency is sufficiently rare that this fact may be disregarded most of the time. Finally, Seam enforces a *single thread per conversation per process* model for the conversation context by serializing concurrent requests in the same long-running conversation context.

Since the session context is multithreaded, and often contains volatile state, session scope components are always protected by Seam from concurrent access. Seam serializes requests to session scope session beans and JavaBeans by default (and detects and breaks any deadlocks that occur). This is not the default behaviour for application scoped components however, since application scoped components do not usually hold volatile state and because synchronization at the global level is *extremely* expensive. However, you can force a serialized threading model on any session bean or JavaBean component by adding the `@Synchronized` annotation.

This concurrency model means that AJAX clients can safely use volatile session and conversational state, without the need for any special work on the part of the developer.

3.2. Seam components

Seam components are POJOs (Plain Old Java Objects). In particular, they are JavaBeans or EJB 3.0 enterprise beans. While Seam does not require that components be EJBs and can even be used without an EJB 3.0 compliant container, Seam was designed with EJB 3.0 in mind and includes deep integration with EJB 3.0. Seam supports the following *component types*.

- EJB 3.0 stateless session beans
- EJB 3.0 stateful session beans
- EJB 3.0 entity beans
- JavaBeans

- EJB 3.0 message-driven beans

3.2.1. Stateless session beans

Stateless session bean components are not able to hold state across multiple invocations. Therefore, they usually work by operating upon the state of other components in the various Seam contexts. They may be used as JSF action listeners, but cannot provide properties to JSF components for display.

Stateless session beans always live in the stateless context.

Stateless session beans are the least interesting kind of Seam component.

3.2.2. Stateful session beans

Stateful session bean components are able to hold state not only across multiple invocations of the bean, but also across multiple requests. Application state that does not belong in the database should usually be held by stateful session beans. This is a major difference between Seam and many other web application frameworks. Instead of sticking information about the current conversation directly in the `HttpSession`, you should keep it in instance variables of a stateful session bean that is bound to the conversation context. This allows Seam to manage the lifecycle of this state for you, and ensure that there are no collisions between state relating to different concurrent conversations.

Stateful session beans are often used as JSF action listener, and as backing beans that provide properties to JSF components for display or form submission.

By default, stateful session beans are bound to the conversation context. They may never be bound to the page or stateless contexts.

Concurrent requests to session-scoped stateful session beans are always serialized by Seam.

3.2.3. Entity beans

Entity beans may be bound to a context variable and function as a seam component. Because entities have a persistent identity in addition to their contextual identity, entity instances are usually bound explicitly in Java code, rather than being instantiated implicitly by Seam.

Entity bean components do not support bijection or context demarcation. Nor does invocation of an entity bean trigger validation.

Entity beans are not usually used as JSF action listeners, but do often function as backing beans that provide properties to JSF components for display or form submission. In particular, it is common to use an entity as a backing bean, together with a stateless session bean action listener to implement create/update/delete type functionality.

By default, entity beans are bound to the conversation context. They may never be bound to the stateless context.

Note that it in a clustered environment is somewhat less efficient to bind an entity bean directly to a conversation or session scoped Seam context variable than it would be to hold a reference to the entity bean in a stateful session bean. For this reason, not all Seam applications define entity beans to be Seam components.

3.2.4. JavaBeans

Javabeans may be used just like a stateless or stateful session bean. However, they do not provide the functionality of a session bean (declarative transaction demarcation, declarative security, efficient clustered state replication, EJB 3.0 persistence, timeout methods, etc).

In a later chapter, we show you how to use Seam and Hibernate without an EJB container. In this use case, components are JavaBeans instead of session beans. Note, however, that in many application servers it is somewhat less efficient to cluster conversation or session scoped Seam JavaBean components than it is to cluster stateful session bean components.

By default, JavaBeans are bound to the event context.

Concurrent requests to session-scoped JavaBeans are always serialized by Seam.

3.2.5. Message-driven beans

Message-driven beans may function as a seam component. However, message-driven beans are called quite differently to other Seam components - instead of invoking them via the context variable, they listen for messages sent to a JMS queue or topic.

Message-driven beans may not be bound to a Seam context. Nor do they have access to the session or conversation state of their "caller". However, they do support bijection and some other Seam functionality.

3.2.6. Interception

In order to perform its magic (bijection, context demarcation, validation, etc), Seam must intercept component invocations. For JavaBeans, Seam is in full control of instantiation of the component, and no special configuration is needed. For entity beans, interception is not required since bijection and context demarcation are not defined. For session beans, we must register an EJB interceptor for the session bean component. We could use an annotation, as follows:

```
@Stateless
@Interceptors(SeamInterceptor.class)
public class LoginAction implements Login {
    ...
}
```

But a much better way is to define the interceptor in `ejb-jar.xml`.

```
<interceptors>
   <interceptor>
      <interceptor-class>org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
   </interceptor>
</interceptors>

<assembly-descriptor>
   <interceptor-binding>
      <ejb-name>*</ejb-name>
      <interceptor-class>org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
   </interceptor-binding>
</assembly-descriptor>
```

3.2.7. Component names

Almost all seam components need a name. We assign a name to a component using the `@Name` annotation:

```
@Name("loginAction")
```

```
@Stateless
public class LoginAction implements Login {
    ...
}
```

This name is the *seam component name* and is not related to any other name defined by the EJB specification. However, seam component names work just like JSF managed bean names and you can think of the two concepts as identical.

Just like in JSF, a seam component instance is usually bound to a context variable with the same name as the component name. So, for example, we would access the `LoginAction` using `Contexts.getStatelessContext().get("loginAction")`. In particular, whenever Seam itself instantiates a component, it binds the new instance to a variable with the component name. However, again like JSF, it is possible for the application to bind a component to some other context variable by programmatic API call. This is only useful if a particular component serves more than one role in the system. For example, the currently logged in `User` might be bound to the `currentUser` session context variable, while a `User` that is the subject of some administration functionality might be bound to the `user` conversation context variable.

For very large applications, and for built-in seam components, qualified names are often used.

```
@Name("com.jboss.myapp.loginAction")
@Stateless
@Interceptors(SeamInterceptor.class)
public class LoginAction implements Login {
    ...
}
```

We may use the qualified component name both in Java code and in JSF's expression language:

```
<h:commandButton type="submit" value="Login"
                 action="#{com.jboss.myapp.loginAction.login}"/>
```

Since this is noisy, Seam also provides a means of aliasing a qualified name to a simple name. Add a line like this to the `components.xml` file:

```
<factory name="loginAction" scope="STATELESS" value="#{com.jboss.myapp.loginAction}"/>
```

All of the built-in Seam components have qualified names, but most of them are aliased to a simple name by the `components.xml` file included in the Seam jar.

3.2.8. Defining the component scope

We can override the default scope (context) of a component using the `@Scope` annotation. This lets us define what context a component instance is bound to, when it is instantiated by Seam.

```
@Name("user")
@Entity
@Scope(SESSION)
public class User {
    ...
}
```

`org.jboss.seam.ScopeType` defines an enumeration of possible scopes.

3.2.9. Components with multiple roles

Some Seam component classes can fulfill more than one role in the system. For example, we often have a `User` class which is usually used as a session-scoped component representing the current user but is used in user administration screens as a conversation-scoped component. The `@Role` annotation lets us define an additional named role for a component, with a different scope—it lets us bind the same component class to different context variables. (Any Seam component *instance* may be bound to multiple context variables, but this lets us do it at the class level, and take advantage of auto-instantiation.)

```
@Name("user")
@Entity
@Scope(CONVERSATION)
@Role(name="currentUser", scope=SESSION)
public class User {
    ...
}
```

The `@Roles` annotation lets us specify as many additional roles as we like.

```
@Name("user")
@Entity
@Scope(CONVERSATION)
@Roles({@Role(name="currentUser", scope=SESSION)
        @Role(name="tempUser", scope=EVENT)})
public class User {
    ...
}
```

3.2.10. Built-in components

Like many good frameworks, Seam eats its own dogfood and is implemented mostly as a set of built-in Seam interceptors (see later) and Seam components. This makes it easy for applications to interact with built-in components at runtime or even customize the basic functionality of Seam by replacing the built-in components with custom implementations. The built-in components are defined in the Seam namespace `org.jboss.seam.core` and the Java package of the same name.

The built-in components may be injected, just like any Seam components, but they also provide convenient static `instance()` methods:

```
FacesMessages.instance().add("Welcome back, #{user.name}!");
```

Seam was designed to integrate tightly in a Java EE 5 environment. However, we understand that there are many projects which are not running in a full EE environment. We also realize the critical importance of easy unit and integration testing using frameworks such as TestNG and JUnit. So, we've made it easy to run Seam in Java SE environments by allowing you to boostrap certain critical infrastructure normally only found in EE environments by installing built-in Seam components.

For example, you can run your EJB3 components in Tomcat or an integration test suite just by installing the built-in component `org.jboss.seam.core.ejb`, which automatically bootstraps the JBoss Embeddable EJB3 container and deploys your EJB components.

Or, if you're not quite ready for the Brave New World of EJB 3.0, you can write a Seam application that uses only JavaBean components, together with Hibernate3 for persistence, by installing a built-in component that manages a Hibernate `SessionFactory`. When using Hibernate outside of a J2EE environment, you will also probably need a JTA transaction manager and JNDI server, which are available via the built-in component `org.jboss.seam.core.microcontainer`. This lets you use the bulletproof JTA/JCA pooling datasource from JBoss application server in an SE environment like Tomcat!

3.3. Bijection

Dependency injection or *inversion of control* is by now a familiar concept to most Java developers. Dependency injection allows a component to obtain a reference to another component by having the container "inject" the other component to a setter method or instance variable. In all dependency injection implementations that we have seen, injection occurs when the component is constructed, and the reference does not subsequently change for the lifetime of the component instance. For stateless components, this is reasonable. From the point of view of a client, all instances of a particular stateless component are interchangeable. On the other hand, Seam emphasizes the use of stateful components. So traditional dependency injection is no longer a very useful construct. Seam introduces the notion of *bijection* as a generalization of injection. In contrast to injection, bijection is:

- *contextual* - bijection is used to assemble stateful components from various different contexts (a component from a "wider" context may even have a reference to a component from a "narrower" context)
- *bidirectional* - values are injected from context variables into attributes of the component being invoked, and also *outjected* from the component attributes back out to the context, allowing the component being invoked to manipulate the values of contextual variables simply by setting its own instance variables
- *dynamic* - since the value of contextual variables changes over time, and since Seam components are stateful, bijection takes place every time a component is invoked

In essence, bijection lets you alias a context variable to a component instance variable, by specifying that the value of the instance variable is injected, outjected, or both. Of course, we use annotations to enable bijection.

The `@In` annotation specifies that a value should be injected, either into an instance variable:

```
@Name("loginAction")
@Stateless
@Interceptors(SeamInterceptor.class)
public class LoginAction implements Login {
    @In User user;
    ...
}
```

or into a setter method:

```
@Name("loginAction")
@Stateless
@Interceptors(SeamInterceptor.class)
public class LoginAction implements Login {
    User user;

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

    ...
}
```

By default, Seam will do a priority search of all contexts, using the name of the property or instance variable that is being injected. You may wish to specify the context variable name explicitly, using, for example, `@In("currentUser")`.

If you want Seam to create an instance of the component when there is no existing component instance bound to the named context variable, you should specify `@In(create=true)`. If the value is optional (it can be null),

specify `@In(required=false)`.

You can even inject the value of an expression:

```
@Name("loginAction")
@Stateless
@Interceptors(SeamInterceptor.class)
public class LoginAction implements Login {
    @In("#{user.username}") String username;
    ...
}
```

(There is much more information about component lifecycle and injection in the next chapter.)

The `@Out` annotation specifies that an attribute should be outjected, either from an instance variable:

```
@Name("loginAction")
@Stateless
@Interceptors(SeamInterceptor.class)
public class LoginAction implements Login {
    @Out User user;
    ...
}
```

or from a getter method:

```
@Name("loginAction")
@Stateless
@Interceptors(SeamInterceptor.class)
public class LoginAction implements Login {
    User user;

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

    ...
}
```

An attribute may be both injected and outjected:

```
@Name("loginAction")
@Stateless
@Interceptors(SeamInterceptor.class)
public class LoginAction implements Login {
    @In @Out User user;
    ...
}
```

or:

```
@Name("loginAction")
@Stateless
@Interceptors(SeamInterceptor.class)
public class LoginAction implements Login {
    User user;

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

    @Out
    public User getUser() {
        return user;
```

```
    }

    ...
}
```

3.4. Lifecycle methods

Session bean and entity bean Seam components support all the usual EJB 3.0 lifecycle callback (`@PostConstruct`, `@PreDestroy`, etc). Seam extends all of these callbacks except `@PreDestroy` to JavaBean components. But Seam also defines its own component lifecycle callbacks.

The `@Create` method is called every time Seam instantiates a component. Unlike the `@PostConstruct` method, this method is called after the component is fully constructed by the EJB container, and has access to all the usual Seam functionality (bijection, etc). Components may define only one `@Create` method.

The `@Destroy` method is called when the context that the Seam component is bound to ends. Components may define only one `@Destroy` method. Stateful session bean components *must* define a method annotated `@Destroy` `@Remove`.

Finally, a related annotation is the `@Startup` annotation, which may be applied to any application or session scoped component. The `@Startup` annotation tells Seam to instantiate the component immediately, when the context begins, instead of waiting until it is first referenced by a client. It is possible to control the order of instantiation of startup components by specifying `@Startup(depends={....})`.

3.5. Logging

Who is not totally fed up with seeing noisy code like this?

```
private static final Log log = LogFactory.getLog(CreateOrderAction.class);

public Order createOrder(User user, Product product, int quantity) {
    if ( log.isDebugEnabled() ) {
        log.debug("Creating new order for user: " + user.username() +
            " product: " + product.name()
            + " quantity: " + quantity);
    }
    return new Order(user, product, quantity);
}
```

It is difficult to imagine how the code for a simple log message could possibly be more verbose. There is more lines of code tied up in logging than in the actual business logic! I remain totally astonished that the Java community has not come up with anything better in 10 years.

Seam provides a logging API built on top of Apache commons-logging that simplifies this code significantly:

```
@Logger private Log log;

public Order createOrder(User user, Product product, int quantity) {
    log.debug("Creating new order for user: #0 product: #1 quantity: #2", user.username(), product.nam
    return new Order(user, product, quantity);
}
```

Note that we don't need the noisy `if ( log.isDebugEnabled() )` guard, since string concatenation happens *inside* the `debug()` method. Note also that we don't usually need to specify the log category explicitly, since Seam knows what component it is injecting the `Log` into.

If `User` and `Product` are Seam components available in the current contexts, it gets even better:

```
@Logger private Log log;

public Order createOrder(User user, Product product, int quantity) {
    log.debug("Creating new order for user: #{user.username} product: #{product.name} quantity: #0", q
    return new Order(user, product, quantity);
}
```

3.6. The `Mutable` interface and `@ReadOnly`

Many application servers feature an amazingly broken implementation of `HttpSession` clustering, where changes to the state of mutable objects bound to the session are only replicated when the application calls `setAttribute()` explicitly. This is a source of bugs that can not effectively be tested for at development time, since they will only manifest when failover occurs. Furthermore, the actual replication message contains the entire serialized object graph bound to the session attribute, which inefficient.

Of course, EJB stateful session beans must perform automatic dirty checking and replication of mutable state and a sophisticated EJB container can introduce optimizations such as attribute-level replication. Unfortunately, not all Seam users have the good fortune to be working in an environment that supports EJB 3.0. So, for session and conversation scoped JavaBean and entity bean components, Seam provides an extra layer of cluster-safe state management over the top of the web container session clustering.

For session or conversation scoped JavaBean components, Seam automatically forces replication to occur by calling `setAttribute()` once in every request that the component was invoked by the application. Of course, this strategy is inefficient for read-mostly components. You can control this behavior by implementing the `org.jboss.seam.core.Mutable` interface, or by extending `org.jboss.seam.core.AbstractMutable`, and writing your own dirty-checking logic inside the component. For example,

```
@Name("account")
public class Account extends AbstractMutable
{
    private BigDecimal balance;

    public void setBalance(BigDecimal balance)
    {
        setDirty(this.balance, balance);
        this.balance = balance;
    }

    public BigDecimal getBalance()
    {
        return balance;
    }

    ...

}
```

Or, you can use the `@ReadOnly` annotation to achieve a similar effect:

```
@Name("account")
public class Account
{
    private BigDecimal balance;

    public void setBalance(BigDecimal balance)
    {
        setDirty(this.balance, balance);
        this.balance = balance;
```

```
    }

    @ReadOnly
    public BigDecimal getBalance()
    {
        return balance;
    }

    ...

}
```

For session or conversation scoped entity bean components, Seam automatically forces replication to occur by calling `setAttribute()` once in every request, *unless the (conversation-scoped) entity is currently associated with a Seam-managed persistence context, in which case no replication is needed*. This strategy is not necessarily efficient, so session or conversation scope entity beans should be used with care. You can always write a stateful session bean or JavaBean component to "manage" the entity bean instance. For example,

```
@Stateful
@Name("account")
public class AccountManager extends AbstractMutable
{
    private Account account; // an entity bean

    @Unwrap
    public void getAccount()
    {
        return account;
    }

    ...

}
```

Note that the `EntityHome` class in the Seam Application Framework provides a great example of this pattern.

3.7. Factory and manager components

We often need to work with objects that are not Seam components. But we still want to be able to inject them into our components using `@In` and use them in value and method binding expressions, etc. Sometimes, we even need to tie them into the Seam context lifecycle (`@Destroy`, for example). So the Seam contexts can contain objects which are not Seam components, and Seam provides a couple of nice features that make it easier to work with non-component objects bound to contexts.

The *factory component pattern* lets a Seam component act as the instantiator for a non-component object. A *factory method* will be called when a context variable is referenced but has no value bound to it. We define factory methods using the `@Factory` annotation. The factory method binds a value to the context variable, and determines the scope of the bound value. There are two styles of factory method. The first style returns a value, which is bound to the context by Seam:

```
@Factory(scope=CONVERSATION)
public List<Customer> getCustomerList() {
    return ... ;
}
```

The second style is a method of type `void` which binds the value to the context variable itself:

```
@DataModel List<Customer> customerList;
```

```
@Factory("customerList")
public void initCustomerList() {
    customerList = ...  ;
}
```

In both cases, the factory method is called when we reference the `customerList` context variable and its value is null, and then has no further part to play in the lifecycle of the value. An even more powerful pattern is the *manager component pattern*. In this case, we have a Seam component that is bound to a context variable, that manages the value of the context variable, while remaining invisible to clients.

A manager component is any component with an `@Unwrap` method. This method returns the value that will be visable to clients, and is called *every time* a context variable is referenced.

```
@Name("customerList")
@Scope(CONVERSATION)
public class CustomerListManager
{
    ...

    @Unwrap
    public List<Customer> getCustomerList() {
        return ... ;
    }
}
```

This pattern is especially useful if we have some heavyweight object that needs a cleanup operation when the context ends. In this case, the manager component may perform cleanup in the `@Destroy` method.

Chapter 4. Configuring Seam components

The philosophy of minimizing XML-based configuration is extremely strong in Seam. Nevertheless, there are various reasons why we might want to configure a Seam component using XML: to isolate deployment-specific information from the Java code, to enable the creation of re-usable frameworks, to configure Seam's built-in functionality, etc. Seam provides two basic approaches to configuring components: configuration via property settings in a properties file or `web.xml`, and configuration via `components.xml`.

4.1. Configuring components via property settings

Seam components may be provided with configuration properties either via servlet context parameters, or via a properties file named `seam.properties` in the root of the classpath.

The configurable Seam component must expose JavaBeans-style property setter methods for the configurable attributes. If a seam component named `com.jboss.myapp.settings` has a setter method named `setLocale()`, we can provide a property named `com.jboss.myapp.settings.locale` in the `seam.properties` file or as a servlet context parameter, and Seam will set the value of the `locale` attribute whenever it instantiates the component.

The same mechanism is used to configure Seam itself. For example, to set the conversation timeout, we provide a value for `org.jboss.seam.core.manager.conversationTimeout` in `web.xml` or `seam.properties`. (There is a built-in Seam component named `org.jboss.seam.core.manager` with a setter method named `setConversationTimeout()`.)

4.2. Configuring components via `components.xml`

The `components.xml` file is a bit more powerful than property settings. It lets you:

- Configure components that have been installed automatically—including both built-in components, and application components that have been annotated with the `@Name` annotation and picked up by Seam's deployment scanner.
- Install classes with no `@Name` annotation as Seam components—this is most useful for certain kinds of infrastructural components which can be installed multiple times different names (for example Seam-managed persistence contexts).
- Install components that *do* have a `@Name` annotation but are not installed by default because of an `@Install` annotation that indicates the component should not be installed.
- Override the scope of a component.

A `components.xml` file may appear in one of three different places:

- The `WEB-INF` directory of a `war`.
- The `META-INF` directory of a `jar`.
- Any directory of a `jar` that contains classes with an `@Name` annotation.

Usually, Seam components are installed when the deployment scanner discovers a class with a `@Name` annota-

tion sitting in an archive with a `seam.properties` file or a `META-INF/components.xml` file. (Unless the component has an `@Install` annotation indicating it should not be installed by default.) The `components.xml` file lets us handle special cases where we need to override the annotations.

For example, the following `components.xml` file installs the JBoss Embeddable EJB3 container:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core">
    <core:ejb/>
</components>
```

This example does the same thing:

```
<components>
    <component class="org.jboss.seam.core.Ejb"/>
</components>
```

This one installs and configures two different Seam-managed persistence contexts:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core"

    <core:managed-persistence-context name="customerDatabase"
                persistence-unit-jndi-name="java:/customerEntityManagerFactory"/>

    <core:managed-persistence-context name="accountingDatabase"
                persistence-unit-jndi-name="java:/accountingEntityManagerFactory"/>

</components>
```

As does this one:

```
<components>
    <component name="customerDatabase"
               class="org.jboss.seam.core.ManagedPersistenceContext">
        <property name="persistenceUnitJndiName">java:/customerEntityManagerFactory</property>
    </component>

    <component name="accountingDatabase"
               class="org.jboss.seam.core.ManagedPersistenceContext">
        <property name="persistenceUnitJndiName">java:/accountingEntityManagerFactory</property>
    </component>
</components>
```

This example creates a session-scoped Seam-managed persistence context (this is not recommended in practice):

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core"

  <core:managed-persistence-context name="productDatabase" scope="session"
              persistence-unit-jndi-name="java:/productEntityManagerFactory"/>

</components>
```

```
<components>

    <component name="productDatabase"
               scope="session"
               class="org.jboss.seam.core.ManagedPersistenceContext">
        <property name="persistenceUnitJndiName">java:/productEntityManagerFactory</property>
    </component>

</components>
```

The <factory> declaration lets you specify a value or method binding expression that will be evaluated to initialize the value of a context variable when it is first referenced.

```
<components>

    <factory name="contact" method="#{contactManager.loadContact}" scope="CONVERSATION"/>

</components>
```

You can create an "alias" (a second name) for a Seam component like so:

```
<components>

    <factory name="user" value="#{actor}" scope="STATELESS"/>

</components>
```

You can even create an "alias" for a commonly used expression:

```
<components>

    <factory name="contact" value="#{contactManager.contact}" scope="STATELESS"/>

</components>
```

Sometimes we want to reuse the same components.xml file with minor changes during both deployment and testing. Seam lets you place wildcards of the form @wildcard@ in the components.xml file which can be replaced either by your Ant build script (at deployment time) or by providing a file named components.properties in the classpath (at development time). You'll see this approach used in the Seam examples.

4.3. Fine-grained configuration files

If you have a large number of components that need to be configured in XML, it makes much more sense to split up the information in components.xml into many small files. Seam lets you put configuration for a class named, for example, com.helloworld.Hello in a resource named com/helloworld/Hello.component.xml. (You might be familiar with this pattern, since it is the same one we use in Hibernate.) The root element of the file may be either a <components> or <component> element.

The first option lets you define multiple components in the file:

```
<components>
    <component class="com.helloworld.Hello" name="hello">
        <property name="name">#{user.name}</property>
    </component>
    <factory name="message" value="#{hello.message}"/>
</components>
```

The second option only lets you define or configure one component, but is less noisy:

```
<component name="hello">
    <property name="name">#{user.name}</property>
</component>
```

In the second option, the class name is implied by the file in which the component definition appears.

Alternatively, you may put configuration for all classes in the com.helloworld package in com/helloworld/components.xml.

4.4. Configurable property types

Properties of string, primitive or primitive wrapper type may be configured just as you would expect:

```
org.jboss.seam.core.manager.conversationTimeout 60000
```

```
<core:manager conversation-timeout="60000"/>
```

```
<component name="org.jboss.seam.core.manager">
    <property name="conversationTimeout">60000</property>
</component>
```

Arrays, sets and lists of strings or primitives are also supported:

```
org.jboss.seam.core.jbpm.processDefinitions order.jpdl.xml, return.jpdl.xml, inventory.jpdl.xml
```

```
<core:jbpm>
    <core:process-definitions>
        <value>order.jpdl.xml</value>
        <value>return.jpdl.xml</value>
        <value>inventory.jpdl.xml</value>
    </core:process-definitions>
</core:jbpm>
```

```
<component name="org.jboss.seam.core.jbpm">
    <property name="processDefinitions">
        <value>order.jpdl.xml</value>
        <value>return.jpdl.xml</value>
        <value>inventory.jpdl.xml</value>
    </property>
</component>
```

Even maps with String-valued keys and string or primitive values are supported:

```
<component name="issueEditor">
    <property name="issueStatuses">
        <key>open</key> <value>open issue</value>
        <key>resolved</key> <value>issue resolved by developer</value>
        <key>closed</key> <value>resolution accepted by user</value>
    </property>
</component>
```

Finally, you may wire together components using a value-binding expression. Note that this is quite different to injection using `@In`, since it happens at component instantiation time instead of invocation time. It is therefore much more similar to the dependency injection facilities offered by traditional IoC containers like JSF or Spring.

```
<drools:managed-working-memory name="policyPricingWorkingMemory" rule-base="#{policyPricingRules}"/>
```

```
<component name="policyPricingWorkingMemory"
          class="org.jboss.seam.drools.ManagedWorkingMemory">
    <property name="ruleBase">#{policyPricingRules}</property>
</component>
```

4.5. Using XML Namespaces

Throughout the examples, there have been two competing ways of declaring components: with and without the use of XML namespaces. The following shows a typical `components.xml` file without namespaces. It uses the Seam Components DTD:

```
<?xml version="1.0" encoding="UTF-8">
<!DOCTYPE components PUBLIC "-//JBoss/Seam Component Configuration DTD 1.1//EN"
                            "http://jboss.com/products/seam/components-1.1.dtd">
<components>

    <component class="org.jboss.seam.core.init">
        <property name="debug">true</property>
        <property name="jndiPattern">@jndiPattern@</property>
    </component>

    <component name="org.jboss.sean.core.ejb" installed="@embeddedEjb@" />

</components>
```

As you can see, this is somewhat verbose. Even worse, the component and attribute names cannot be validated at development time.

The namespaced version looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation=
                "http://jboss.com/products/seam/core http://jboss.com/products/seam/core-1.1.xsd
                 http://jboss.com/products/seam/components http://jboss.com/products/seam/components-1

    <core:init debug="true" jndi-pattern="@jndiPattern@"/>

    <core:ejb installed="@embeddedEjb@"/>

</components>
```

Even though the schema declarations are verbose, the actual XML content is lean and easy to understand. The schemas provide detailed information about each component and the attributes available, allowing XML editors to offer intelligent autocomplete. The use of namespaced elements makes generating and maintaining correct `components.xml` files much simpler.

Now, this works great for the built-in Seam components, but what about user components? There are two options. First, Seam supports mixing the two models, allowing the use of the generic `<component>` declarations for user components, along with namespaced declarations for built-in components. But even better, Seam allows you to quickly declare namespaces for your own components.

Any Java package can be associated with an XML namespace by annotating the package with the `@Namespace` annotation. (Package-level annotations are declared in a file named `package-info.java` in the package directory.) Here is an example from the seampay demo:

```
@Namespace(value="http://jboss.com/products/seam/examples/seampay")
package org.jboss.seam.example.seampay;

import org.jboss.seam.annotations.Namespace;
```

That is all you need to do to use the namespaced style in `components.xml`! Now we can write:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:pay="http://jboss.com/products/seam/examples/seampay"
            ... >
```

```
    <pay:payment-home new-instance="#{newPayment}"
                      created-message="Created a new payment to #{newPayment.payee}" />

    <pay:payment name="newPayment"
                 payee="Somebody"
                 account="#{selectedAccount}"
                 payment-date="#{currentDatetime}"
                 created-date="#{currentDatetime}" />
     ...
</components>
```

Or:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:pay="http://jboss.com/products/seam/examples/seampay"
            ... >

    <pay:payment-home>
        <pay:new-instance>"#{newPayment}"</pay:new-instance>
        <pay:created-message>Created a new payment to #{newPayment.payee}</pay:created-message>
    </pay:payment-home>

    <pay:payment name="newPayment">
        <pay:payee>Somebody"</pay:payee>
        <pay:account>#{selectedAccount}</pay:account>
        <pay:payment-date>#{currentDatetime}</pay:payment-date>
        <pay:created-date>#{currentDatetime}</pay:created-date>
     </pay:payment>
     ...
</components>
```

These examples illustrate the two usage models of a namespaced element. In the first declaration, the `<pay:payment-home>` references the `paymentHome` component:

```
package org.jboss.seam.example.seampay;
...
@Name("paymentHome")
public class PaymentController
    extends EntityHome<Payment>
{
    ...
}
```

The element name is the hyphenated form of the component name. The attributes of the element are the hyphenated form of the property names.

In the second declaration, the `<pay:payment>` element refers to the `Payment` class in the `org.jboss.seam.example.seampay` package. In this case `Payment` is an entity that is being declared as a Seam component:

```
package org.jboss.seam.example.seampay;
...
@Entity
public class Payment
    implements Serializable
{
    ...
}
```

If we want validation and autocompletion to work for user-defined components, we will need a schema. Seam does not yet provide a mechanism to automatically generate a schema for a set of components, so it is necessary to generate one manually. The schema definitions for the standard Seam packages can be used for guidance.

The following are the the namespaces used by Seam:

- components — `http://jboss.com/products/seam/components`
- core — `http://jboss.com/products/seam/core`
- drools — `http://jboss.com/products/seam/drools`
- framework — `http://jboss.com/products/seam/framework`
- jms — `http://jboss.com/products/seam/jms`
- remoting — `http://jboss.com/products/seam/remoting`
- theme — `http://jboss.com/products/seam/theme`

Chapter 5. Events, interceptors and exception handling

Complementing the contextual component model, there are two further basic concepts that facilitate the extreme loose-coupling that is the distinctive feature of Seam applications. The first is a strong event model where events may be mapped to event listeners via JSF-like method binding expressions. The second is the pervasive use of annotations and interceptors to apply cross-cutting concerns to components which implement business logic.

5.1. Seam events

The Seam component model was developed for use with *event-driven applications*, specifically to enable the development of fine-grained, loosely-coupled components in a fine-grained eventing model. Events in Seam come in several types, most of which we have already seen:

- JSF events
- jBPM transition events
- Seam page actions
- Seam component-driven events
- Seam contextual events

All of these various kinds of events are mapped to Seam components via JSF EL method binding expressions. For a JSF event, this is defined in the JSF template:

```
<h:commandButton value="Click me!" action="#{helloWorld.sayHello}"/>
```

For a jBPM transition event, it is specified in the jBPM process definition or pageflow definition:

```
<start-page name="hello" view-id="/hello.jsp">
    <transition to="hello">
        <action expression="#{helloWorld.sayHello}"/>
    </transition>
</start-page>
```

You can find out more information about JSF events and jBPM events elsewhere. Lets concentrate for now upon the two additional kinds of events defined by Seam.

5.1.1. Page actions

A Seam page action is an event that occurs just before we render a page. We declare page actions in `WEB-INF/pages.xml`. We can define a page action for either a particular JSF view id:

```
<pages>
    <page view-id="/hello.jsp" action="#{helloWorld.sayHello}"/>
<pages>
```

Or we can use a wildcard to specify an action that applies to all view ids that match the pattern:

```
<pages>
    <page view-id="/hello/*" action="#{helloWorld.sayHello}"/>
<pages>
```

If multiple wildcarded page actions match the current view-id, Seam will call all the actions, in order of least-specific to most-specific.

The page action method can return a JSF outcome. If the outcome is non-null, Seam will delegate to the defined JSFadn Seam navigation rules and a different view may end up being rendered.

Furthermore, the view id mentioned in the `<page>` element need not correspond to a real JSP or Facelets page! So, we can reproduce the functionality of a traditional action-oriented framework like Struts or WebWork using page actions. For example:

```
TODO: translate struts action into page action
```

This is quite useful if you want to do complex things in response to non-faces requests (for example, HTTP GET requests).

Page parameters

A JSF faces request (a form submission) encapsulates both an "action" (a method binding) and "parameters" (input value bindings). A page action might also needs parameters!

Since GET requests are bookmarkable, page parameters are passed as human-readable request parameters. (Unlike JSF form inputs, which are anything but!)

Seam lets us provide a value binding that maps a named request parameter to an attribute of a model object.

```
<pages>
    <page view-id="/hello.jsp" action="#{helloWorld.sayHello}">
        <param name="firstName" value="#{person.firstName}"/>
        <param name="lastName" value="#{person.lastName}"/>
    </page>
<pages>
```

The `<param>` declaration is bidirectional, just like a value binding for a JSF input:

- When a non-faces (GET) request for the view id occurs, Seam sets the value of the named request parameter onto the model object, after performing appropriate type conversions.
- Any `<s:link>` or `<s:button>` transparently includes the request parameter. The value of the parameter is determined by evaluating the value binding during the render phase (when the `<s:link>` is rendered).
- Any navigation rule with a `<redirect/>` to the view id transparently includes the request parameter. The value of the parameter is determined by evaluating the value binding at the end of the invoke application phase.
- The value is transparently propagated with any JSF form submission for the page with the given view id. (This means that view parameters behave like `PAGE`-scoped context variables for faces requests.

The essential idea behind all this is that *however* we get from any other page to `/hello.jsp` (or from `/hello.jsp` back to `/hello.jsp`), the value of the model attribute referred to in the value binding is "remembered", without the need for a conversation (or other server-side state).

This all sounds pretty complex, and you're probably wondering if such an exotic construct is really worth the

effort. Actually, the idea is very natural once you "get it". It is definitely worth taking the time to understand this stuff. Page parameters are the most elegant way to propagate state across a non-faces request. They are especially cool for problems like search screens with bookmarkable results pages, where we would like to be able to write our application code to handle both POST and GET requests with the same code. Page parameters eliminate repetitive listing of request parameters in the view definition and make redirects much easier to code.

Note that you don't need an actual page action method binding to use a page parameter. The following is perfectly valid:

```
<pages>
    <page view-id="/hello.jsp">
        <param name="firstName" value="#{person.firstName}"/>
        <param name="lastName" value="#{person.lastName}"/>
    </page>
<pages>
```

You can even specify a JSF converter:

```
<pages>
    <page view-id="/calculator.jsp" action="#{calculator.calculate}">
        <param name="x" value="#{calculator.lhs}"/>
        <param name="y" value="#{calculator.rhs}"/>
        <param name="op" converterId="com.my.calculator.OperatorConverter" value="#{calculator.op}"/>
    </page>
<pages>
```

```
<pages>
    <page view-id="/calculator.jsp" action="#{calculator.calculate}">
        <param name="x" value="#{calculator.lhs}"/>
        <param name="y" value="#{calculator.rhs}"/>
        <param name="op" converter="#{operatorConverter}" value="#{calculator.op}"/>
    </page>
</pages>
```

Navigation

You can use standard JSF navigation rules defined in `faces-config.xml` in a Seam application. However, JSF navigation rules have a number of annoying limitations:

- It is not possible to specify request parameters to be used when redirecting.
- It is not possible to begin or end conversations from a rule.
- Rules work by evaluating the return value of the action method; it is not possible to evaluate an arbitrary EL expression.

A further problem is that "orchestration" logic gets scattered between `pages.xml` and `faces-config.xml`. It's better to unify this logic into `pages.xml`.

This JSF navigation rule:

```
<navigation-rule>
    <from-view-id>/editDocument.xhtml</from-view-id>

    <navigation-case>
        <from-action>#{documentEditor.update}</from-action>
        <from-outcome>success</from-outcome>
        <to-view-id>/viewDocument.xhtml</to-view-id>
        <redirect/>
    </navigation-case>
```

```
</navigation-rule>
```

Can be rewritten as follows:

```
<page view-id="/editDocument.xhtml">

    <navigation from-action="#{documentEditor.update}">
        <rule if-outcome="success">
            <redirect view-id="/viewDocument.xhtml"/>
        <rule/>
    </navigation-case>

</navigation-rule>
```

But it would be even nicer if we didn't have to pollute our `DocumentEditor` component with string-valued return values (the JSF outcomes). So Seam lets us write:

```
<page view-id="/editDocument.xhtml">

    <navigation from-action="#{documentEditor.update}"
                  evaluate="#{documentEditor.errors.size}">
        <rule if-outcome="0">
            <redirect view-id="/viewDocument.xhtml"/>
        <rule/>
    </navigation-case>

</navigation-rule>
```

Or even:

```
<page view-id="/editDocument.xhtml">

    <navigation from-action="#{documentEditor.update}">
        <rule if="#{documentEditor.errors.empty}">
            <redirect view-id="/viewDocument.xhtml"/>
        <rule/>
    </navigation-case>

</navigation-rule>
```

The first form evaluates a value binding to determine the outcome value to be used by the subsequent rules. The second approach ignores the outcome and evaluates a value binding for each possible rule.

Of course, when an update succeeds, we probably want to end the current conversation. We can do that like this:

```
<page view-id="/editDocument.xhtml">

    <navigation from-action="#{documentEditor.update}">
        <rule if="#{documentEditor.errors.empty}">
            <end-conversation/>
            <redirect view-id="/viewDocument.xhtml"/>
        <rule/>
    </navigation-case>

</navigation-rule>
```

But ending the conversation loses any state associated with the conversation, including the document we are currently interested in! One solution would be to use an immediate render instead of a redirect:

```
<page view-id="/editDocument.xhtml">
```

```
    <navigation from-action="#{documentEditor.update}">
        <rule if="#{documentEditor.errors.empty}">
            <end-conversation/>
            <render view-id="/viewDocument.xhtml"/>
        <rule/>
    </navigation-case>

</navigation-rule>
```

But the correct solution is to pass the document id as a request parameter:

```
<page view-id="/editDocument.xhtml">

    <navigation from-action="#{documentEditor.update}">
        <rule if="#{documentEditor.errors.empty}">
            <end-conversation/>
            <redirect view-id="/viewDocument.xhtml">
                <param name="documentId" value="#{documentEditor.documentId}"/>
            </redirect>
        <rule/>
    </navigation-case>

</navigation-rule>
```

Null outcomes are a special case in JSF. The null outcome is interpreted to mean "redisplay the page". The following navigation rule matches any non-null outcome, but *not* the null outcome:

```
<page view-id="/editDocument.xhtml">

    <navigation from-action="#{documentEditor.update}">
        <rule>
            <render view-id="/viewDocument.xhtml"/>
        <rule/>
    </navigation-case>

</navigation-rule>
```

If you want to perform navigation when a null outcome occurs, use the following form instead:

```
<page view-id="/editDocument.xhtml">

    <navigation from-action="#{documentEditor.update}">
        <render view-id="/viewDocument.xhtml"/>
    </navigation-case>

</navigation-rule>
```

Fine-grained files for definition of page actions and parameters

If you have a lot of different page actions and page parameters, you will almost certainly want to split the declarations up over multiple files. You can define actions and parameters for a page with the view id `/calc/calculator.jsp` in a resource named `calc/calculator.page.xml`. The root element in this case is the `<page>` element, and the view id is implied:

```
<page action="#{calculator.calculate}">
    <param name="x" value="#{calculator.lhs}"/>
    <param name="y" value="#{calculator.rhs}"/>
    <param name="op" converter="#{operatorConverter}" value="#{calculator.op}"/>
</page>
```

5.1.2. Component-driven events

Seam components can interact by simply calling each others methods. Stateful components may even implement the observer/observable pattern. But to enable components to interact in a more loosely-coupled fashion than is possible when the components call each others methods directly, Seam provides *component-driven events*.

We specify event listeners (observers) in `WEB-INF/events.xml`.

```
<events>
    <event type="hello">
        <action expression="#{helloListener.sayHelloBack}"/>
        <action expression="#{logger.logHello}"/>
    </event>
<events>
```

Where the *event type* is just an arbitrary string.

When an event occurs, the actions registered for that event will be called in the order they appear in `events.xml`. How does a component raise an event? Seam provides a built-in component for this.

```
@Name("helloWorld")
public class HelloWorld {
    public void sayHello() {
        FacesMessages.instance().add("Hello World!");
        Events.instance().raiseEvent("hello");
    }
}
```

Notice that this event producer has no dependency upon event consumers. The event listener may now be implemented with absolutely no dependency upon the producer:

```
@Name("helloListener")
public class HelloListener {
    public void sayHelloBack() {
        FacesMessages.instance().add("Hello to you too!");
    }
}
```

If you don't like the `events.xml` file, we can use an annotation instead:

```
@Name("helloListener")
public class HelloListener {
    @Observer("hello")
    public void sayHelloBack() {
        FacesMessages.instance().add("Hello to you too!");
    }
}
```

You might wonder why I've not mentioned anything about event objects in this discussion. In Seam, there is no need for an event object to propagate state between event producer and listener. State is held in the Seam contexts, and is shared between components. However, if you really want to pass an event object, you can:

```
@Name("helloWorld")
public class HelloWorld {
    private String name;
    public void sayHello() {
        FacesMessages.instance().add("Hello World, my name is #0.", name);
        Events.instance().raiseEvent("hello", name);
    }
}
```

```
@Name("helloListener")
```

```
public class HelloListener {
    @Observer("hello")
    public void sayHelloBack(String name) {
        FacesMessages.instance().add("Hello #0!", name);
    }
}
```

5.1.3. Contextual events

Seam defines a number of built-in events that the application can use to perform special kinds of framework integration. The events are:

- `org.jboss.seam.preSetVariable.<name>` — called when the context variable <name> is set
- `org.jboss.seam.postSetVariable.<name>` — called when the context variable <name> is set
- `org.jboss.seam.preRemoveVariable.<name>` — called when the context variable <name> is unset
- `org.jboss.seam.postRemoveVariable.<name>` — called when the context variable <name> is unset
- `org.jboss.seam.preDestroyContext.<SCOPE>` — called before the <SCOPE> context is destroyed
- `org.jboss.seam.postDestroyContext.<SCOPE>` — called after the <SCOPE> context is destroyed
- `org.jboss.seam.beginConversation` — called whenever a long-running conversation begins
- `org.jboss.seam.endConversation` — called whenever a long-running conversation ends
- `org.jboss.seam.beginPageflow.<name>` — called when the pageflow <name> begins
- `org.jboss.seam.endPageflow.<name>` — called when the pageflow <name> ends
- `org.jboss.seam.createProcess.<name>` — called when the process <name> is created
- `org.jboss.seam.endProcess.<name>` — called when the process <name> ends
- `org.jboss.seam.initProcess.<name>` — called when the process <name> is associated with the conversation
- `org.jboss.seam.initTask.<name>` — called when the task <name> is associated with the conversation
- `org.jboss.seam.startTask.<name>` — called when the task <name> is started
- `org.jboss.seam.endTask.<name>` — called when the task <name> is ended
- `org.jboss.seam.postCreate.<name>` — called when the component <name> is created
- `org.jboss.seam.preDestroy.<name>` — called when the component <name> is destroyed
- `org.jboss.seam.beforePhase` — called before the start of a JSF phase
- `org.jboss.seam.afterPhase` — called after the end of a JSF phase

Seam components may observe any of these events in just the same way they observe component-driven events.

5.2. Seam interceptors

EJB 3.0 introduced a standard interceptor model for session bean components. To add an interceptor to a bean, you need to write a class with a method annotated `@AroundInvoke` and annotate the bean with an `@Interceptors` annotation that specifies the name of the interceptor class. For example, the following interceptor checks that the user is logged in before allowing invoking an action listener method:

```
public class LoggedInInterceptor {

   @AroundInvoke
   public Object checkLoggedIn(InvocationContext invocation) throws Exception {

      boolean isLoggedIn = Contexts.getSessionContext().get("loggedIn")!=null;
      if (isLoggedIn) {
         //the user is already logged in
         return invocation.proceed();
      }
      else {
         //the user is not logged in, fwd to login page
         return "login";
      }
   }

}
```

To apply this interceptor to a session bean which acts as an action listener, we must annotate the session bean `@Interceptors(LoggedInInterceptor.class)`. This is a somewhat ugly annotation. Seam builds upon the interceptor framework in EJB3 by allowing you to use `@Interceptors` as a meta-annotation. In our example, we would create an `@LoggedIn` annotation, as follows:

```
@Target(TYPE)
@Retention(RUNTIME)
@Interceptors(LoggedInInterceptor.class)
public @interface LoggedIn {}
```

We can now simply annotate our action listener bean with `@LoggedIn` to apply the interceptor.

```
@Stateless
@Name("changePasswordAction")
@LoggedIn
@Interceptors(SeamInterceptor.class)
public class ChangePasswordAction implements ChangePassword {

    ...

    public String changePassword() { ... }

}
```

If interceptor ordering is important (it usually is), you can add `@Interceptor` annotations to your interceptor classes to specify a partial order of interceptors.

```
@Interceptor(around={BijectionInterceptor.class,
                     ValidationInterceptor.class,
                     ConversationInterceptor.class},
             within=RemoveInterceptor.class)
public class LoggedInInterceptor
{
    ...
}
```

You can even have a "client-side" interceptor, that runs around any of the built-in functionality of EJB3:

```
@Interceptor(type=CLIENT)
public class LoggedInInterceptor
{
    ...
}
```

EJB interceptors are stateful, with a lifecycle that is the same as the component they intercept. For interceptors which do not need to maintain state, Seam lets you get a performance optimization by specifying `@Interceptor(stateless=true)`.

Much of the functionality of Seam is implemented as a set of built-in Seam interceptors, including the interceptors named in the previous example. You don't have to explicitly specify these interceptors by annotating your components; they exist for all interceptable Seam components.

You can even use Seam interceptors with JavaBean components, not just EJB3 beans!

EJB defines interception not only for business methods (using `@AroundInvoke`), but also for the lifecycle methods `@PostConstruct`, `@PreDestroy`, `@PrePassivate` and `@PostActive`. Seam supports all these lifecycle methods on both component and interceptor not only for EJB3 beans, but also for JavaBean components (except `@PreDestroy` which is not meaningful for JavaBean components).

5.3. Managing exceptions

JSF is surprisingly limited when it comes to exception handling. As a partial workaround for this problem, Seam lets you define how a particular class of exception is to be treated by annotating the exception class, or declaring the exception class in an XML file. This facility is meant to be combined with the EJB 3.0-standard `@ApplicationException` annotation which specifies whether the exception should cause a transaction rollback.

Note that Seam applies the EJB 3.0 exception rollback rules also to Seam JavaBean components: *system exceptions* always cause a transaction rollback, *application exceptions* do not cause a rollback by default, but do if `@ApplicationException(rollback=true)` is specified. (An application exception is any checked exception, or any unchecked exception annotated `@ApplicationException`. A system exception is any unchecked exception without an `@ApplicationException` annotation.)

This exception results in a HTTP 404 error whenever it propagates out of the Seam component layer. It does not roll back the current transaction.

```
@HttpError(errorCode=404)
public class ApplicationException extends Exception { ... }
```

This exception results in a browser redirect whenever it propagates out of the Seam component layer. It also ends the current conversation. It also rolls back the current transaction.

```
@Redirect(viewId="/failure.xhtml", end=true)
@ApplicationException(rollback=true)
public class UnrecoverableApplicationException extends RuntimeException { ... }
```

Note that `@Redirect` does not work for exceptions which occur during the render phase of the JSF lifecycle.

This exception results in immediate rendering of the view, along with a message to the user, when it propagates out of the Seam component layer. It also rolls back the current transaction.

```
@Render(viewId="/error.xhtml", message="Unexpected error")
public class SystemException extends RuntimeException { ... }
```

Note that `@Render` only works when the exception occurs during the `INVOKE_APPLICATION` phase.

Since we can't add annotations to all the exception classes we are interested in, Seam also lets us specify this functionality in `WEB-INF/exceptions.xml`.

```
<exceptions>

   <exception class="javax.persistence.EntityNotFoundException">
      <http-error error-code="404"/>
   </exception>

   <exception class="javax.persistence.PersistenceException">
      <render>Database access failed</render>
      <end-conversation/>
   </exception>

   <exception>
      <redirect view-id="/search.xhtml">Unexpected failure</redirect>
      <end-conversation/>
   </exception>

</exceptions>
```

The last `<exception>` declaration does not specify a class, and is a catch-all for any exception for which handling is not otherwise specified via annotations or in `exceptions.xml`.

Chapter 6. Conversations and workspace management

It's time to understand Seam's conversation model in more detail.

Historically, the notion of a Seam "conversation" came about as a merger of three different ideas:

- The idea of a *workspace*, which I encountered in a project for the Victorian government in 2002. In this project I was forced to implement workspace management on top of Struts, an experience I pray never to repeat.
- The idea of an *application transaction* with optimistic semantics, and the realization that existing frameworks based around a stateless architecture could not provide effective management of extended persistence contexts. (The Hibernate team is truly fed up with copping the blame for `LazyInitializationException`s, which are not really Hibernate's fault, but rather the fault of the extremely limiting persistence context model supported by stateless architectures such as the Spring framework or the traditional *stateless session facade* (anti)pattern in J2EE.)
- The idea of a workflow *task*.

By unifying these ideas and providing deep support in the framework, we have a powerful construct that lets us build richer and more efficient applications with less code than before.

6.1. Seam's conversation model

The examples we have seen so far make use of a very simple conversation model that follows these rules:

- There is always a conversation context active during the apply request values, process validations, update model values, invoke application and render response phases of the JSF request lifecycle.
- At the end of the restore view phase of the JSF request lifecycle, Seam attempts to restore any previous long-running conversation context. If none exists, Seam creates a new temporary conversation context.
- When an `@Begin` method is encountered, the temporary conversation context is promoted to a long running conversation.
- When an `@End` method is encountered, any long-running conversation context is demoted to a temporary conversation.
- At the end of the render response phase of the JSF request lifecycle, Seam stores the contents of a long running conversation context or destroys the contents of a temporary conversation context.
- Any faces request (a JSF postback) will propagate the conversation context. By default, non-faces requests (GET requests, for example) do not propagate the conversation context, but see below for more information on this.
- If the JSF request lifecycle is foreshortened by a redirect, Seam transparently stores and restores the current conversation context—unless the conversation was already ended via `@End(beforeRedirect=true)`.

Seam transparently propagates the conversation context across JSF postbacks and redirects. If you don't do anything special, a *non-faces request* (a GET request for example) will not propagate the conversation context and

will be processed in a new temporary conversation. This is usually - but not always - the desired behavior.

If you want to propagate a Seam conversation across a non-faces request, you need to explicitly code the Seam *conversation id* as a request parameter:

```
<a href="main.jsf?conversationId=#{conversation.id}">Continue</a>
```

Or, the more JSF-ish:

```
<h:outputLink value="main.jsf">
    <f:param name="conversationId" value="#{conversation.id}"/>
    <h:outputText value="Continue"/>
</h:outputLink>
```

If you use the Seam tag library, this is equivalent:

```
<h:outputLink value="main.jsf">
    <s:conversationId/>
    <h:outputText value="Continue"/>
</h:outputLink>
```

If you wish to disable propagation of the conversation context for a postback, a similar trick is used:

```
<h:commandLink action="main" value="Exit">
    <f:param name="conversationPropagation" value="none"/>
</h:commandLink>
```

If you use the Seam tag library, this is equivalent:

```
<h:commandLink action="main" value="Exit">
    <s:conversationPropagation type="none"/>
</h:commandLink>
```

Note that disabling conversation context propagation is absolutely not the same thing as ending the conversation.

The `conversationPropagation` request parameter, or the `<s:conversationPropagation>` tag may even be used to begin and end conversation, or begin a nested conversation.

```
<h:commandLink action="main" value="Exit">
    <s:conversationPropagation type="end"/>
</h:commandLink>
```

```
<h:commandLink action="main" value="Select Child">
    <s:conversationPropagation type="nested"/>
</h:commandLink>
```

```
<h:commandLink action="main" value="Select Hotel">
    <s:conversationPropagation type="begin"/>
</h:commandLink>
```

```
<h:commandLink action="main" value="Select Hotel">
    <s:conversationPropagation type="join"/>
</h:commandLink>
```

This conversation model makes it easy to build applications which behave correctly with respect to multi-window operation. For many applications, this is all that is needed. Some complex applications have either or both of the following additional requirements:

- A conversation spans many smaller units of user interaction, which execute serially or even concurrently. The smaller *nested conversations* have their own isolated set of conversation state, and also have access to the state of the outer conversation.
- The user is able to switch between many conversations within the same browser window. This feature is called *workspace management*.

6.2. Nested conversations

A nested conversation is created by invoking a method marked `@Begin(nested=true)` inside the scope of an existing conversation. A nested conversation has its own conversation context, and also has read-only access to the context of the outer conversation. (It can read the outer conversation's context variables, but not write to them.) When an `@End` is subsequently encountered, the nested conversation will be destroyed, and the outer conversation will resume, by "popping" the conversation stack. Conversations may be nested to any arbitrary depth.

Certain user activity (workspace management, or the back button) can cause the outer conversation to be resumed before the inner conversation is ended. In this case it is possible to have multiple concurrent nested conversations belonging to the same outer conversation. If the outer conversation ends before a nested conversation ends, Seam destroys all nested conversation contexts along with the outer context.

A conversation may be thought of as a *continuable state*. Nested conversations allow the application to capture a consistent continuable state at various points in a user interaction, thus insuring truly correct behavior in the face of backbuttoning and workspace management.

TODO: an example to show how a nested conversation prevents bad stuff happening when you backbutton.

6.3. Starting conversations with GET requests

JSF does not define any kind of action listener that is triggered when a page is accessed via a non-faces request (for example, a HTTP GET request). This can occur if the user bookmarks the page, or if we navigate to the page via an `<h:outputLink>`.

Sometimes we want to begin a conversation immediately the page is accessed. Since there is no JSF action method, we can't solve the problem in the usual way, by annotating the action with `@Begin`.

A further problem arises if the page needs some state to be fetched into a context variable. We've already seen two ways to solve this problem. If that state is held in a Seam component, we can fetch the state in a `@Create` method. If not, we can define a `@Factory` method for the context variable.

If none of these options works for you, Seam lets you define a *page action* in the `pages.xml` file.

```
<pages>
    <page view-id="/messageList.jsp" action="#{messageManager.list}"/>
    ...
</pages>
```

This action method is called at the beginning of the render response phase, any time the page is about to be rendered. If a page action returns a non-null outcome, Seam will process any appropriate JSF and Seam navigation rules, possibly resulting in a completely different page being rendered.

If *all* you want to do before rendering the page is begin a conversation, you could use a built-in action method

that does just that:

```
<pages>
    <page view-id="/messageList.jsp" action="#{conversation.begin}"/>
    ...
</pages>
```

Note that you can also call this built-in action from a JSF control, and, similarly, you can use `#{conversation.end}` to end conversations.

If you want more control, to join existing conversations or begin a nested conversion, to begin a pageflow or an atomic conversation, you should use the `<begin-conversation>` element.

```
<pages>
    <page view-id="/messageList.jsp">
       <begin-conversation nested="true" pageflow="AddItem"/>
    <page>
    ...
</pages>
```

There is also an `<end-conversation>` element.

```
<pages>
    <page view-id="/home.jsp">
       <end-conversation/>
    <page>
    ...
</pages>
```

To solve the first problem, we now have five options:

- Annotate the `@Create` method with `@Begin`
- Annotate the `@Factory` method with `@Begin`
- Annotate the Seam page action method with `@Begin`
- Use `<begin-conversation>` in `pages.xml`.
- Use `#{conversation.begin}` as the Seam page action method

6.4. Using `<s:link>` and `<s:button>`

JSF command links always perform a form submission via JavaScript, which breaks the web browser's "open in new window" or "open in new tab" feature. In plain JSF, you need to use an `<h:outputLink>` if you need this functionality. But there are two major limitations to `<h:outputLink>`.

- JSF provides no way to attach an action listener to an `<h:outputLink>`.
- JSF does not propagate the selected row of a `DataModel` since there is no actual form submission.

Seam provides the notion of a *page action* to help solve the first problem, but this does nothing to help us with the second problem. We *could* work around this by using the RESTful approach of passing a request parameter and requerying for the selected object on the server side. In some cases—such as the Seam blog example application—this is indeed the best approach. The RESTful style supports bookmarking, since it does not require server-side state. In other cases, where we don't care about bookmarks, the use of `@DataModel` and

`@DataModelSelection` is just so convenient and transparent!

To fill in this missing functionality, and to make conversation propagation even simpler to manage, Seam provides the `<s:link>` JSF tag.

The link may specify just the JSF view id:

```
<s:link view="/login.xhtml" value="Login"/>
```

Or, it may specify an action method (in which case the action outcome determines the page that results):

```
<s:link action="#{login.logout}" value="Logout"/>
```

If you specify *both* a JSF view id and an action method, the 'view' will be used *unless* the action method returns a non-null outcome:

```
<s:link view="/loggedOut.xhtml"  action="#{login.logout}" value="Logout"/>
```

The link automatically propagates the selected row of a `DataModel` using inside `<h:dataTable>`:

```
<s:link view="/hotel.xhtml" action="#{hotelSearch.selectHotel}" value="#{hotel.name}"/>
```

You can leave the scope of an existing conversation:

```
<s:link view="/main.xhtml" propagation="none"/>
```

You can begin, end, or nest conversations:

```
<s:link action="#{issueEditor.viewComment}" propagation="nest"/>
```

If the link begins a conversation, you can even specify a pageflow to be used:

```
<s:link action="#{documentEditor.getDocument}" propagation="begin"
        pageflow="EditDocument"/>
```

The `taskInstance` attribute if for use in jBPM task lists:

```
<s:link action="#{documentApproval.approveOrReject}" taskInstance="#{task}"/>
```

(See the DVD Store demo application for examples of this.)

Finally, if you need the "link" to be rendered as a button, use `<s:button>`:

```
<s:button action="#{login.logout}" value="Logout"/>
```

6.5. Success messages

It is quite common to display a message to the user indicating success or failure of an action. It is convenient to use a JSF `FacesMessage` for this. Unfortunately, a successful action often requires a browser redirect, and JSF does not propagate faces messages across redirects. This makes it quite difficult to display success messages in plain JSF.

The built in conversation-scoped Seam component named `facesMessages` solves this problem. (You must have the Seam redirect filter installed.)

```
@Name("editDocumentAction")
@Stateless
public class EditDocumentBean implements EditDocument {
    @In EntityManager em;
    @In Document document;
    @In FacesMessages facesMessages;

    public String update() {
        em.merge(document);
        facesMessages.add("Document updated");
    }
}
```

Any message added to `facesMessages` is used in the very next render response phase for the current conversation. This even works when there is no long-running conversation since Seam preserves even temporary conversation contexts across redirects.

You can even include JSF EL expressions in a faces message summary:

```
facesMessages.add("Document #{document.title} was updated");
```

You may display the messages in the usual way, for example:

```
<h:messages globalOnly="true"/>
```

6.6. Using an "explicit" conversation id

Ordinarily, Seam generates a meaningless unique id for each conversation in each session. You can customize the id value when you begin the conversation.

This feature can be used to customize the conversation id generation algorithm like so:

```
@Begin(id="#{myConversationIdGenerator.nextId}")
public void editHotel() { ... }
```

Or it can be used to assign a meaningful conversation id:

```
@Begin(id="hotel#{hotel.id}")
public String editHotel() { ... }
```

```
@Begin(id="hotel#{hotelsDataModel.rowData.id}")
public String selectHotel() { ... }
```

```
@Begin(id="entry#{params['blogId']}")
public String viewBlogEntry() { ... }
```

```
@BeginTask(id="task#{taskInstance.id}")
public String approveDocument() { ... }
```

Clearly, these example result in the same conversation id every time a particular hotel, blog or task is selected. So what happens if a conversation with the same conversation id already exists when the new conversation begins? Well, Seam detects the existing conversation and redirects to that conversation without running the `@Begin` method again. This feature helps control the number of workspaces that are created when using workspace management.

6.7. Workspace management

Workspace management is the ability to "switch" conversations in a single window. Seam makes workspace management completely transparent at the level of the Java code. To enable workspace management, all you need to do is:

- Provide *description* text for each view id (when using JSF or Seam navigation rules) or page node (when using jPDL pageflows). This description text is displayed to the user by the workspace switchers.
- Include one or more of the standard workspace switcher JSP or facelets fragments in your pages. The standard fragments support workspace management via a drop down menu, a list of conversations, or breadcrumbs.

6.7.1. Workspace management and JSF navigation

When you use JSF or Seam navigation rules, Seam switches to a conversation by restoring the current `view-id` for that conversation. The descriptive text for the workspace is defined in a file called `pages.xml` that Seam expects to find in the `WEB-INF` directory, right next to `faces-config.xml`:

```
<pages>
    <page view-id="/main.xhtml">Search hotels: #{hotelBooking.searchString}</page>
    <page view-id="/hotel.xhtml">View hotel: #{hotel.name}</page>
    <page view-id="/book.xhtml">Book hotel: #{hotel.name}</page>
    <page view-id="/confirm.xhtml">Confirm: #{booking.description}</page>
</pages>
```

Note that if this file is missing, the Seam application will continue to work perfectly! The only missing functionality will be the ability to switch workspaces.

6.7.2. Workspace management and jPDL pageflow

When you use a jPDL pageflow definition, Seam switches to a conversation by restoring the current jBPM process state. This is a more flexible model since it allows the same `view-id` to have different descriptions depending upon the current `<page>` node. The description text is defined by the `<page>` node:

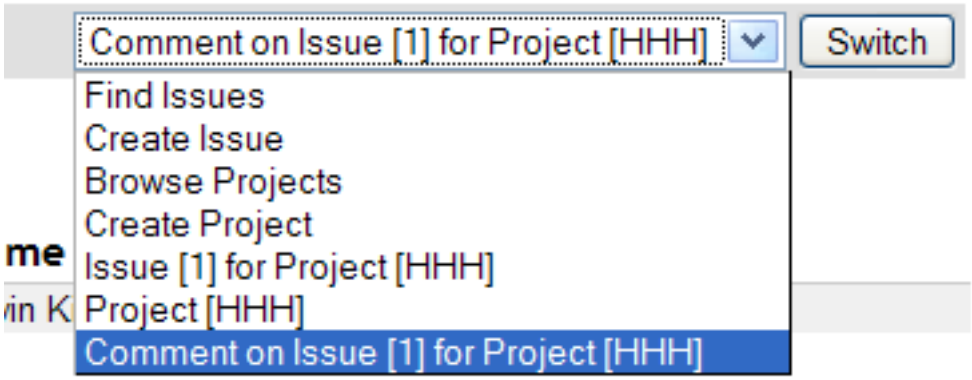
```
<pageflow-definition name="shopping">

   <start-state name="start">
      <transition to="browse"/>
   </start-state>

   <page name="browse" view-id="/browse.xhtml">
      <description>DVD Search: #{search.searchPattern}</description>
      <transition to="browse"/>
      <transition name="checkout" to="checkout"/>
   </page>

   <page name="checkout" view-id="/checkout.xhtml">
      <description>Purchase: $#{cart.total}</description>
      <transition to="checkout"/>
      <transition name="complete" to="complete"/>
   </page>

   <page name="complete" view-id="/complete.xhtml">
      <end-conversation />
   </page>

</pageflow-definition>
```

6.7.3. The conversation switcher

Include the following fragment in your JSP or facelets page to get a drop-down menu that lets you switch to any current conversation, or to any other page of the application:

```
<h:selectOneMenu value="#{switcher.conversationIdOrOutcome}">
    <f:selectItem itemLabel="Find Issues" itemValue="findIssue"/>
    <f:selectItem itemLabel="Create Issue" itemValue="editIssue"/>
    <f:selectItems value="#{switcher.selectItems}"/>
</h:selectOneMenu>
<h:commandButton action="#{switcher.select}" value="Switch"/>
```

In this example, we have a menu that includes an item for each conversation, together with two additional items that let the user begin a new conversation.

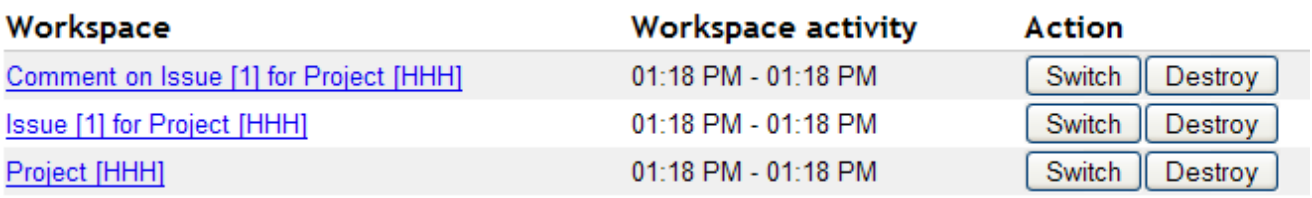


6.7.4. The conversation list

The conversation list is very similar to the conversation switcher, except that it is displayed as a table:

```
<h:dataTable value="#{conversationList}" var="entry"
        rendered="#{not empty conversationList}">
    <h:column>
        <f:facet name="header">Workspace</f:facet>
        <h:commandLink action="#{entry.select}" value="#{entry.description}"/>
        <h:outputText value="[current]" rendered="#{entry.current}"/>
    </h:column>
    <h:column>
        <f:facet name="header">Activity</f:facet>
        <h:outputText value="#{entry.startDatetime}">
            <f:convertDateTime type="time" pattern="hh:mm a"/>
        </h:outputText>
        <h:outputText value=" - "/>
        <h:outputText value="#{entry.lastDatetime}">
            <f:convertDateTime type="time" pattern="hh:mm a"/>
        </h:outputText>
    </h:column>
    <h:column>
        <f:facet name="header">Action</f:facet>
        <h:commandButton action="#{entry.select}" value="#{msg.Switch}"/>
        <h:commandButton action="#{entry.destroy}" value="#{msg.Destroy}"/>
    </h:column>
</h:dataTable>
```

We imagine that you will want to customize this for your own application.



The conversation list is nice, but it takes up a lot of space on the page, so you probably don't want to put it on *every* page.

Notice that the conversation list lets the user destroy workspaces.

6.7.5. Breadcrumbs

Breadcrumbs are useful in applications which use a nested conversation model. The breadcrumbs are a list of links to conversations in the current conversation stack:

```
<t:dataList value="#{conversationStack}" var="entry">
    <h:outputText value=" | "/>
    <h:commandLink value="#{entry.description}" action="#{entry.select}"/>
</t:dataList>
```

Notice that here we are using the MyFaces `<t:dataList>` component, since JSF amazingly does not provide any standard component for looping.

Please refer to the Seam Issue Tracker demo to see all this functionality in action!

6.8. Seam-managed persistence contexts and atomic conversations

Seam provides built-in components for EJB 3.0 and Hibernate persistence context management that support the use of persistence contexts scoped to the conversation. This useful feature allows you to program optimistic transactions that span multiple requests to the server without the need to use the merge() operation or to re-load data at the beginning of each request, and without the need to wrestle with the dreaded `LazyInitializationException` or `NonUniqueObjectException`. Please see the configuration chapter for information about configuring Seam-managed persistence contexts and Seam-managed transactions.

As with any optimistic transaction management, transaction isolation and consistency can be achieved via use of optimistic locking. Fortunately, both Hibernate and EJB 3.0 make it very easy to use optimistic locking, by providing the `@Version` annotation.

By default, the persistence context is flushed (synchronized with the database) at the end of each transaction. This is sometimes the desired behavior. But very often, we would prefer that all changes are held in memory and only written to the database when the conversation ends successfully. This allows for truly atomic conversations. As the result of a truly stupid and shortsighted decision by certain non-JBoss, non-Sun and non-Sybase members of the EJB 3.0 expert group, there is currently no simple, usable and portable way to implement atomic conversations using EJB 3.0 persistence. However, Hibernate provides this feature as a vendor extension to the `FlushModeTypes` defined by the specification, and it is our expectation that other vendors will soon provide

a similar extension.

Seam lets you specify `FlushModeType.MANUAL` when beginning a conversation. Currently, this works only when Hibernate is the underlying persistence provider, but we plan to support other equivalent vendor extensions.

```
@In EntityManager em; //a Seam-managed persistence context

@Begin(flushMode=MANUAL)
public void beginClaimWizard() {
    claim = em.find(Claim.class, claimId);
}
```

Now, the `claim` object remains managed by the persistence context for the rest ot the conversation. We can make changes to the claim:

```
public void addPartyToClaim() {
    Party party = ....;
    claim.addParty(party);
}
```

But these changes will not be flushed to the database until we explicitly force the flush to occur:

```
@End
public void commitClaim() {
    em.flush();
}
```

6.9. Seam and Servlets

Requests sent direct to some servlet other than the JSF servlet are not processed through the JSF lifecycle, so Seam provides a servlet filter that can be applied to any servlet that needs access to Seam components.

```
<filter>
    <filter-name>Seam Servlet Filter</filter-name>
    <filter-class>org.jboss.seam.servlet.SeamServletFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>Seam Servlet Filter</filter-name>
    <url-pattern>*.ajax</url-pattern>
</filter-mapping>
```

This servlet filter is responsible for initializing all Seam contexts before passing control to the servlet. It expects to find the conversation id of any conversation context in a request parameter named `conversationId`. You are responsible for ensuring that it gets sent in the request.

You are also responsible for ensuring propagation of any new conversation id back to the client. Seam exposes the conversation id as a property of the built in component `conversation`.

Seam also provides the Seam Remoting framework, a simple way to expose any method of a Seam component for invocation by an asynchronous JavaScript request simply by annotating the methods that should be accessible in the client. See the Seam Remoting chapter for further information.

6.10. Seam and SOAP

TODO

Chapter 7. Pageflows and business processes

JBoss jBPM is a business process management engine for any Java SE or EE environment. jBPM lets you represent a business process or user interaction as a graph of nodes representing wait states, decisions, tasks, web pages, etc. The graph is defined using a simple, very readable, XML dialect called jPDL, and may be edited and visualised graphically using an eclipse plugin. jPDL is an extensible language, and is suitable for a range of problems, from defining web application page flow, to traditional workflow management, all the way up to orchestration of services in a SOA environment.

Seam applications use jBPM for two different problems:

- Defining the pageflow involved in complex user interactions. A jPDL process definition defines the page flow for a single conversation. A Seam conversation is considered to be a relatively short-running interaction with a single user.
- Defining the overarching business process. The business process may span multiple conversations with multiple users. Its state is persistent in the jBPM database, so it is considered long-running. Coordination of the activities of multiple users is a much more complex problem than scripting an interaction with a single user, so jBPM offers sophisticated facilities for task management and dealing with multiple concurrent paths of execution.

Don't get these two things confused ! They operate at very different levels or granularity. *Pageflow*, *conversation* and *task* all refer to a single interaction with a single user. A business process spans many tasks. Futhermore, the two applications of jBPM are totally orthogonal. You can use them together or independently or not at all.

You don't have to know jDPL to use Seam. If you're perfectly happy defining pageflow using JSF or Seam navigation rules, and if your application is more data-driven that process-driven, you probably don't need jBPM. But we're finding that thinking of user interaction in terms of a well-defined graphical representation is helping us build more robust applications.

7.1. Pageflow in Seam

There are two ways to define pageflow in Seam:

- Using JSF or Seam navigation rules - the *stateless navigation model*
- Using jPDL - the *stateful navigation model*

Very simple applications will only need the stateless navigation model. Very complex applications will use both models in different places. Each model has its strengths and weaknesses!

7.1.1. The two navigation models

The stateless model defines a mapping from a set of named, logical outcomes of an event directly to the resulting page of the view. The navigation rules are entirely oblivious to any state held by the application other than what page was the source of the event. This means that your action listener methods must sometimes make decisions about the page flow, since only they have access to the current state of the application.

Here is an example page flow definition using JSF navigation rules:

```
<navigation-rule>
    <from-view-id>/numberGuess.jsp</from-view-id>

    <navigation-case>
        <from-outcome>guess</from-outcome>
        <to-view-id>/numberGuess.jsp</to-view-id>
        <redirect/>
    </navigation-case>

    <navigation-case>
        <from-outcome>win</from-outcome>
        <to-view-id>/win.jsp</to-view-id>
        <redirect/>
    </navigation-case>

    <navigation-case>
        <from-outcome>lose</from-outcome>
        <to-view-id>/lose.jsp</to-view-id>
        <redirect/>
    </navigation-case>

</navigation-rule>
```

Here is the same example page flow definition using Seam navigation rules:

```
<page view-id="/numberGuess.jsp">

    <navigation>
        <rule if-outcome="guess">
            <redirect view-id="/numberGuess.jsp"/>
        </rule>
        <rule if-outcome="win">
            <redirect view-id="/win.jsp"/>
        </rule>
        <rule if-outcome="lose">
            <redirect view-id="/lose.jsp"/>
        </rule>
    </navigation-case>

</navigation-rule>
```

If you find navigation rules overly verbose, you can return view ids directly from your action listener methods:

```
public String guess() {
    if (guess==randomNumber) return "/win.jsp";
    if (++guessCount==maxGuesses) return "/lose.jsp";
    return null;
}
```

Note that this results in a redirect. You can even specify parameters to be used in the redirect:

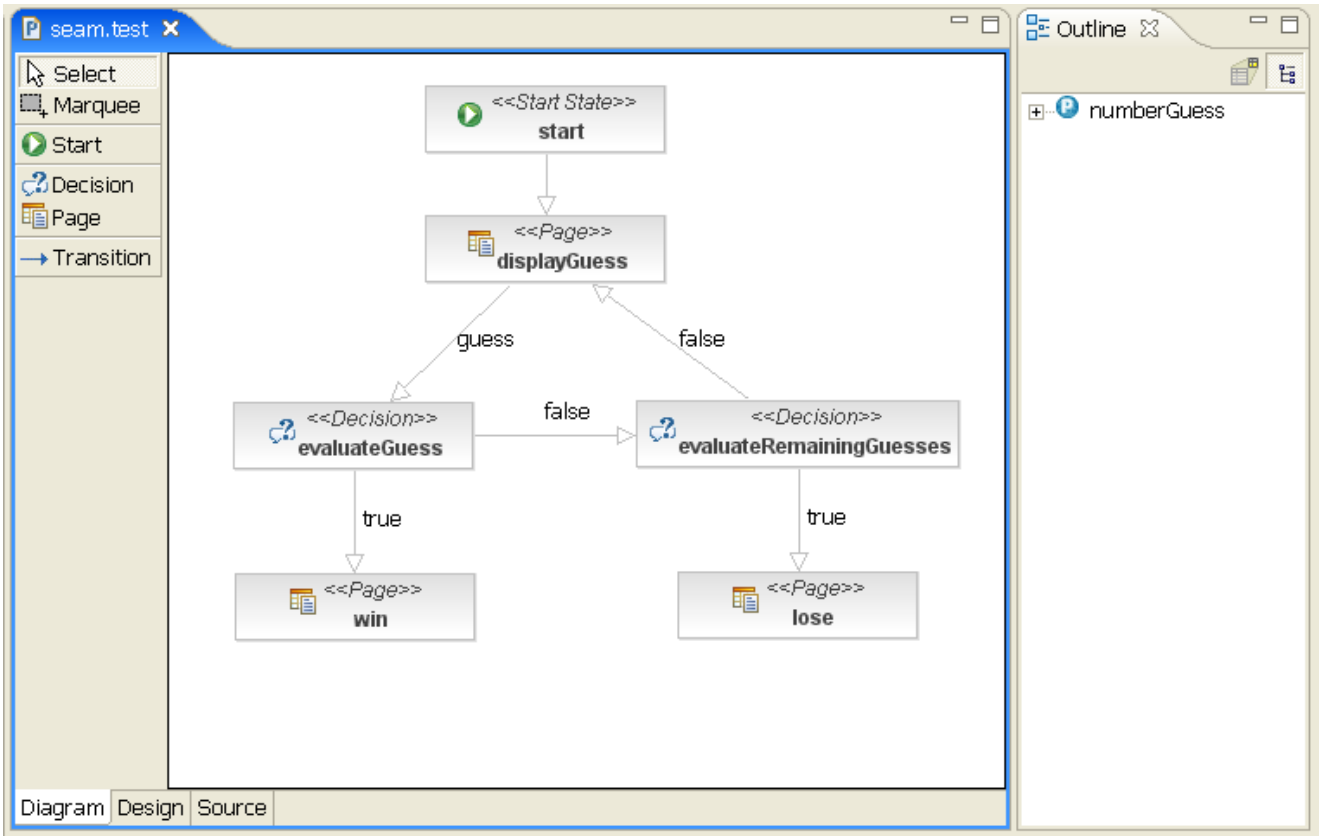
```
public String search() {
    return "/searchResults.jsp?searchPattern=#{searchAction.searchPattern}";
}
```

The stateful model defines a set of transitions between a set of named, logical application states. In this model, it is possible to express the flow of any user interaction entirely in the jPDL pageflow definition, and write action listener methods that are completely unaware of the flow of the interaction.

Here is an example page flow definition using jPDL:

```
<pageflow-definition name="numberGuess">

   <start-page name="displayGuess" view-id="/numberGuess.jsp">
      <redirect/>
```

```
      <transition name="guess" to="evaluateGuess">
              <action expression="#{numberGuess.guess}" />
      </transition>
   </start-page>

   <decision name="evaluateGuess" expression="#{numberGuess.correctGuess}">
      <transition name="true" to="win"/>
      <transition name="false" to="evaluateRemainingGuesses"/>
   </decision>

   <decision name="evaluateRemainingGuesses" expression="#{numberGuess.lastGuess}">
      <transition name="true" to="lose"/>
      <transition name="false" to="displayGuess"/>
   </decision>

   <page name="win" view-id="/win.jsp">
      <redirect/>
      <end-conversation />
   </page>

   <page name="lose" view-id="/lose.jsp">
      <redirect/>
      <end-conversation />
   </page>

</pageflow-definition>
```



There are two things we notice immediately here:

- The JSF/Seam navigation rules are *much* simpler. (However, this obscures the fact that the underlying Java code is more complex.)
- The jPDL makes the user interaction immediately understandable, without us needing to even look at the JSP or Java code.

In addition, the stateful model is more *constrained*. For each logical state (each step in the page flow), there are

a constrained set of possible transitions to other states. The stateless model is an *ad hoc* model which is suitable to relatively unconstrained, freeform navigation where the user decides where he/she wants to go next, not the application.

The stateful/stateless navigation distinction is quite similar to the traditional view of modal/modeless interaction. Now, Seam applications are not usually modal in the simple sense of the word - indeed, avoiding application modal behavior is one of the main reasons for having conversations! However, Seam applications can be, and often are, modal at the level of a particular conversation. It is well-known that modal behavior is something to avoid as much as possible; it is very difficult to predict the order in which your users are going to want to do things! However, there is no doubt that the stateful model has its place.

The biggest contrast between the two models is the back-button behavior.

7.1.2. Seam and the back button

When JSF or Seam navigation rules are used, Seam lets the user freely navigate via the back, forward and refresh buttons. It is the responsibility of the application to ensure that conversational state remains internally consistent when this occurs. Experience with the combination of web application frameworks like Struts or WebWork - that do not support a conversational model - and stateless component models like EJB stateless session beans or the Spring framework has taught many developers that this is close to impossible to do! However, our experience is that in the context of Seam, where there is a well-defined conversational model, backed by stateful session beans, it is actually quite straightforward. Usually it is as simple as combining the use of `no-conversation-view-id` with null checks at the beginning of action listener methods. We consider support for freeform navigation to be almost always desirable.

In this case, the `no-conversation-view-id` declaration goes in `pages.xml`. It tells Seam to redirect to a different page if a request originates from a page rendered during a conversation, and that conversation no longer exists:

```
<page view-id="/checkout.xhtml"
        no-conversation-view-id="/main.xhtml"/>
```

On the other hand, in the stateful model, backbuttoning is interpreted as an undefined transition back to a previous state. Since the stateful model enforces a defined set of transitions from the current state, back buttoning is be default disallowed in the stateful model! Seam transparently detects the use of the back button, and blocks any attempt to perform an action from a previous, "stale" page, and simply redirects the user to the "current" page (and displays a faces message). Whether you consider this a feature or a limitation of the stateful model depends upon your point of view: as an application developer, it is a feature; as a user, it might be frustrating! You can enable backbutton navigation from a particular page node by setting `back="enabled"`.

```
<page name="checkout"
        view-id="/checkout.xhtml"
        back="enabled">
    <redirect/>
    <transition to="checkout"/>
    <transition name="complete" to="complete"/>
</page>
```

This allows backbuttoning *from* the `checkout` state to *any previous state!*

Of course, we still need to define what happens if a request originates from a page rendered during a pageflow, and the conversation with the pageflow no longer exists. In this case, the `no-conversation-view-id` declaration goes into the pageflow definition:

```
<page name="checkout"
        view-id="/checkout.xhtml"
```

```
        back="enabled"
        no-conversation-view-id="/main.xhtml">
    <redirect/>
    <transition to="checkout"/>
    <transition name="complete" to="complete"/>
</page>
```

In practice, both navigation models have their place, and you'll quickly learn to recognize when to prefer one model over the other.

7.2. Using jPDL pageflows

7.2.1. Installing pageflows

We need to install the Seam jBPM-related components, and tell them where to find our pageflow definition. We can specify this Seam configuration in `components.xml`.

```
<component class="org.jboss.seam.core.Jbpm">
    <property name="pageflowDefinitions">pageflow.jpdl.xml</property>
</component>
```

The first line installs jBPM, the second points to a jPDL-based pageflow definition.

7.2.2. Starting pageflows

We "start" a jPDL-based pageflow by specifying the name of the process definition using a `@Begin`, `@BeginTask` or `@StartTask` annotation:

```
@Begin(pageflow="numberguess")
public void begin() { ... }
```

If we are beginning the pageflow during the `RENDER_RESPONSE` phase—during a `@Factory` or `@Create` method, for example—we consider ourselves to be already at the page being rendered, and use a `<start-page>` node as the first node in the pageflow, as in the example above.

But if the pageflow is begun as the result of an action listener invocation, the outcome of the action listener determines which is the first page to be rendered. In this case, we use a `<start-state>` as the first node in the pageflow, and declare a transition for each possible outcome:

```
<pageflow-definition name="viewEditDocument">

    <start-state name="start">
        <transition name="documentFound" to="displayDocument"/>
        <transition name="documentNotFound" to="notFound"/>
    </start-state>

    <page name="displayDocument" view-id="/document.jsp">
        <transition name="edit" to="editDocument"/>
        <transition name="done" to="main"/>
    </page>

    ...

    <page name="notFound" view-id="/404.jsp">
        <end-conversation/>
    </page>
```

```
</pageflow-definition>
```

7.2.3. Page nodes and transitions

Each <page> node represents a state where the system is waiting for user input:

```
<page name="displayGuess" view-id="/numberGuess.jsp">
    <redirect/>
    <transition name="guess" to="evaluateGuess">
        <action expression="#{numberGuess.guess}" />
    </transition>
</page>
```

The view-id is the JSF view id. The <redirect/> element has the same effect as <redirect/> in a JSF navigation rule: namely, a post-then-redirect behavior, to overcome problems with the browser's refresh button. (Note that Seam propagates conversation contexts over these browser redirects. So there is no need for a Ruby on Rails style "flash" construct in Seam!)

The transition name is the name of a JSF outcome triggered by clicking a command button or command link in numberGuess.jsp.

```
<h:commandButton type="submit" value="Guess" action="guess"/>
```

When the transition is triggered by clicking this button, jBPM will activate the transition action by calling the guess() method of the numberGuess component. Notice that the syntax used for specifying actions in the jPDL is just a familiar JSF EL expression, and that the transition action handler is just a method of a Seam component in the current Seam contexts. So we have exactly the same event model for jBPM events that we already have for JSF events! (The *One Kind of Stuff* principle.)

In the case of a null outcome (for example, a command button with no action defined), Seam will signal the transition with no name if one exists, or else simply redisplay the page if all transitions have names. So we could slightly simplify our example pageflow and this button:

```
<h:commandButton type="submit" value="Guess"/>
```

Would fire the following un-named transition:

```
<page name="displayGuess" view-id="/numberGuess.jsp">
    <redirect/>
    <transition to="evaluateGuess">
        <action expression="#{numberGuess.guess}" />
    </transition>
</page>
```

It is even possible to have the button call an action method, in which case the action outcome will determine the transition to be taken:

```
<h:commandButton type="submit" value="Guess" action="#{numberGuess.guess}"/>
```

```
<page name="displayGuess" view-id="/numberGuess.jsp">
    <transition name="correctGuess" to="win"/>
    <transition name="incorrectGuess" to="evaluateGuess"/>
</page>
```

However, this is considered an inferior style, since it moves responsibility for controlling the flow out of the pageflow definition and back into the other components. It is much better to centralize this concern in the page-

flow itself.

7.2.4. Controlling the flow

Usually, we don't need the more powerful features of jPDL when defining pageflows. We do need the `<decision>` node, however:

```
<decision name="evaluateGuess" expression="#{numberGuess.correctGuess}">
    <transition name="true" to="win"/>
    <transition name="false" to="evaluateRemainingGuesses"/>
</decision>
```

A decision is made by evaluating a JSF EL expression in the Seam contexts.

7.2.5. Ending the flow

We end the conversation using `<end-conversation>` or `@End`. (In fact, for readability, use of *both* is encouraged.)

```
<page name="win" view-id="/win.jsp">
    <redirect/>
    <end-conversation/>
</page>
```

Optionally, we can end a task, specify a jBPM `transition` name. In this case, Seam will signal the end of the current task in the overarching business process.

```
<page name="win" view-id="/win.jsp">
    <redirect/>
    <end-task transition="success"/>
</page>
```

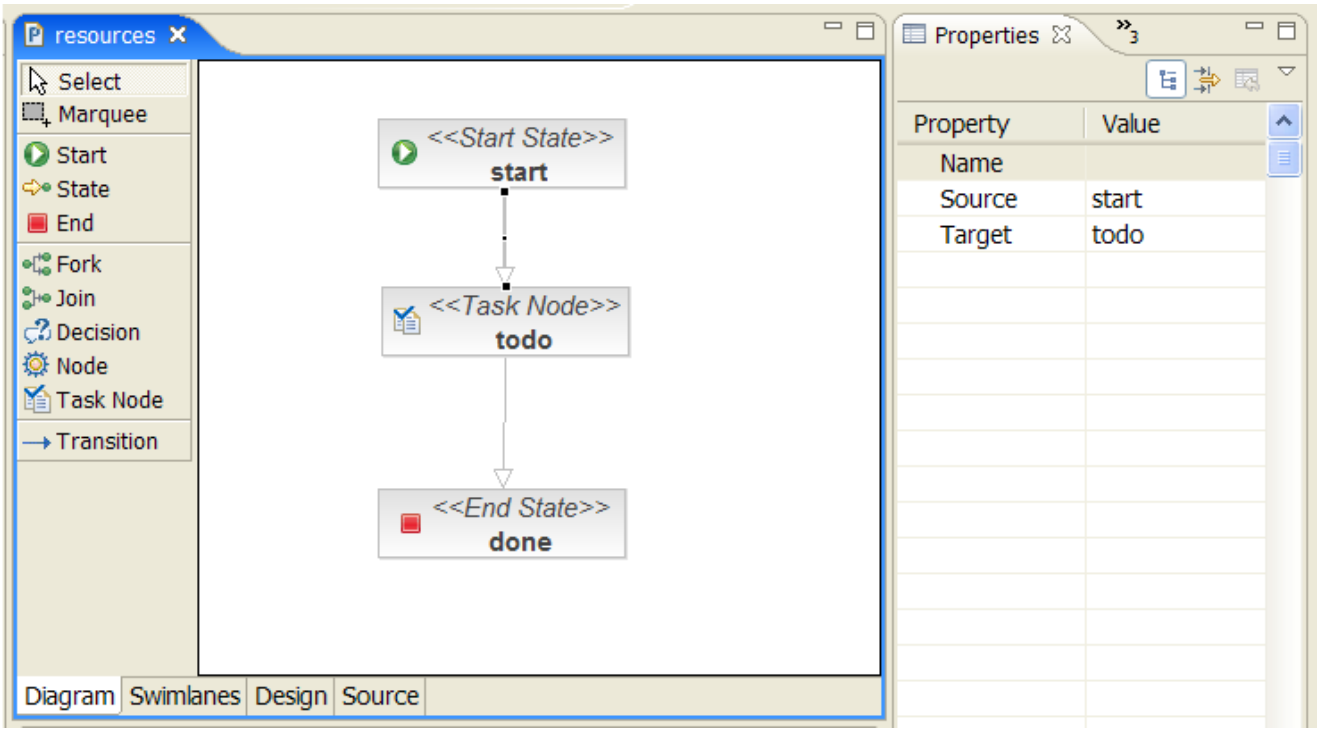
7.3. Business process management in Seam

A business process is a well-defined set of tasks that must be performed by users or software systems according to well-defined rules about *who* can perform a task, and *when* it should be performed. Seam's jBPM integration makes it easy to display lists of tasks to users and let them manage their tasks. Seam also lets the application store state associated with the business process in the `BUSINESS_PROCESS` context, and have that state made persistent via jBPM variables.

A simple business process definition looks much the same as a page flow definition (*One Kind of Stuff*), except that instead of `<page>` nodes, we have `<task-node>` nodes. In a long-running business process, the wait states are where the system is waiting for some user to log in and perform a task.

```
<process-definition name="todo">

   <start-state name="start">
      <transition to="todo"/>
   </start-state>

   <task-node name="todo">
      <task name="todo" description="#{todoList.description}">
         <assignment actor-id="#{actor.id}"/>
      </task>
      <transition to="done"/>
   </task-node>
```

```
   <end-state name="done"/>

</process-definition>
```



It is perfectly possible that we might have both jPDL business process definitions and jPDL pageflow definitions in the same project. If so, the relationship between the two is that a single `<task>` in a business process corresponds to a whole pageflow `<pageflow-definition>`

7.4. Using jPDL business process definitions

7.4.1. Installing process definitions

We need to install jBPM, and tell it where to find the business process definitions:

```
<component class="org.jboss.seam.core.Jbpm">
    <property name="processDefinitions">todo.jpdl.xml</property>
</component>
```

7.4.2. Initializing actor ids

We always need to know what user is currently logged in. jBPM "knows" users by their *actor id* and *group actor ids*. We specify the current actor ids using the built in Seam component named `actor`:

```
@In Actor actor;

public String login() {
    ...
    actor.setId( user.getUserName() );
    actor.getGroupActorIds().addAll( user.getGroupNames() );
    ...
}
```

7.4.3. Initiating a business process

To initiate a business process instance, we use the `@CreateProcess` annotation:

```
@CreateProcess(definition="todo")
public void createTodo() { ... }
```

7.4.4. Task assignment

When a process starts, task instances are created. These must be assigned to users or user groups. We can either hardcode our actor ids, or delegate to a Seam component:

```
<task name="todo" description="#{todoList.description}">
    <assignment actor-id="#{actor.id}"/>
</task>
```

In this case, we have simply assigned the task to the current user. We can also assign tasks to a pool:

```
<task name="todo" description="#{todoList.description}">
    <assignment pooled-actors="employees"/>
</task>
```

7.4.5. Task lists

Several built-in Seam components make it easy to display task lists. The `pooledTaskInstanceList` is a list of pooled tasks that users may assign to themselves:

```
<h:dataTable value="#{pooledTaskInstanceList}" var="task">
    <h:column>
        <f:facet name="header">Description</f:facet>
        <h:outputText value="#{task.description}"/>
    </h:column>
    <h:column>
        <s:link action="#{pooledTask.assignToCurrentActor}" value="Assign" taskInstance="#{task}"/>
    </h:column>
</h:dataTable>
```

Note that instead of `<s:link>` we could have used a plain JSF `<h:commandLink>`:

```
<h:commandLink action="#{pooledTask.assignToCurrentActor}">
    <f:param name="taskId" value="#{task.id}"/>
</h:commandLink>
```

The `pooledTask` component is a built-in component that simply assigns the task to the current user.

The `taskInstanceListByType` component includes tasks of a particular type that are assigned to the current user:

```
<h:dataTable value="#{taskInstanceListByType['todo']}" var="task">
    <h:column>
        <f:facet name="header">Description</f:facet>
        <h:outputText value="#{task.description}"/>
    </h:column>
    <h:column>
        <s:link action="#{todoList.start}" value="Start Work" taskInstance="#{task}"/>
    </h:column>
</h:dataTable>
```

7.4.6. Performing a task

To begin work on a task, we use either `@StartTask` or `@BeginTask` on the listener method:

```
@StartTask
public String start() { ... }
```

These annotations begin a special kind of conversation that has significance in terms of the overarching business process. Work done by this conversation has access to state held in the business process context.

If we end the conversation using `@EndTask`, Seam will signal the completion of the task:

```
@EndTask(transition="completed")
public String completed() { ... }
```

(Alternatively, we could have used `<end-conversation>` as shown above.)

At this point, jBPM takes over and continues executing the business process definition. (In more complex processes, several tasks might need to be completed before process execution can resume.)

Please refer to the jBPM documentation for a more thorough overview of the sophisticated features that jBPM provides for managing complex business processes.

Chapter 8. Internationalization and themes

Seam makes it easy to build internationalized applications by providing several built-in components for handling multi-language UI messages.

8.1. Locales

Each user login session has an associated instance of `java.util.Locale` (available to the application as a session-scoped component named `locale`). Under normal circumstances, you won't need to do any special configuration to set the locale. Seam just delegates to JSF to determine the active locale:

- If there is a locale associated with the HTTP request (the browser locale), and that locale is in the list of supported locales from `faces-config.xml`, use that locale for the rest of the session.
- Otherwise, if a default locale was specified in the `faces-config.xml`, use that locale for the rest of the session.
- Otherwise, use the default locale of the server.

It is *possible* to set the locale manually via the Seam configuration properties `org.jboss.seam.core.localeSelector.language`, `org.jboss.seam.core.localeSelector.country` and `org.jboss.seam.core.localeSelector.variant`, but we can't think of any good reason to ever do this.

It is, however, useful to allow the user to set the locale manually via the application user interface. Seam provides built-in functionality for overriding the locale determined by the algorithm above. All you have to do is add the following fragment to a form in your JSP or Facelets page:

```
<h:selectOneMenu value="#{localeSelector.language}">
    <f:selectItem itemLabel="English" itemValue="en"/>
    <f:selectItem itemLabel="Deutsch" itemValue="de"/>
    <f:selectItem itemLabel="Francais" itemValue="fr"/>
</h:selectOneMenu>
<h:commandButton action="#{localeSelector.select}" value="#{messages['ChangeLanguage']}"/>
```

Or, if you want a list of all supported locales from `faces-config.xml`, just use:

```
<h:selectOneMenu value="#{localeSelector.localeString}">
    <f:selectItems value="#{localeSelector.supportedLocales}"/>
</h:selectOneMenu>
<h:commandButton action="#{localeSelector.select}" value="#{messages['ChangeLanguage']}"/>
```

When this use selects an item from the drop-down, and clicks the button, the Seam and JSF locales will be overridden for the rest of the session.

8.2. Labels

JSF supports internationalization of user interface labels and descriptive text via the use of `<f:loadBundle />`. You can use this approach in Seam applications. Alternatively, you can take advantage of the Seam `messages` component to display templated labels with embedded EL expressions.

8.2.1. Defining labels

Each login session has an associated instance of `java.util.ResourceBundle` (available to the application as a session-scoped component named `org.jboss.seam.core.resourceBundle`). You'll need to make your internationalized labels available via this special resource bundle. By default, the resource bundle used by Seam is named `messages` and so you'll need to define your labels in files named `messages.properties`, `messages_en.properties`, `messages_en_AU.properties`, etc. These files usually belong in the `WEB-INF/classes` directory.

So, in `messages_en.properties`:

```
Hello=Hello
```

And in `messages_en_AU.properties`:

```
Hello=G'day
```

You can select a different name for the resource bundle by setting the Seam configuration property named `org.jboss.seam.core.resourceBundle.bundleNames`. You can even specify a list of resource bundle names to be searched (depth first) for messages.

```
<core:resource-bundle>
    <core:bundle-names>
        <value>mycompany_messages</value>
        <value>standard_messages</value>
    </core:bundle-names>
</core:resource-bundle>
```

If you want to define a message just for a particular page, you can specify it in a resource bundle with the same name as the JSF view id, with the leading `/` and trailing file extension removed. So we could put our message in `welcome/hello_en.properties` if we only needed to display the message on `/welcome/hello.jsp`.

You can even specify an explicit bundle name in `pages.xml`:

```
<page view-id="/welcome/hello.jsp" bundle="HelloMessages"/>
```

Then we could use messages defined in `HelloMessages.properties` on `/welcome/hello.jsp`.

8.2.2. Displaying labels

If you define your labels using the Seam resource bundle, you'll be able to use them without having to type `<f:loadBundle ... />` on every page. Instead, you can simply type:

```
<h:outputText value="#{messages['Hello']}"/>
```

or:

```
<h:outputText value="#{messages.Hello}"/>
```

Even better, the messages themselves may contain EL expressions:

```
Hello=Hello, #{user.firstName} #{user.lastName}
```

```
Hello=G'day, #{user.firstName}
```

You can even use the messages in your code:

```
@In private Map<String, String> messages;
```

```
@In("#{messages['Hello']}") private String helloMessage;
```

8.2.3. Faces messages

The `facesMessages` component is a super-convenient way to display success or failure messages to the user. The functionality we just described also works for faces messages:

```
@Name("hello")
@Stateless
public class HelloBean implements Hello {
    @In FacesMessages facesMessages;

    public String sayIt() {
        facesMessages.addFromResourceBundle("Hello");
    }
}
```

This will display `Hello, Gavin King` or `G'day, Gavin`, depending upon the user's locale.

8.3. Timezones

There is also a session-scoped instance of `java.util.Timezone`, named `org.jboss.seam.core.timezone`, and a Seam component for changing the timezone named `org.jboss.seam.core.timezoneSelector`. By default, the timezone is the default timezone of the server. Unfortunately, the JSF specification says that all dates and times should be assumed to be UTC, and displayed as UTC, unless a timezone is explicitly specified using `<f:convertDateTime>`. This is an extremely inconvenient default behavior.

Seam overrides this behavior, and defaults all dates and times to the Seam timezone. In addition, Seam provides the `<s:convertDateTime>` tag which always performs conversions in the Seam timezone.

8.4. Themes

Seam applications are also very easily skinnable. The theme API is very similar to the localization API, but of course these two concerns are orthogonal, and some applications support both localization and themes.

First, configure the set of supported themes:

```
<theme:theme-selector cookie-enabled="true">
    <theme:available-themes>
        <value>default</value>
        <value>accessible</value>
        <value>printable</value>
    </theme:available-themes>
</theme:theme-selector>
```

Note that the first theme listed is the default theme.

Themes are defined in a properties file with the same name as the theme. For example, the `default` theme is defined as a set of entries in `default.properties`. For example, `default.properties` might define:

```
css ../screen.css
template template.xhtml
```

Usually the entries in a theme resource bundle will be paths to CSS styles or images and names of facelets templates (unlike localization resource bundles which are usually text).

Now we can use these entries in our JSP or facelets pages. For example, to theme the stylesheet in a facelets page:

```
<link href="#{theme.css}" rel="stylesheet" type="text/css" />
```

Most powerfully, facelets lets us theme the template used by a `<ui:composition>`:

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    template="#{theme.template}">
```

Just like the locale selector, there is a built-in theme selector to allow the user to freely switch themes:

```
<h:selectOneMenu value="#{themeSelector.theme}">
    <f:selectItems value="#{themeSelector.themes}"/>
</h:selectOneMenu>
<h:commandButton action="#{themeSelector.select}" value="Select Theme"/>
```

8.5. Persisting locale and theme preferences via cookies

The locale selector, theme selector and timezone selector all support persistence of locale and theme preference to a cookie. Simply set the `cookie-enabled` configuration property:

```
<theme:theme-selector cookie-enabled="true">
    <theme:available-themes>
        <value>default</value>
        <value>accessible</value>
        <value>printable</value>
    </theme:available-themes>
</theme:theme-selector>

<core:locale-selector cookie-enabled="true"/>
```

Chapter 9. Asynchronicity and messaging

Seam makes it very easy to perform work asynchronously from a web request. When most people think of asynchronicity in Java EE, they think of using JMS. This is certainly one way to approach the problem in Seam, and is the right way when you have strict and well-defined quality of service requirements. Seam makes it easy to send and recieve JMS messages using Seam components.

But for many usecases, JMS is overkill. Seam layers a simple asynchronous method and event facility over the EJB 3.0 timer service.

9.1. Asynchronicity

Asynchronous events and method calls have the same quality of service expectations as the container's EJB timer service. If you're not familiar with the Timer service, don't worry, you don't need to interact with it directly if you want to use asynchronous methods in Seam.

To use asynchronous methods and events, you need to add the following line to `components.xml`:

```
<core:dispatcher/>
```

Note that this functionality is not available in environments which do not support EJB 3.0.

9.1.1. Asynchronous methods

In simplest form, an asynchronous call just lets a method call be processed asynchronously (in a different thread) from the caller. We usually use an asynchronous call when we want to return an immediate response to the client, and let some expensive work be processed in the background. This pattern works very well in applications which use AJAX, where the client can automatically poll the server for the result of the work.

For EJB components, we annotate the local interface to specify that a method is processed asynchronously.

```
@Local
public interface PaymentHandler
{
    @Asynchronous
    public void processPayment(Payment payment);
}
```

(For JavaBean components we can annotate the component implementation class if we like.)

The use of asynchronicity is transparent to the bean class:

```
@Stateless
@Name("paymentHandler")
public class PaymentHandlerBean implements PaymentHandler
{
    public void processPayment(Payment payment)
    {
        //do some work!
    }
}
```

And also transparent to the client:

```
@Stateful
```

```
@Name("paymentAction")
public class CreatePaymentAction
{
    @In(create=true) PaymentHandler paymentHandler;
    @In Bill bill;

    public String pay()
    {
        paymentHandler.processPayment( new Payment(bill) );
        return "success";
    }
}
```

The asynchronous method is processed in a completely new event context and does not have access to the session or conversation context state of the caller. However, the business process context *is* propagated.

Asynchronous method calls may be scheduled for later execution using the `@Duration`, `@Expiration` and `@IntervalDuration` annotations.

```
@Local
public interface PaymentHandler
{
    @Asynchronous
    public void processScheduledPayment(Payment payment, @Expiration Date date);

    @Asynchronous
    public void processRecurringPayment(Payment payment, @Expiration Date date, @IntervalDuration Date
}
```

```
@Stateful
@Name("paymentAction")
public class CreatePaymentAction
{
    @In(create=true) PaymentHandler paymentHandler;
    @In Bill bill;

    public String schedulePayment()
    {
        paymentHandler.processScheduledPayment( new Payment(bill), bill.getDueDate() );
        return "success";
    }

    public String scheduleRecurringPayment()
    {
        paymentHandler.processRecurringPayment( new Payment(bill), bill.getDueDate(), ONE_MONTH );
        return "success";
    }
}
```

Both client and server may access the `Timer` object associated with the invocation.

```
@Local
public interface PaymentHandler
{
    @Asynchronous
    public Timer processScheduledPayment(Payment payment, @Expiration Date date);
}
```

```
@Stateless
@Name("paymentHandler")
public class PaymentHandlerBean implements PaymentHandler
{
    @In Timer timer;

    public Timer processScheduledPayment(Payment payment, @Expiration Date date)
    {
```

```
        //do some work!

        return timer; //note that return value is completely ignored
    }

}
```

```
@Stateful
@Name("paymentAction")
public class CreatePaymentAction
{
    @In(create=true) PaymentHandler paymentHandler;
    @In Bill bill;

    public String schedulePayment()
    {
        Timer timer = paymentHandler.processScheduledPayment( new Payment(bill), bill.getDueDate() );
        return "success";
    }
}
```

Asynchronous methods cannot return any other value to the caller.

9.1.2. Asynchronous events

Component-driven events may also be asynchronous. To raise an event for asynchronous processing, simply call the `raiseAsynchronousEvent()` methods of the `Events` class. To schedule a timed event, call one of the `raiseTimedEvent()` methods. Components may observe asynchronous events in the usual way, but remember that only the business process context is propagated to the asynchronous thread.

9.2. Messaging in Seam

Seam makes it easy to send and receive JMS messages to and from Seam components.

9.2.1. Configuration

To configure Seam's infrastructure for sending JMS messages, you need to tell Seam about any topics and queues you want to send messages to, and also tell Seam where to find the `QueueConnectionFactory` and/or `TopicConnectionFactory`.

Seam defaults to using `UIL2ConnectionFactory` which is the usual connection factory for use with JBossMQ. If you are using some other JMS provider, you need to set one or both of `queueConnection.queueConnectionFactoryJndiName` and `topicConnection.topicConnectionFactoryJndiName` in `seam.properties`, `web.xml` or `components.xml`.

You also need to list topics and queues in `components.xml` to install Seam managed `TopicPublishers` and `QueueSenders`:

```
<jms:managed-topic-publisher name="stockTickerPublisher" auto-create="true" topic-jndi-name="topic/sto

<jms:managed-queue-sender name="paymentQueueSender" auto-create="true" queue-jndi-name="queue/paymentQ
```

9.2.2. Sending messages

Now, you can inject a JMS `TopicPublisher` and `TopicSession` into any component:

```
@In
private TopicPublisher stockTickerPublisher;
@In
private TopicSession topicSession;

public void publish(StockPrice price) {
      try
      {
         topicPublisher.publish( topicSession.createObjectMessage(price) );
      }
      catch (Exception ex)
      {
         throw new RuntimeException(ex);
      }
}
```

Or, for working with a queue:

```
@In
private QueueSender paymentQueueSender;
@In
private QueueSession queueSession;

public void publish(Payment payment) {
      try
      {
         paymentQueueSender.publish( queueSession.createObjectMessage(payment) );
      }
      catch (Exception ex)
      {
         throw new RuntimeException(ex);
      }
}
```

9.2.3. Receiving messages using a message-driven bean

You can process messages using any EJB3 message driven bean. Message-driven beans may even be Seam components, in which case it is possible to inject other event and application scoped Seam components.

9.2.4. Receiving messages in the client

Seam Remoting lets you subscribe to a JMS topic from client-side JavaScript. This is described in the next chapter.

Chapter 10. Remoting

Seam provides a convenient method of remotely accessing components from a web page, using AJAX (Asynchronous Javascript and XML). The framework for this functionality is provided with almost no up-front development effort - your components only require simple annotating to become accessible via AJAX. This chapter describes the steps required to build an AJAX-enabled web page, then goes on to explain the features of the Seam Remoting framework in more detail.

10.1. Configuration

To use remoting, the Seam Remoting servlet must first be configured in your `web.xml` file:

```
<servlet>
  <servlet-name>Seam Remoting</servlet-name>
  <servlet-class>org.jboss.seam.remoting.SeamRemotingServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>Seam Remoting</servlet-name>
  <url-pattern>/seam/remoting/*</url-pattern>
</servlet-mapping>
```

The next step is to import the necessary Javascript into your web page. There are a minimum of two scripts that must be imported. The first one contains all the client-side framework code that enables remoting functionality:

```
<script type="text/javascript" src="seam/remoting/resource/remote.js"></script>
```

The second script contains the stubs and type definitions for the components you wish to call. It is generated dynamically based on the local interface of your components, and includes type definitions for all of the classes that can be used to call the remotable methods of the interface. The name of the script reflects the name of your component. For example, if you have a stateless session bean annotated with `@Name("customerAction")`, then your script tag should look like this:

```
<script type="text/javascript" src="seam/remoting/interface.js?customerAction"></script>
```

If you wish to access more than one component from the same page, then include them all as parameters of your script tag:

```
<script type="text/javascript" src="seam/remoting/interface.js?customerAction&accountAction"></scrip
```

10.2. The "Seam" object

Client-side interaction with your components is all performed via the `Seam` Javascript object. This object is defined in `remote.js`, and you'll be using it to make asynchronous calls against your component. It is split into two areas of functionality; `Seam.Component` contains methods for working with components and `Seam.Remoting` contains methods for executing remote requests. The easiest way to become familiar with this object is to start with a simple example.

10.2.1. A Hello World example

Let's step through a simple example to see how the `Seam` object works. First of all, let's create a new Seam component called `helloAction`.

```
@Stateless
@Name("helloAction")
public class HelloAction implements HelloLocal {
  public String sayHello(String name) {
    return "Hello, " + name;
  }
}
```

You also need to create a local interface for our new component - take special note of the `@WebRemote` annotation, as it's required to make our method accessible via remoting:

```
@Local
public interface HelloLocal {
  @WebRemote
  public String sayHello(String name);
}
```

That's all the server-side code we need to write. Now for our web page - create a new page and import the following scripts:

```
<script type="text/javascript" src="seam/remoting/resource/remote.js"></script>
<script type="text/javascript" src="seam/remoting/interface.js?helloAction"></script>
```

To make this a fully interactive user experience, let's add a button to our page:

```
<button onclick="javascript:sayHello()">Say Hello</button>
```

We'll also need to add some more script to make our button actually do something when it's clicked:

```
```

```
<script type="text/javascript">
  //<![CDATA[

  function sayHello() {
    var name = prompt("What is your name?");
    Seam.Component.getInstance("helloAction").sayHello(name, sayHelloCallback);
  }

  function sayHelloCallback(result) {
    alert(result);
  }

  // ]]>
</script>
```

We're done! Deploy your application and browse to your page. Click the button, and enter a name when prompted. A message box will display the hello message confirming that the call was successful. If you want to save some time, you'll find the full source code for this Hello World example in Seam's `/examples/remoting/helloworld` directory.

So what does the code of our script actually do? Let's break it down into smaller pieces. To start with, you can see from the Javascript code listing that we have implemented two methods - the first method is responsible for prompting the user for their name and then making a remote request. Take a look at the following line:

```
Seam.Component.getInstance("helloAction").sayHello(name, sayHelloCallback);
```

The first section of this line, `Seam.Component.getInstance("helloAction")` returns a proxy, or "stub" for our `helloAction` component. We can invoke the methods of our component against this stub, which is exactly what happens with the remainder of the line: `sayHello(name, sayHelloCallback);`.

What this line of code in its completeness does, is invoke the `sayHello` method of our component, passing in `name` as a parameter. The second parameter, `sayHelloCallback` isn't a parameter of our component's `sayHello` method, instead it tells the Seam Remoting framework that once it receives the response to our request, it should pass it to the `sayHelloCallback` Javascript method. This callback parameter is entirely optional, so feel free to leave it out if you're calling a method with a `void` return type or if you don't care about the result.

The `sayHelloCallback` method, once receiving the response to our remote request then pops up an alert message displaying the result of our method call.

10.2.2. Seam.Component

The `Seam.Component` Javascript object provides a number of client-side methods for working with your Seam components. The two main methods, `newInstance()` and `getInstance()` are documented in the following sections however their main difference is that `newInstance()` will always create a new instance of a component type, and `getInstance()` will return a singleton instance.

Seam.Component.newInstance()

Use this method to create a new instance of an entity or Javabean component. The object returned by this method will have the same getter/setter methods as its server-side counterpart, or alternatively if you wish you can access its fields directly. Take the following Seam entity component for example:

```
@Name("customer")
@Entity
public class Customer implements Serializable
{
  private Integer customerId;
  private String firstName;
  private String lastName;

  @Column public Integer getCustomerId() {
    return customerId;
  }

  public void setCustomerId(Integer customerId} {
    this.customerId = customerId;
  }

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

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

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

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

To create a client-side Customer you would write the following code:

```
var customer = Seam.Component.newInstance("customer");
```

Then from here you can set the fields of the customer object:

```
customer.setFirstName("John");
// Or you can set the fields directly
customer.lastName = "Smith";
```

Seam.Component.getInstance()

The `getInstance()` method is used to get a reference to a Seam session bean component stub, which can then be used to remotely execute methods against your component. This method returns a singleton for the specified component, so calling it twice in a row with the same component name will return the same instance of the component.

To continue our example from before, if we have created a new `customer` and we now wish to save it, we would pass it to the `saveCustomer()` method of our `customerAction` component:

```
Seam.Component.getInstance("customerAction").saveCustomer(customer);
```

Seam.Component.getComponentName()

Passing an object into this method will return its component name if it is a component, or `null` if it is not.

```
if (Seam.Component.getComponentName(instance) == "customer")
  alert("Customer");
else if (Seam.Component.getComponentName(instance) == "staff")
  alert("Staff member");
```

10.2.3. Seam.Remoting

Most of the client side functionality for Seam Remoting is contained within the `Seam.Remoting` object. While you shouldn't need to directly call most of its methods, there are a couple of important ones worth mentioning.

Seam.Remoting.createType()

If your application contains or uses Javabean classes that aren't Seam components, you may need to create these types on the client side to pass as parameters into your component method. Use the `createType()` method to create an instance of your type. Pass in the fully qualified Java class name as a parameter:

```
var widget = Seam.Remoting.createType("com.acme.widgets.MyWidget");
```

Seam.Remoting.getTypeName()

This method is the equivalent of `Seam.Component.getComponentName()` but for non-component types. It will return the name of the type for an object instance, or `null` if the type is not known. The name is the fully qualified name of the type's Java class.

10.3. Client Interfaces

In the configuration section above, the interface, or "stub" for our component is imported into our page via `seam/remoting/interface.js`:

```
<script type="text/javascript" src="seam/remoting/interface.js?customerAction"></script>
```

By including this script in our page, the interface definitions for our component, plus any other components or types that are required to execute the methods of our component are generated and made available for the remoting framework to use.

There are two types of client stub that can be generated, "executable" stubs and "type" stubs. Executable stubs are behavioural, and are used to execute methods against your session bean components, while type stubs contain state and represent the types that can be passed in as parameters or returned as a result.

The type of client stub that is generated depends on the type of your Seam component. If the component is a session bean, then an executable stub will be generated, otherwise if it's an entity or JavaBean, then a type stub

will be generated. There is one exception to this rule; if your component is a JavaBean (ie it is not a session bean nor an entity bean) and any of its methods are annotated with @WebRemote, then an executable stub will be generated for it instead of a type stub. This allows you to use remoting to call methods of your JavaBean components in a non-EJB environment where you don't have access to session beans.

10.4. The Context

The Seam Remoting Context contains additional information which is sent and received as part of a remoting request/response cycle. At this stage it only contains the conversation ID but may be expanded in the future.

10.4.1. Setting and reading the Conversation ID

If you intend on using remote calls within the scope of a conversation then you need to be able to read or set the conversation ID in the Seam Remoting Context. To read the conversation ID after making a remote request call `Seam.Remoting.getContext().getConversationId()`. To set the conversation ID before making a request, call `Seam.Remoting.getContext().setConversationId()`.

If the conversation ID hasn't been explicitly set with `Seam.Remoting.getContext().setConversationId()`, then it will be automatically assigned the first valid conversation ID that is returned by any remoting call. If you are working with multiple conversations within your page, then you may need to explicitly set the conversation ID before each call. If you are working with just a single conversation, then you don't need to do anything special.

10.5. Batch Requests

Seam Remoting allows multiple component calls to be executed within a single request. It is recommended that this feature is used wherever it is appropriate to reduce network traffic.

The method `Seam.Remoting.startBatch()` will start a new batch, and any component calls executed after starting a batch are queued, rather than being sent immediately. When all the desired component calls have been added to the batch, the `Seam.Remoting.executeBatch()` method will send a single request containing all of the queued calls to the server, where they will be executed in order. After the calls have been executed, a single response containining all return values will be returned to the client and the callback functions (if provided) triggered in the same order as execution.

If you start a new batch via the `startBatch()` method but then decide you don't want to send it, the `Seam.Remoting.cancelBatch()` method will discard any calls that were queued and exit the batch mode.

To see an example of a batch being used, take a look at `/examples/remoting/chatroom`.

10.6. Working with Data types

10.6.1. Primitives / Basic Types

This section describes the support for basic data types. On the server side these values are generally compatible with either their primitive type or their corresponding wrapper class.

String

Simply use Javascript String objects when setting String parameter values.

Number

There is support for all number types supported by Java. On the client side, number values are always serialized as their String representation and then on the server side they are converted to the correct destination type. Conversion into either a primitive or wrapper type is supported for `Byte`, `Double`, `Float`, `Integer`, `Long` and `Short` types.

Boolean

Booleans are represented client side by Javascript Boolean values, and server side by a Java boolean.

10.6.2. JavaBeans

In general these will be either Seam entity or JavaBean components, or some other non-component class. Use the appropriate method (either `Seam.Component.newInstance()` for Seam components or `Seam.Remoting.createType()` for everything else) to create a new instance of the object.

It is important to note that only objects that are created by either of these two methods should be used as parameter values, where the parameter is not one of the other valid types mentioned anywhere else in this section. In some situations you may have a component method where the exact parameter type cannot be determined, such as:

```
@Name("myAction")
public class MyAction implements MyActionLocal {
  public void doSomethingWithObject(Object obj) {
    // code
  }
}
```

In this case you might want to pass in an instance of your `myWidget` component, however the interface for `myAction` won't include `myWidget` as it is not directly referenced by any of its methods. To get around this, `MyWidget` needs to be explicitly imported:

```
<script type="text/javascript" src="seam/remoting/interface.js?myAction&myWidget"></script>
```

This will then allow a `myWidget` object to be created with `Seam.Component.newInstance("myWidget")`, which can then be passed to `myAction.doSomethingWithObject()`.

10.6.3. Dates and Times

Date values are serialized into a String representation that is accurate to the millisecond. On the client side, use a Javascript Date object to work with date values. On the server side, use any `java.util.Date` (or descendent, such as `java.sql.Date` or `java.sql.Timestamp` class.

10.6.4. Enums

On the client side, enums are treated the same as Strings. When setting the value for an enum parameter, simply use the String representation of the enum. Take the following component as an example:

```
@Name("paintAction")
public class paintAction implements paintLocal {
  public enum Color {red, green, blue, yellow, orange, purple};

  public void paint(Color color) {
    // code
  }
}
```

To call the `paint()` method with the color `red`, pass the parameter value as a String literal:

```
Seam.Component.getInstance("paintAction").paint("red");
```

The inverse is also true - that is, if a component method returns an enum parameter (or contains an enum field anywhere in the returned object graph) then on the client-side it will be represented as a String.

10.6.5. Collections

Bags

Bags cover all collection types including arrays, collections, lists, sets, (but excluding Maps - see the next section for those), and are implemented client-side as a Javascript array. When calling a component method that accepts one of these types as a parameter, your parameter should be a Javascript array. If a component method returns one of these types, then the return value will also be a Javascript array. The remoting framework is clever enough on the server side to convert the bag to an appropriate type for the component method call.

Maps

As there is no native support for Maps within Javascript, a simple Map implementation is provided with the Seam Remoting framework. To create a Map which can be used as a parameter to a remote call, create a new `Seam.Remoting.Map` object:

```
var map = new Seam.Remoting.Map();
```

This Javascript implementation provides basic methods for working with Maps: `size()`, `isEmpty()`, `keySet()`, `values()`, `get(key)`, `put(key, value)`, `remove(key)` and `contains(key)`. Each of these methods are equivalent to their Java counterpart. Where the method returns a collection, such as `keySet()` and `values()`, a Javascript Array object will be returned that contains the key or value objects (respectively).

10.7. Debugging

To aid in tracking down bugs, it is possible to enable a debug mode which will display the contents of all the packets send back and forth between the client and server in a popup window. To enable debug mode, either execute the `setDebug()` method in Javascript:

```
```

```
Seam.Remoting.setDebug(true);
```

Or configure it via components.xml:

```
<remoting:remoting-config debug="true"/>
```

To turn off debugging, call `setDebug(false)`. If you want to write your own messages to the debug log, call `Seam.Remoting.log(message)`.

10.8. The Loading Message

The default loading message that appears in the top right corner of the screen can be modified, its rendering customised or even turned off completely.

10.8.1. Changing the message

To change the message from the default "Please Wait..." to something different, set the value of `Seam.Remoting.loadingMessage`:

```
Seam.Remoting.loadingMessage = "Loading...";
```

10.8.2. Hiding the loading message

To completely suppress the display of the loading message, override the implementation of `displayLoadingMessage()` and `hideLoadingMessage()` with functions that instead do nothing:

```
// don't display the loading indicator
Seam.Remoting.displayLoadingMessage = function() {};
Seam.Remoting.hideLoadingMessage = function() {};
```

10.8.3. A Custom Loading Indicator

It is also possible to override the loading indicator to display an animated icon, or anything else that you want. To do this override the `displayLoadingMessage()` and `hideLoadingMessage()` messages with your own implementation:

```
Seam.Remoting.displayLoadingMessage = function() {
  // Write code here to display the indicator
};

Seam.Remoting.hideLoadingMessage = function() {
  // Write code here to hide the indicator
};
```

10.9. Controlling what data is returned

When a remote method is executed, the result is serialized into an XML response that is returned to the client. This response is then unmarshaled by the client into a Javascript object. For complex types (i.e. Javabeans) that include references to other objects, all of these referenced objects are also serialized as part of the response. These objects may reference other objects, which may reference other objects, and so forth. If left unchecked, this object "graph" could potentially be enormous, depending on what relationships exist between your objects. And as a side issue (besides the potential verbosity of the response), you might also wish to prevent sensitive information from being exposed to the client.

Seam Remoting provides a simple means to "constrain" the object graph, by specifying the `exclude` field of the remote method's `@WebRemote` annotation. This field accepts a String array containing one or more paths specified using dot notation. When invoking a remote method, the objects in the result's object graph that match these paths are excluded from the serialized result packet.

For all our examples, we'll use the following `Widget` class:

```
@Name("widget")
public class Widget
{
  private String value;
  private String secret;
  private Widget child;
  private Map<String,Widget> widgetMap;
  private List<Widget> widgetList;

  // getters and setters for all fields
}
```

10.9.1. Constraining normal fields

If your remote method returns an instance of `Widget`, but you don't want to expose the `secret` field because it contains sensitive information, you would constrain it like this:

```
  @WebRemote(exclude = {"secret"})
  public Widget getWidget();
```

The value "secret" refers to the `secret` field of the returned object. Now, suppose that we don't care about exposing this particular field to the client. Instead, notice that the `Widget` value that is returned has a field `child` that is also a `Widget`. What if we want to hide the `child`'s `secret` value instead? We can do this by using dot notation to specify this field's path within the result's object graph:

```
  @WebRemote(exclude = {"child.secret"})
  public Widget getWidget();
```

10.9.2. Constraining Maps and Collections

The other place that objects can exist within an object graph are within a `Map` or some kind of collection (`List`, `Set`, `Array`, etc). Collections are easy, and are treated like any other field. For example, if our `Widget` contained a list of other `Widgets` in its `widgetList` field, to constrain the `secret` field of the `Widgets` in this list the annotation would look like this:

```
```

```
@WebRemote(exclude = {"widgetList.secret"})
public Widget getWidget();
```

To constrain a `Map`'s key or value, the notation is slightly different. Appending `[key]` after the `Map`'s field name will constrain the `Map`'s key object values, while `[value]` will constrain the value object values. The following example demonstrates how the values of the `widgetMap` field have their `secret` field constrained:

```
@WebRemote(exclude = {"widgetMap[value].secret"})
public Widget getWidget();
```

10.9.3. Constraining objects of a specific type

There is one last notation that can be used to constrain the fields of a type of object no matter where in the result's object graph it appears. This notation uses either the name of the component (if the object is a Seam component) or the fully qualified class name (only if the object is not a Seam component) and is expressed using square brackets:

```
@WebRemote(exclude = {"[widget].secret"})
public Widget getWidget();
```

10.9.4. Combining Constraints

Constraints can also be combined, to filter objects from multiple paths within the object graph:

```
@WebRemote(exclude = {"widgetList.secret", "widgetMap[value].secret"})
public Widget getWidget();
```

10.10. JMS Messaging

Seam Remoting provides experimental support for JMS Messaging. This section describes the JMS support that is currently implemented, but please note that this may change in the future. It is currently not recommended that this feature is used within a production environment.

10.10.1. Configuration

Before you can subscribe to a JMS topic, you must first configure a list of the topics that can be subscribed to by Seam Remoting. List the topics under `org.jboss.seam.remoting.messaging.subscriptionRegistry.allowedTopics` in `seam.properties`, `web.xml` or `components.xml`.

```
<remoting:remoting-config poll-timeout="5" poll-interval="1"/>
```

10.10.2. Subscribing to a JMS Topic

The following example demonstrates how to subscribe to a JMS Topic:

```
function subscriptionCallback(message)
{
  if (message instanceof Seam.Remoting.TextMessage)
    alert("Received message: " + message.getText());
}

Seam.Remoting.subscribe("topicName", subscriptionCallback);
```

The `Seam.Remoting.subscribe()` method accepts two parameters, the first being the name of the JMS Topic to subscribe to, the second being the callback function to invoke when a message is received.

There are two types of messages supported, Text messages and Object messages. If you need to test for the type of message that is passed to your callback function you can use the `instanceof` operator to test whether the message is a `Seam.Remoting.TextMessage` or `Seam.Remoting.ObjectMessage`. A `TextMessage` contains the text value in its `text` field (or alternatively call `getText()` on it), while an `ObjectMessage` contains its object value in its `object` field (or call its `getObject()` method).

10.10.3. Unsubscribing from a Topic

To unsubscribe from a topic, call `Seam.Remoting.unsubscribe()` and pass in the topic name:

```
Seam.Remoting.unsubscribe("topicName");
```

10.10.4. Tuning the Polling Process

There are two parameters which you can modify to control how polling occurs. The first one is `Seam.Remoting.pollInterval`, which controls how long to wait between subsequent polls for new messages. This parameter is expressed in seconds, and its default setting is 10.

The second parameter is `Seam.Remoting.pollTimeout`, and is also expressed as seconds. It controls how long a request to the server should wait for a new message before timing out and sending an empty response. Its default is 0 seconds, which means that when the server is polled, if there are no messages ready for delivery then an empty response will be immediately returned.

Caution should be used when setting a high `pollTimeout` value; each request that has to wait for a message means that a server thread is tied up until a message is received, or until the request times out. If many such requests are being served simultaneously, it could mean a large number of threads become tied up because of this reason.

It is recommended that you set these options via components.xml, however they can be overridden via Javascript if desired. The following example demonstrates how to configure the polling to occur much more aggressively. You should set these parameters to suitable values for your application:

Via components.xml:

```
<component name="org.jboss.seam.remoting.remotingConfig">
  <property name="pollTimeout">5</property>
  <property name="pollInterval">1</property>
</component>
```

```

```

Via JavaScript:

```
// Only wait 1 second between receiving a poll response and sending the next poll request.
Seam.Remoting.pollInterval = 1;

// Wait up to 5 seconds on the server for new messages
Seam.Remoting.pollTimeout = 5;
```

Chapter 11. Seam and JBoss Rules

Seam makes it easy to call JBoss Rules (Drools) rulebases from Seam components or jBPM process definitions.

11.1. Installing rules

The first step is to make an instance of `org.drools.RuleBase` available in a Seam context variable. In most rules-driven applications, rules need to be dynamically deployable, so you will need to implement some solution that allows you to deploy rules and make them available to Seam (a future release of Drools will provide a Rule Server that solves this problem). For testing purposes, Seam provides a built-in component that compiles a static set of rules from the classpath. You can install this component via `components.xml`:

```
<drools:rule-base name="policyPricingRules">
    <drools:rule-files>
        <value>policyPricingRules</value>
    </drools:rule-files>
</drools:rule-base>
```

This component compiles rules from a set of `.drl` files and caches an instance of `org.drools.RuleBase` in the Seam `APPLICATION` context. Note that it is quite likely that you will need to install multiple rule bases in a rule-driven application.

If you want to use a Drools DSL, you alse need to specify the DSL definition:

```
<drools:rule-base name="policyPricingRules" dsl-file="policyPricing.dsl">
    <drools:rule-files>
        <value>policyPricingRules</value>
    </drools:rule-files>
</drools:rule-base>
```

Next, we need to make an instance of `org.drools.WorkingMemory` available to each conversation. (Each `WorkingMemory` accumulates facts relating to the current conversation.)

```
<drools:managed-working-memory name="policyPricingWorkingMemory" auto-create="true" rule-base="#{polic
```

Notice that we gave the `policyPricingWorkingMemory` a reference back to our rule base via the `ruleBase` configuration property.

11.2. Using rules from a Seam component

We can now inject our `WorkingMemory` into any Seam component, assert facts, and fire rules:

```
@In WorkingMemory policyPricingWorkingMemory;

@In Policy policy;
@In Customer customer;

public void pricePolicy() throws FactException
{
    policyPricingWorkingMemory.assertObject(policy);
    policyPricingWorkingMemory.assertObject(customer);
    policyPricingWorkingMemory.fireAllRules();
}
```

11.3. Using rules from a jBPM process definition

You can even allow a rule base to act as a jBPM action handler, decision handler, or assignment handler—in either a pageflow or business process definition.

```
<decision name="approval">

    <handler class="org.jboss.seam.drools.DroolsDecisionHandler">
        <workingMemoryName>orderApprovalRulesWorkingMemory</workingMemoryName>
        <assertObjects>
            <element>#{customer}</element>
            <element>#{order}</element>
            <element>#{order.lineItems}</element>
        </assertObjects>
    </handler>

    <transition name="approved" to="ship">
        <action class="org.jboss.seam.drools.DroolsActionHandler">
            <workingMemoryName>shippingRulesWorkingMemory</workingMemoryName>
            <assertObjects>
                <element>#{customer}</element>
                <element>#{order}</element>
                <element>#{order.lineItems}</element>
            </assertObjects>
        </action>
    </transition>

    <transition name="rejected" to="cancelled"/>

</decision>
```

The `<assertObjects>` element specifies EL expressions that return an object or collection of objects to be asserted as facts into the `WorkingMemory`.

There is also support for using Drools for jBPM task assignments:

```
<task-node name="review">
    <task name="review" description="Review Order">
        <assignment handler="org.jboss.seam.drools.DroolsAssignmentHandler">
            <workingMemoryName>orderApprovalRulesWorkingMemory</workingMemoryName>
            <assertObjects>
                <element>#{actor}</element>
                <element>#{customer}</element>
                <element>#{order}</element>
                <element>#{order.lineItems}</element>
            </assertObjects>
        </assignment>
    </task>
    <transition name="rejected" to="cancelled"/>
    <transition name="approved" to="approved"/>
</task-node>
```

Certain objects are available to the rules as Drools globals, namely the jBPM `Assignable`, as `assignable` and a Seam `Decision` object, as `decision`. Rules which handle decisions should call `decision.setOutcome("result")` to determine the result of the decision. Rules which perform assignments should set the actor id using the `Assignable`.

```
package org.jboss.seam.examples.shop

import org.jboss.seam.drools.Decision

global Decision decision

rule "Approve Order For Loyal Customer"
```

```
  when
    Customer( loyaltyStatus == "GOLD" )
    Order( totalAmount <= 10000 )
  then
    decision.setOutcome("approved");
end
```

```
package org.jboss.seam.examples.shop

import org.jbpm.taskmgmt.exe.Assignable

global Assignable assignable

rule "Assign Review For Small Order"
  when
    Order( totalAmount <= 100 )
  then
    assignable.setPooledActors( new String[] {"reviewers"} );
end
```

Chapter 12. JSF form validation in Seam

In plain JSF, validation is defined in the view:

```
<h:form>
    <div>
        <h:messages/>
    </div>
    <div>
        Country:
        <h:inputText value="#{location.country}" required="true">
            <my:validateCountry/>
        </h:inputText>
    </div>
    <div>
        Zip code:
        <h:inputText value="#{location.zip}" required="true">
            <my:validateZip/>
        </h:inputText>
    </div>
    <div>
        <h:commandButton/>
    </div>
</h:form>
```

In practice, this approach usually violates DRY, since most "validation" actually enforces constraints that are part of the data model, and exist all the way down to the database schema definition. Seam provides support for model-based constraints defined using Hibernate Validator.

Let's start by defining our constraints, on our `Location` class:

```
public class Location {
    private String country;
    private String zip;

    @NotNull
    @Length(max=30)
    public String getCountry() { return country; }
    public void setCountry(String c) { country = c; }

    @NotNull
    @Length(max=6)
    @Pattern("^\d*$")
    public String getZip() { return zip; }
    public void setZip(String z) { zip = z; }
}
```

Well, that's a decent first cut, but in practice it might be more elegant to use custom constraints instead of the ones built into Hibernate Validator:

```
public class Location {
    private String country;
    private String zip;

    @NotNull
    @Country
    public String getCountry() { return country; }
    public void setCountry(String c) { country = c; }

    @NotNull
    @ZipCode
    public String getZip() { return zip; }
    public void setZip(String z) { zip = z; }
}
```

Whichever route we take, we no longer need to specify the type of validation to be used in the JSF page. Instead, we can use `<s:validate>` to validate against the constraint defined on the model object.

```
<h:form>
    <div>
        <h:messages/>
    </div>
    <div>
        Country:
        <h:inputText value="#{location.country}" required="true">
            <s:validate/>
        </h:inputText>
    </div>
    <div>
        Zip code:
        <h:inputText value="#{location.zip}" required="true">
            <s:validate/>
        </h:inputText>
    </div>
    <div>
        <h:commandButton/>
    </div>
</h:form>
```

Note: specifying `@NotNull` on the model does *not* eliminate the requirement for `required="true"` to appear on the control! This is due to a limitation of the JSF validation architecture.

This version is not much less verbose than what we started with, so let's try `<s:validateAll>`:

```
<h:form>
    <div>
        <h:messages/>
    </div>
    <s:validateAll>
        <div>
            Country:
            <h:inputText value="#{location.country}" required="true"/>
        </div>
        <div>
            Zip code:
            <h:inputText value="#{location.zip}" required="true"/>
        </div>
        <div>
            <h:commandButton/>
        </div>
    </s:validateAll>
</h:form>
```

This tag simply adds an `<s:validate>` to every input in the form. For a large form, it can save a lot of typing!

Now we need to do something about displaying feedback to the user when validation fails. Currently we are displaying all messages at the top of the form. What we would really like to do is display the message next to the field with the error (this is possible in plain JSF), highlight the field (this is not possible) and, for good measure, display some image next the the field (also not possible).

Let's try out `<s:decorate>`:

```
<h:form>
    <div>
        <h:messages globalOnly="true"/>
    </div>
    <s:validateAll>
        <div>
            Country:
            <s:decorate>
```

```
                <f:facet name="beforeInvalidField"><h:graphicImage src="img/error.gif"/></f:facet>
                <f:facet name="afterInvalidField"><s:message/></f:facet>
                <f:facet name="aroundInvalidField"><s:span styleClass="error"/></f:facet>
                <h:inputText value="#{location.country}" required="true"/>
            </s:decorate>
        </div>
        <div>
            Zip code:
            <s:decorate>
                <f:facet name="beforeInvalidField"><h:graphicImage src="img/error.gif"/></f:facet>
                <f:facet name="afterInvalidField"><s:message/></f:facet>
                <f:facet name="aroundInvalidField"><s:span styleClass="error"/></f:facet>
                <h:inputText value="#{location.zip}" required="true"/>
            </s:decorate>
        </div>
        <div>
            <h:commandButton/>
        </div>
    </s:validateAll>
</h:form>
```

Well, that looks much better to the user, but it is extremely verbose. Fortunately, the facets of `<s:decorate>` may be defined on any parent element:

```
<h:form>
    <f:facet name="beforeInvalidField">
        <h:graphicImage src="img/error.gif"/>
    </f:facet>
    <f:facet name="afterInvalidField">
        <s:message/>
    </f:facet>
    <f:facet name="aroundInvalidField">
        <s:span styleClass="error"/>
    </f:facet>
    <div>
        <h:messages globalOnly="true"/>
    </div>
    <s:validateAll>
        <div>
            Country:
            <s:decorate>
                <h:inputText value="#{location.country}" required="true"/>
            </s:decorate>
        </div>
        <div>
            Zip code:
            <s:decorate>
                <h:inputText value="#{location.zip}" required="true"/>
            </s:decorate>
        </div>
        <div>
            <h:commandButton/>
        </div>
    </s:validateAll>
</h:form>
```

This approach *defines* constraints on the model, and *presents* constraint violations in the view—a significantly better design.

Finally, we can use Ajax4JSF to display validation messages as the user is typing:

```
<h:form>
    <f:facet name="beforeInvalidField">
        <h:graphicImage src="img/error.gif"/>
    </f:facet>
    <f:facet name="aroundInvalidField">
        <s:span styleClass="error"/>
```

```
    </f:facet>
    <div>
        <h:messages globalOnly="true"/>
    </div>
    <s:validateAll>
        <div>
            Country:
            <s:decorate>
                <h:inputText value="#{location.country}" required="true">
                    <a:support event="onblur" reRender="countryError"/>
                </h:inputText>
                <a:outputPanel id="countryError"><s:message/></a:outputPanel>
            </s:decorate>
        </div>
        <div>
            Zip code:
            <s:decorate>
                <h:inputText value="#{location.zip}" required="true">
                    <a:support event="onblur" reRender="zipError"/>
                </h:inputText>
                <a:outputPanel id="zipError><s:message/></a:outputPanel>
            </s:decorate>
        </div>
        <div>
            <h:commandButton/>
        </div>
    </s:validateAll>
</h:form>
```

Chapter 13. Configuring Seam and packaging Seam applications

Configuration is a very boring topic and an extremely tedious pastime. Unfortunately, several lines of XML are required to integrate Seam into your JSF implementation and servlet container. There's no need to be too put off by the following sections; you'll never need to type any of this stuff yourself, since you can just copy and paste from the example applications!

13.1. Basic Seam configuration

First, let's look at the basic configuration that is needed whenever we use Seam with JSF.

13.1.1. Integrating Seam with JSF and your servlet container

Seam requires the following entry in your `web.xml` file:

```
<listener>
    <listener-class>org.jboss.seam.servlet.SeamListener</listener-class>
</listener>
```

This listener is responsible for bootstrapping Seam, and for destroying session and application contexts.

To integrate with the JSF request lifecycle, we also need a JSF `PhaseListener` registered in in the `faces-config.xml` file:

```
<lifecycle>
    <phase-listener>org.jboss.seam.jsf.SeamPhaseListener</phase-listener>
</lifecycle>
```

The actual listener class here varies depending upon how you want to manage transaction demarcation (more on this below).

If you are using Sun's JSF 1.2 reference implementation, you should add this to `faces-config.xml`:

```
<application>
    <el-resolver>org.jboss.seam.jsf.SeamELResolver</el-resolver>
</application>
```

(This line should not strictly speaking be necessary, but it works around a minor bug in the RI.)

Some JSF implementations have a broken implementation of server-side state saving that interferes with Seam's conversation propagation. If you have problems with conversation propagation during form submissions, try switching to client-side state saving. You'll need this in `web.xml`:

```
<context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>client</param-value>
</context-param>
```

13.1.2. Integrating Seam with your EJB container

We need to apply the `SeamInterceptor` to our Seam components. The simplest way to do this is to add the fol-

lowing interceptor binding to the `<assembly-descriptor>` in `ejb-jar.xml`:

```
<interceptor-binding>
    <ejb-name>*</ejb-name>
    <interceptor-class>org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
</interceptor-binding>
```

Seam needs to know where to go to find session beans in JNDI. One way to do this is specify the `@JndiName` annotation on every session bean Seam component. However, this is quite tedious. A better approach is to specify a pattern that Seam can use to calculate the JNDI name from the EJB name. Unfortunately, there is no standard mapping to global JNDI defined in the EJB3 specification, so this mapping is vendor-specific. We must specify a pattern using the configuration property named `org.jboss.seam.core.init.jndiPattern`. We may specify this using `components.xml`, `web.xml` or even `seam.properties`.

For JBoss AS, the following pattern is correct:

```
<core:init jndi-name="myEarName/#{ejbName}/local" />
```

Or:

```
<context-param>
    <param-name>org.jboss.seam.core.init.jndiPattern</param-name>
    <param-value>myEarName/#{ejbName}/local</param-value>
</context-param>
```

Where `myEarName` is the name of the EAR in which the bean is deployed.

Outside the context of an EAR (when using the JBoss Embeddable EJB3 container), the following pattern is the one to use:

```
<core:init jndi-name="#{ejbName}/local" />
```

Or:

```
<context-param>
    <param-name>org.jboss.seam.core.init.jndiPattern</param-name>
    <param-value>#{ejbName}/local</param-value>
</context-param>
```

13.1.3. Enabling conversation propagation with redirects

If you want to use post-then-redirect in JSF, and you want Seam to propagate the conversation context across the browser redirects, you need to register a servlet filter:

```
<filter>
    <filter-name>Seam Redirect Filter</filter-name>
    <filter-class>org.jboss.seam.servlet.SeamRedirectFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>Seam Redirect Filter</filter-name>
    <url-pattern>*.jsf</url-pattern>
</filter-mapping>
```

This filter intercepts any browser redirects and adds a request parameter that specifies the Seam conversation id.

13.1.4. Using facelets

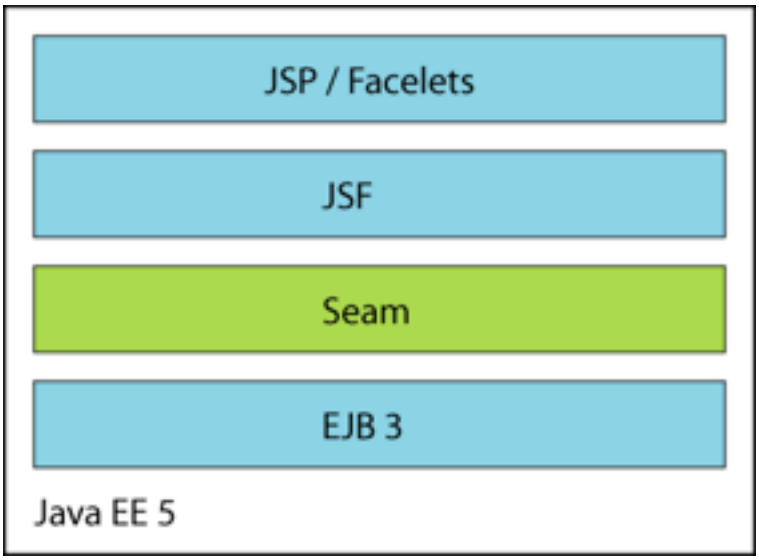
If you want follow our advice and use facelets instead of JSP, add the following lines to `faces-config.xml`:

```
<application>
    <view-handler>com.sun.facelets.FaceletViewHandler</view-handler>
</application>
```

And the following lines to `web.xml`:

```
<context-param>
    <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
    <param-value>.xhtml</param-value>
</context-param>
```

13.2. Configuring Seam in Java EE 5



If you're running in a Java EE 5 environment, this is all the configuration required to start using Seam! But there is one final item you need to know about. You must place a `seam.properties`, `META-INF/seam.properties` or `META-INF/components.xml` file in any archive in which your Seam components are deployed (even an empty properties file will do). At startup, Seam will scan any archives with `seam.properties` files for seam components. If that doesn't work for you, you can also add components by installing them explicitly via `components.xml`. (We don't recommend this alternative approach.)

13.2.1. Packaging

Once you've packaged all this stuff together into an EAR, the archive structure will look something like this:

```
my-application.ear/
    jboss-seam.jar
    el-api.jar
    el-ri.jar
    META-INF/
        MANIFEST.MF
        application.xml
    my-application.war/
        META-INF/
            MANIFEST.MF
        WEB-INF/
            web.xml
            components.xml
```

```
            faces-config.xml
            lib/
                jsf-facelets.jar
                jboss-seam-ui.jar
        login.jsp
        register.jsp
        ...
    my-application.jar/
        META-INF/
            MANIFEST.MF
            persistence.xml
        seam.properties
        org/
            jboss/
                myapplication/
                    User.class
                    Login.class
                    LoginBean.class
                    Register.class
                    RegisterBean.class
                    ...
```

You must include `jboss-seam.jar`, `el-api.jar` and `el-ri.jar` in the EAR classpath. Make sure you reference all of these jars from `application.xml`.

If you want to use jBPM or Drools, you must include the needed jars in the EAR classpath. Make sure you reference all of the jars from `application.xml`.

If you want to use facelets (our recommendation), you must include `jsf-facelets.jar` in the `WEB-INF/lib` directory of the WAR.

If you want to use the Seam tag library (most Seam applications do), you must include `jboss-seam-ui.jar` in the `WEB-INF/lib` directory of the WAR.

If you want to use the Seam debug page (only works for applications using facelets), you must include `jboss-seam-debug.jar` in the `WEB-INF/lib` directory of the WAR.

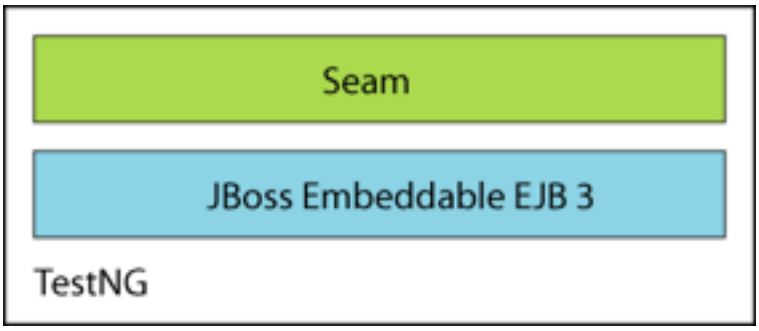
Seam ships with several example applications that are deployable in any Java EE container that supports EJB 3.0.

I really wish that was all there was to say on the topic of configuration but unfortunately we're only about a third of the way there. If you're too overwhelmed by all this tedious configuration stuff, feel free to skip over the rest of this section and come back to it later.

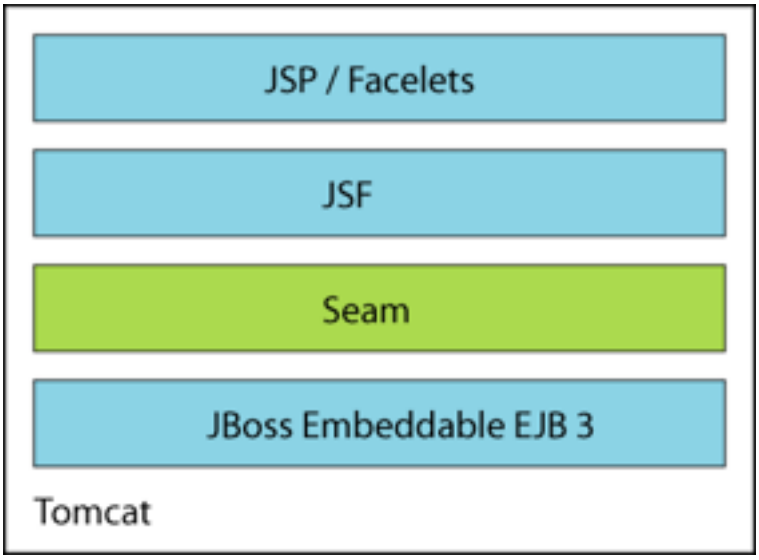
13.3. Configuring Seam with the JBoss Embeddable EJB3 container

The JBoss Embeddable EJB3 container lets you run EJB3 components outside the context of the Java EE 5 application server. This is especially, but not only, useful for testing.

The Seam booking example application includes a TestNG integration test suite that runs on the Embeddable EJB3 container.



The booking example application may even be deployed to Tomcat.



13.3.1. Installing the Embeddable EJB3 container

Seam ships with a build of the Embeddable EJB3 container in the `embedded-ejb` directory. To use the Embeddable EJB3 container with Seam, add the `embedded-ejb/conf` directory, and all jars in the `lib` and `embedded-ejb/lib` directories to your classpath. Then, add the following line to `components.xml`:

```
<core:ejb />
```

This setting installs the built-in component named `org.jboss.seam.core.ejb`. This component is responsible for bootstrapping the EJB container when Seam is started, and shutting it down when the web application is undeployed.

13.3.2. Configuring a datasource with the Embeddable EJB3 container

You should refer to the Embeddable EJB3 container documentation for more information about configuring the container. You'll probably at least need to set up your own datasource. Embeddable EJB3 is implemented using the JBoss Microcontainer, so it's very easy to add new services to the minimal set of services provided by default. For example, I can add a new datasource by putting this `jboss-beans.xml` file in my classpath:

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="urn:jboss:bean-deployer bean-deployer_1_0.xsd"
            xmlns="urn:jboss:bean-deployer">

   <bean name="bookingDatasourceBootstrap"
        class="org.jboss.resource.adapter.jdbc.local.LocalTxDataSource">
      <property name="driverClass">org.hsqldb.jdbcDriver</property>
```

```
      <property name="connectionURL">jdbc:hsqldb:.</property>
      <property name="userName">sa</property>
      <property name="jndiName">java:/bookingDatasource</property>
      <property name="minSize">0</property>
      <property name="maxSize">10</property>
      <property name="blockingTimeout">1000</property>
      <property name="idleTimeout">100000</property>
      <property name="transactionManager">
        <inject bean="TransactionManager"/>
      </property>
      <property name="cachedConnectionManager">
        <inject bean="CachedConnectionManager"/>
      </property>
      <property name="initialContextProperties">
        <inject bean="InitialContextProperties"/>
      </property>
   </bean>

   <bean name="bookingDatasource" class="java.lang.Object">
      <constructor factoryMethod="getDatasource">
         <factory bean="bookingDatasourceBootstrap"/>
      </constructor>
   </bean>

</deployment>
```

13.3.3. Packaging

The archive structure of a WAR-based deployment on an servlet engine like Tomcat will look something like this:

```
my-application.war/
    META-INF/
        MANIFEST.MF
    WEB-INF/
        web.xml
        components.xml
        faces-config.xml
        lib/
            jboss-seam.jar
            jboss-seam-ui.jar
            el-api.jar
            el-ri.jar
            jsf-facelets.jar
            myfaces-api.jar
            myfaces-impl.jar
            jboss-ejb3.jar
            jboss-jca.jar
            jboss-j2ee.jar
            ...
            mc-conf.jar/
                ejb3-interceptors-aop.xml
                embedded-jboss-beans.xml
                default.persistence.properties
                jndi.properties
                login-config.xml
                security-beans.xml
                log4j.xml
            my-application.jar/
                META-INF/
                    MANIFEST.MF
                    persistence.xml
                    jboss-beans.xml
                log4j.xml
                seam.properties
                org/
                    jboss/
                        myapplication/
```

```
                            User.class
                            Login.class
                            LoginBean.class
                            Register.class
                            RegisterBean.class
                            ...
    login.jsp
    register.jsp
    ...
```

The `mc-conf.jar` just contains the standard JBoss Microcontainer configuration files for Embeddable EJB3. You won't usually need to edit these files yourself.

Most of the Seam example applications may be deployed to Tomcat by running `ant deploy.tomcat`.

13.4. Seam managed transactions

EJB session beans feature declarative transaction management. The EJB container is able to start a transaction transparently when the bean is invoked, and end it when the invocation ends. If we write a session bean method that acts as a JSF action listener, we can do all the work associated with that action in one transaction, and be sure that it is committed or rolled back when we finish processing the action. This is a great feature, and all that is needed by many Seam applications.

There is just one problem with this approach. ORM solutions like Hibernate and EJB 3.0 persistence support lazy fetching of entity associations inside a transaction context, but throw `LazyInitializationExceptions` if you try to access an unfetched association outside the context of a transaction. This is a problem if your view page tries to access data that was not fetched during the transaction. Hibernate users developed the *open session in view* pattern to work around this problem. This pattern is usually implemented as a transaction which spans the entire request. There are several problems with this idea, the most serious being that we can't be sure that a transaction has been successful until we commit it, but by the time we commit the transaction, we have already rendered the view. Furthermore, this is at best a partial solution to the problem, because we can still meet the dreaded `LazyInitializationException` if we try to re-use the entity object in the next request.

Seam *completely solves* the problem of unwanted `LazyInitializationExceptions`, while working around the biggest problem in the *open session in view* pattern. The solution comes in two parts:

- use an extended persistence context that is scoped to the conversation, instead of to the request
- use two transactions per request; the first spans the beginning of the update model values phase until the end of the invoke application phase; the second spans the render response phase

13.4.1. Enabling Seam-managed transactions

To make use of *Seam managed transactions*, you need to use `SeamExtendedManagedPersistencePhaseListener` in place of `SeamPhaseListener`.

```
<lifecycle>
     <phase-listener>
        org.jboss.seam.jsf.SeamExtendedManagedPersistencePhaseListener
    </phase-listener>
</lifecycle>
```

It's also a good idea to add a servlet filter to rollback uncommitted transactions when uncaught exceptions occur.

```
<filter>
    <filter-name>Seam Exception Filter</filter-name>
    <filter-class>org.jboss.seam.servlet.SeamExceptionFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>Seam Exception Filter</filter-name>
    <url-pattern>*.jsf</url-pattern>
</filter-mapping>
```

13.4.2. Using a Seam-managed persistence context

You'll need to use a *managed persistence context* (for EJB3) or *managed session* (for Hibernate) in your components. We'll see how to use a managed session later. Configuring a managed persistence context is easy. In `components.xml`, we can write:

```
<core:managed-persistence-context name="bookingDatabase" auto-create="true"
    persistence-unit-jndi-name="java:/EntityManagerFactories/bookingData"/>
```

This configuration creates a conversation-scoped Seam component named `bookingDatabase` that manages the lifecycle of `EntityManager` instances for the persistence unit (`EntityManagerFactory` instance) with JNDI name `java:/EntityManagerFactories/bookingData`.

Of course, you need to make sure that you have bound the `EntityManagerFactory` into JNDI. In JBoss, you can do this by adding the following property setting to `persistence.xml`.

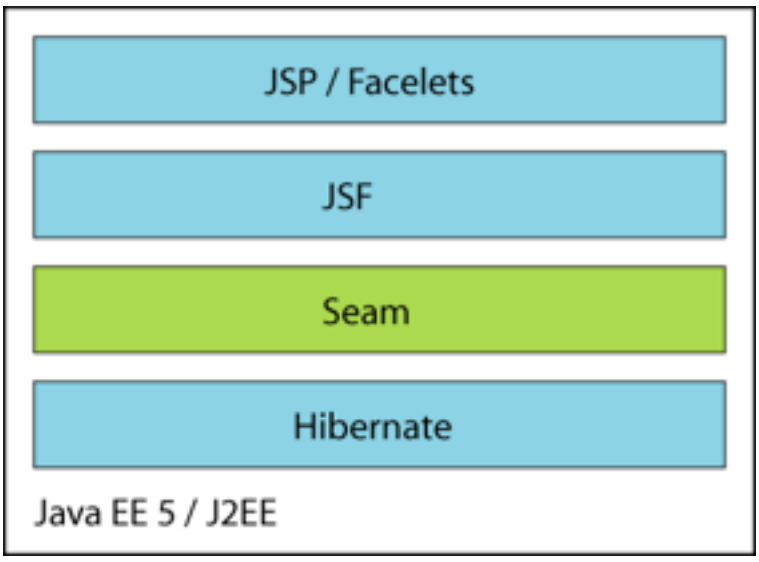
```
<property name="jboss.entity.manager.factory.jndi.name"
          value="java:/EntityManagerFactories/bookingData"/>
```

Now we can have our `EntityManager` injected using:

```
@In EntityManager bookingDatabase;
```

13.5. Configuring Seam with Hibernate in Java EE

Seam is useful even if you're not yet ready to take the plunge into EJB 3.0. In this case you would use Hibernate3 instead of EJB 3.0 persistence, and plain JavaBeans instead of session beans. You'll miss out on some of the nice features of session beans but it will be very easy to migrate to EJB 3.0 when you're ready and, in the meantime, you'll be able to take advantage of Seam's unique declarative state management architecture.



Seam JavaBean components do not provide declarative transaction demarcation like session beans do. You *could* manage your transactions manually using the JTA `UserTransaction` (you could even implement your own declarative transaction management in a Seam interceptor). But most applications will use Seam managed transactions when using Hibernate with JavaBeans. Follow the instructions above to enable `SeamExtendedManagedPersistencePhaseListener`.

The Seam distribution includes a version of the booking example application that uses Hibernate and JavaBeans instead of EJB3. This example application is ready to deploy into any J2EE application server.

13.5.1. Boostrapping Hibernate in Seam

Seam will bootstrap a Hibernate `SessionFactory` from your `hibernate.cfg.xml` file if you install the built-in component named `org.jboss.seam.core.hibernate`.

13.5.2. Using a Seam-managed Hibernate Session

We will also need to configure a *managed session* if we want a Seam managed Hibernate `Session` to be available via injection.

To configure our Seam component, as usual, we use `components.xml`:

```
<core:managed-hibernate-session name="hibernateSessionFactory"/>

<core:managed-hibernate-session name="bookingDatabase" auto-create="true"
    session-factory-jndi-name="java:/bookingSessionFactory"/>
```

Where `java:/bookingSessionFactory` is the name of the session factory specified in `hibernate.cfg.xml`.

```
<session-factory name="java:/bookingSessionFactory">
                <property name="transaction.flush_before_completion">true</property>
                <property name="connection.release_mode">after_statement</property>
                <property name="transaction.manager_lookup_class">org.hibernate.transaction.JBossTrans
                <property name="transaction.factory_class">org.hibernate.transaction.JTATransactionFac
                <property name="connection.datasource">java:/bookingDatasource</property>
    ...
</session-factory>
```

Note that Seam does not flush the session, so you should always enable `hibernate.transaction.flush_before_completion` to ensure that the session is automatically flushed before the JTA transaction commits.

We can now have a managed Hibernate `Session` injected into our JavaBean components using the following code:

```
@In Session bookingDatabase;
```

13.5.3. Packaging

We can package our application as a WAR, in the following structure:

```
my-application.war/
    META-INF/
        MANIFEST.MF
    WEB-INF/
        web.xml
        components.xml
```

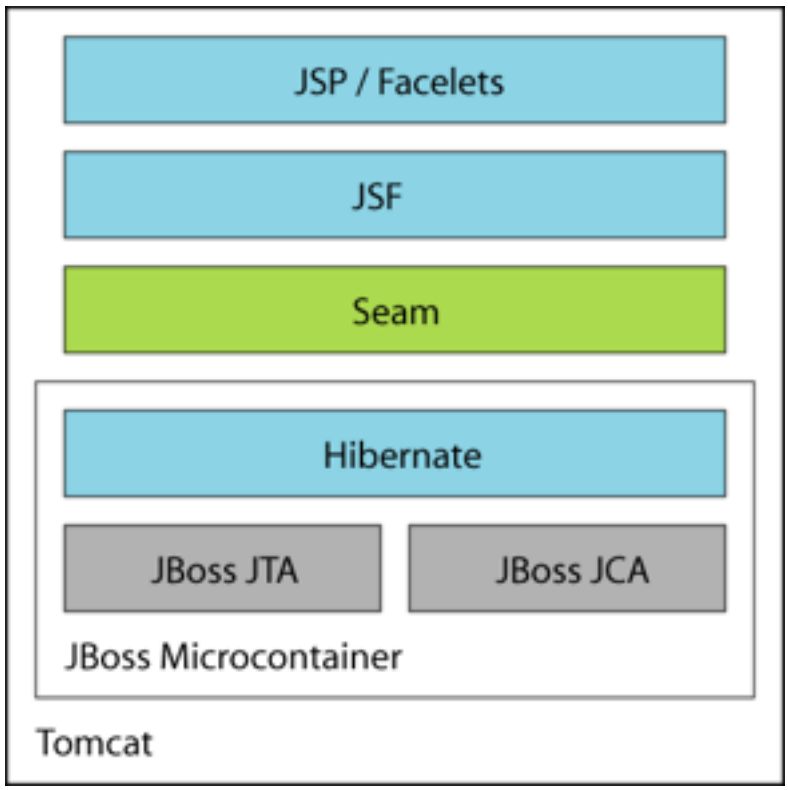
```
        faces-config.xml
        lib/
            jboss-seam.jar
            jboss-seam-ui.jar
            el-api.jar
            el-ri.jar
            jsf-facelets.jar
            hibernate3.jar
            ...
            my-application.jar/
                META-INF/
                   MANIFEST.MF
                seam.properties
                hibernate.cfg.xml
                org/
                    jboss/
                        myapplication/
                            User.class
                            Login.class
                            Register.class
                            ...
    login.jsp
    register.jsp
    ...
```

If we want to deploy Hibernate in a non-J2EE environment like Tomcat or TestNG, we need to do a little bit more work.

13.6. Configuring Seam with Hibernate in Java SE

The Seam support for Hibernate requires JTA and a JCA datasource. If you are running in a non-EE environment like Tomcat or TestNG, you can run these services, and Hibernate itself, in the JBoss Microcontainer.

You can even deploy the Hibernate version of the booking example in Tomcat.



Seam ships with an example Microcontainer configuration in `microcontainer/conf/jboss-beans.xml` that

provides all the things you need to run Seam with Hibernate in any non-EE environment. Just add the `microcontainer/conf` directory, and all jars in the `lib` and `microcontainer/lib` directories to your classpath. Refer to the documentation for the JBoss Microcontainer for more information.

13.6.1. Using Hibernate and the JBoss Microcontainer

The built-in Seam component named `org.jboss.seam.core.microcontainer` bootstraps the microcontainer. As before, we probably want to use a Seam managed session.

```
<core:microcontainer/>

<core:managed-hibernate-session name="bookingDatabase" auto-create="true"
    session-factory-jndi-name="java:/bookingSessionFactory"/>
```

Where `java:/bookingSessionFactory` is the name of the Hibernate session factory specified in `hibernate.cfg.xml`.

You'll need to provide a `jboss.beans.xml` file that installs JNDI, JTA, your JCA datasource and Hibernate into the microcontainer:

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="urn:jboss:bean-deployer bean-deployer_1_0.xsd"
            xmlns="urn:jboss:bean-deployer">

   <bean name="Naming" class="org.jnp.server.SingletonNamingServer"/>

   <bean name="TransactionManagerFactory" class="org.jboss.seam.microcontainer.TransactionManagerFacto
   <bean name="TransactionManager" class="java.lang.Object">
      <constructor factoryMethod="getTransactionManager">
         <factory bean="TransactionManagerFactory"/>
      </constructor>
   </bean>

   <bean name="bookingDatasourceFactory" class="org.jboss.seam.microcontainer.DataSourceFactory">
      <property name="driverClass">org.hsqldb.jdbcDriver</property>
      <property name="connectionUrl">jdbc:hsqldb:.</property>
      <property name="userName">sa</property>
      <property name="jndiName">java:/hibernateDatasource</property>
      <property name="minSize">0</property>
      <property name="maxSize">10</property>
      <property name="blockingTimeout">1000</property>
      <property name="idleTimeout">100000</property>
      <property name="transactionManager"><inject bean="TransactionManager"/></property>
   </bean>
   <bean name="bookingDatasource" class="java.lang.Object">
      <constructor factoryMethod="getDataSource">
         <factory bean="bookingDatasourceFactory"/>
      </constructor>
   </bean>

   <bean name="bookingDatabaseFactory" class="org.jboss.seam.microcontainer.HibernateFactory"/>
   <bean name="bookingDatabase" class="java.lang.Object">
      <constructor factoryMethod="getSessionFactory">
         <factory bean="bookingDatabaseFactory"/>
      </constructor>
      <depends>bookingDatasource</depends>
   </bean>

</deployment>
```

13.6.2. Packaging

The WAR could have the following structure:

```
my-application.war/
    META-INF/
        MANIFEST.MF
    WEB-INF/
        web.xml
        components.xml
        faces-config.xml
        lib/
            jboss-seam.jar
            jboss-seam-ui.jar
            el-api.jar
            el-ri.jar
            jsf-facelets.jar
            hibernate3.jar
            ...
            jboss-microcontainer.jar
            jboss-jca.jar
            ...
            myfaces-api.jar
            myfaces-impl.jar
            mc-conf.jar/
                jndi.properties
                log4j.xml
            my-application.jar/
                META-INF/
                    MANIFEST.MF
                    jboss-beans.xml
                seam.properties
                hibernate.cfg.xml
                log4j.xml
                org/
                    jboss/
                        myapplication/
                            User.class
                            Login.class
                            Register.class
                            ...
    login.jsp
    register.jsp
    ...
```

13.7. Configuring jBPM in Seam

Seam's jBPM integration is not installed by default, so you'll need to enable jBPM by installing a built-in component. You'll also need to explicitly list your process and pageflow definitions. In `components.xml`:

```
<core:jbpm>
    <core:pageflow-definitions>
        <value>createDocument.jpdl.xml</value>
        <value>editDocument.jpdl.xml</value>
        <value>approveDocument.jpdl.xml</value>
    </core:pageflow-definitions>
    <core:process-definitions>
        <value>documentLifecycle.jpdl.xml</value>
    </core:process-definitions>
</core:jbpm>
```

No further special configuration is needed if you only have pageflows. If you do have business process definitions, you need to provide a jBPM configuration, and a Hibernate configuration for jBPM. The Seam DVD Store demo includes example `jbpm.cfg.xml` and `hibernate.cfg.xml` files that will work with Seam:

```
<jbpm-configuration>
```

```
  <jbpm-context>
    <service name="persistence">
       <factory>
          <bean class="org.jbpm.persistence.db.DbPersistenceServiceFactory">
             <field name="isTransactionEnabled"><false/></field>
          </bean>
       </factory>
    </service>
    <service name="message" factory="org.jbpm.msg.db.DbMessageServiceFactory" />
    <service name="scheduler" factory="org.jbpm.scheduler.db.DbSchedulerServiceFactory" />
    <service name="logging" factory="org.jbpm.logging.db.DbLoggingServiceFactory" />
    <service name="authentication" factory="org.jbpm.security.authentication.DefaultAuthenticationServ
  </jbpm-context>

</jbpm-configuration>
```

The most important thing to notice here is that jBPM transaction control is disabled. Seam or EJB3 should control the JTA transactions.

13.7.1. Packaging

There is not yet any well-defined packaging format for jBPM configuration and process/pageflow definition files. In the Seam examples we've decided to simply package all these files into the root of the EAR. In future, we will probably design some other standard packaging format. So the EAR looks something like this:

```
my-application.ear/
    jboss-seam.jar
    el-api.jar
    el-ri.jar
    jbpm-3.1.jar
    META-INF/
        MANIFEST.MF
        application.xml
    my-application.war/
        META-INF/
            MANIFEST.MF
        WEB-INF/
            web.xml
            components.xml
            faces-config.xml
            lib/
                jsf-facelets.jar
                jboss-seam-ui.jar
        login.jsp
        register.jsp
        ...
    my-application.jar/
        META-INF/
            MANIFEST.MF
            persistence.xml
        seam.properties
        org/
            jboss/
                myapplication/
                    User.class
                    Login.class
                    LoginBean.class
                    Register.class
                    RegisterBean.class
                    ...
    jbpm.cfg.xml
    hibernate.cfg.xml
    createDocument.jpdl.xml
    editDocument.jpdl.xml
    approveDocument.jpdl.xml
    documentLifecycle.jpdl.xml
```

Remember to add `jbpm-3.1.jar` to the manifest of your EJB-JAR and WAR.

13.8. Configuring Seam in a Portal

To run a Seam application as a portlet, you'll need to provide certain portlet metadata (`portlet.xml`, etc) in addition to the usual Java EE metadata. See the `examples/portal` directory for an example of the booking demo preconfigured to run on JBoss Portal.

In addition, you'll need to use a portlet-specific phase listener instead of `SeamPhaseListener` or `SeamExtendedManagedPersistencePhaseListener`. The `SeamPortletPhaseListener` and `SeamExtendedManagedPersistencePortletPhaseListener` are adapted to the portlet lifecycle.

Chapter 14. The Seam Application Framework

Seam makes it really easy to create applications by writing plain Java classes with annotations, which don't need to extend any special interfaces or superclasses. But we can simplify some common programming tasks even further, by providing a set of pre-built components which can be re-used either by configuration in `components.xml` (for very simple cases) or extension.

The *Seam Application Framework* can reduce the amount of code you need to write when doing basic database access in a web application, using either Hibernate or JPA.

We should emphasize that the framework is extremely simple, just a handful of simple classes that are easy to understand and extend. The "magic" is in Seam itself—the same magic you use when creating any Seam application even without using this framework.

14.1. Introduction

The components provided by the Seam application framework may be used in one of two different approaches. The first way is to install and configure an instance of the component in `components.xml`, just like we have done with other kinds of built-in Seam components. For example, the following fragment from `components.xml` installs a component which can perform basic CRUD operations for a `Contact` entity:

```
<framework:entity-home name="personHome"
                       entity-class="eg.Person"
                       entity-manager="#{personDatabase}">
    <framework:id>#{param.personId}</framework:id>
</framework:entity-home>
```

If that looks a bit too much like "programming in XML" for your taste, you can use extension instead:

```
@Stateful
@Name("personHome")
public class PersonHome extends EntityHome<Person> implements LocalPersonHome {
    @RequestParameter String personId;
    @In EntityManager personDatabase;

    public Object getId() { return personId; }
    public EntityManager getEntityManager() { return personDatabase; }

}
```

The second approach has one huge advantage: you can easily add extra functionality, and override the built-in functionality (the framework classes were carefully designed for extension and customization).

A second advantage is that your classes may be EJB stateful sessin beans, if you like. (They do not have to be, they can be plain JavaBean components if you prefer.)

At this time, the Seam Application Framework provides just four built-in components: `EntityHome` and `HibernateEntityHome` for CRUD, along with `EntityQuery` and `HibernateEntityQuery` for queries.

The Home and Query components are written so that they can function with a scope of session, event or conversation. Which scope you use depends upon the state model you wish to use in your application.

The Seam Application Framework only works with Seam-managed persistence contexts. By default, the components will look for a persistence context named `entityManager`.

14.2. Home objects

A Home object provides persistence operations for a particular entity class. Suppose we have our trusty `Person` class:

```
@Entity
public class Person {
    @Id private Long id;
    private String firstName;
    private String lastName;
    private Country nationality;

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

We can define a `personHome` component either via configuration:

```
<framework:entity-home name="personHome" entity-class="eg.Person" />
```

Or via extension:

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {}
```

A Home object provides the following operations: `persist()`, `remove()`, `update()` and `getInstance()`. Before you can call the `remove()`, or `update()` operations, you must first set the identifier of the object you are interested in, using the `setId()` method.

We can use a Home directly from a JSF page, for example:

```
<h1>Create Person</h1>
<h:form>
    <div>First name: <h:inputText value="#{personHome.instance.firstName}"/></div>
    <div>Last name: <h:inputText value="#{personHome.instance.lastName}"/></div>
    <div>
        <h:commandButton value="Create Person" action="#{personHome.persist}"/>
    </div>
</h:form>
```

Usually, it is much nicer to be able to refer to the `Person` merely as `person`, so let's make that possible by adding a line to `components.xml`:

```
<factory name="person"
         value="#{personHome.instance}"/>

<framework:entity-home name="personHome"
                       entity-class="eg.Person" />
```

(If we are using configuration.) Or by adding a `@Factory` method to `PersonHome`:

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {

    @Factory("person")
    public Person initPerson() { return getInstance(); }

}
```

(If we are using extension.) This change simplifies our JSF page to the following:

```
<h1>Create Person</h1>
<h:form>
    <div>First name: <h:inputText value="#{person.firstName}"/></div>
    <div>Last name: <h:inputText value="#{person.lastName}"/></div>
    <div>
        <h:commandButton value="Create Person" action="#{personHome.persist}"/>
    </div>
</h:form>
```

Well, that lets us create new `Person` entries. Yes, that is all the code that is required! Now, if we want to be able to display, update and delete pre-existing `Person` entries in the database, we need to be able to pass the entry identifier to the `PersonHome`. Page parameters are a great way to do that:

```
<pages>
    <page viewid="/editPerson.jsp">
        <param name="personId" value="#{personHome.id}"/>
    </page>
</pages>
```

Now we can add the extra operations to our JSF page:

```
<h1>
    <h:outputText rendered="#{!personHome.managed}" value="Create Person"/>
    <h:outputText rendered="#{personHome.managed}" value="Edit Person"/>
</h1>
<h:form>
    <div>First name: <h:inputText value="#{person.firstName}"/></div>
    <div>Last name: <h:inputText value="#{person.lastName}"/></div>
    <div>
        <h:commandButton value="Create Person" action="#{personHome.persist}" rendered="#{!personHome
        <h:commandButton value="Update Person" action="#{personHome.update}" rendered="#{personHome.ma
        <h:commandButton value="Delete Person" action="#{personHome.remove}" rendered="#{personHome.ma
    </div>
</h:form>
```

When we link to the page with no request parameters, the page will be displayed as a "Create Person" page. When we provide a value for the `personId` request parameter, it will be an "Edit Person" page.

Suppose we need to create `Person` entries with their nationality initialized. We can do that easily, via configuration:

```
<factory name="person"
         value="#{personHome.instance}"/>

<framework:entity-home name="personHome"
                       entity-class="eg.Person"
                       new-instance="#{newPerson}"/>

<component name="newPerson"
           class="eg.Person">
    <property name="nationality">#{country}</property>
</component>
```

Or by extension:

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {

    @In Country country;

    @Factory("person")
    public Person initPerson() { return getInstance(); }

    protected Person createInstance() {
```

```
        return new Person(country);
    }

}
```

Of course, the `Country` could be an object managed by another Home object, for example, `CountryHome`.

To add more sophisticated operations (association management, etc), we can just add methods to `PersonHome`.

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {

    @In Country country;

    @Factory("person")
    public Person initPerson() { return getInstance(); }

    protected Person createInstance() {
        return new Person(country);
    }

    public void migrate()
    {
        getInstance().setCountry(country);
        update();
    }

}
```

The Home object automatically displays faces messages when an operation is successful. To customize these messages we can, again, use configuration:

```
<factory name="person"
         value="#{personHome.instance}"/>

<framework:entity-home name="personHome"
                       entity-class="eg.Person"
                       new-instance="#{newPerson}">
    <framework:created-message>New person #{person.firstName} #{person.lastName} created</framework:cr
    <framework:deleted-message>Person #{person.firstName} #{person.lastName} deleted</framework:delete
    <framework:updated-message>Person #{person.firstName} #{person.lastName} updated</framework:update
</framework:entity-home>

<component name="newPerson"
           class="eg.Person">
    <property name="nationality">#{country}</property>
</component>
```

Or extension:

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {

    @In Country country;

    @Factory("person")
    public Person initPerson() { return getInstance(); }

    protected Person createInstance() {
        return new Person(country);
    }

    protected String getCreatedMessage() { return "New person #{person.firstName} #{person.lastName} c
    protected String getUpdatedMessage() { return "Person #{person.firstName} #{person.lastName} updat
    protected String getDeletedMessage() { return "Person #{person.firstName} #{person.lastName} delet

}
```

But the best way to specify the messages is to put them in a resource bundle known to Seam (the bundle named `messages`, by default).

```
Person_created=New person #{person.firstName} #{person.lastName} created
Person_deleted=Person #{person.firstName} #{person.lastName} deleted
Person_updated=Person #{person.firstName} #{person.lastName} updated
```

This enables internationalization, and keeps your code and configuration clean of presentation concerns.

The final step is to add validation functionality to the page, using `<s:validateAll>` and `<s:decorate>`, but I'll leave that for you to figure out.

14.3. Query objects

If we need a list of all `Person` instance in the database, we can use a Query object. For example:

```
<framework:entity-query name="people"
                        ejbql="select p from Person p"/>
```

We can use it from a JSF page:

```
<h1>List of people</h1>
<h:dataTable value="#{people}" var="person">
    <h:column>
        <s:link view-id="/editPerson.jsp" value="#{person.firstName} #{person.lastName}">
            <f:param name="personId" value="#{person.id}"/>
        </s:link>
    </h:column>
</h:dataTable>
```

We probably need to support pagination:

```
<framework:entity-query name="people"
                        ejbql="select p from Person p"
                        order="lastName"
                        max-results="20"/>
```

We'll use a page parameter to determine the page to display:

```
<pages>
    <page viewid="/searchPerson.jsp">
        <param name="firstResult" value="#{people.firstResult}"/>
    </page>
</pages>
```

The JSF code for a pagination control is a bit verbose, but manageable:

```
<h1>Search for people</h1>
<h:dataTable value="#{people.resultList}" var="person">
    <h:column>
        <s:link view-id="/editPerson.jsp" value="#{person.firstName} #{person.lastName}">
            <f:param name="personId" value="#{person.id}"/>
        </s:link>
    </h:column>
</h:dataTable>

<s:link view="/search.xhtml" rendered="#{people.previousExists}" value="First Page">
    <f:param name="firstResult" value="0"/>
</s:link>
```

```
<s:link view="/search.xhtml" rendered="#{people.previousExists}" value="Previous Page">
    <f:param name="firstResult" value="#{people.previousFirstResult}"/>
</s:link>

<s:link view="/search.xhtml" rendered="#{people.nextExists}" value="Next Page">
    <f:param name="firstResult" value="#{people.nextFirstResult}"/>
</s:link>

<s:link view="/search.xhtml" rendered="#{people.nextExists}" value="Last Page">
    <f:param name="firstResult" value="#{people.lastFirstResult}"/>
</s:link>
```

Real search screens let the user enter a bunch of optional search criteria to narrow the list of results returned. The Query object lets you specify optional "restrictions" to support this important usecase:

```
<component name="examplePerson" class="Person"/>

<framework:entity-query name="people"
                        ejbql="select p from Person p"
                        order="lastName"
                        max-results="20">
    <framework:restrictions>
        <value>lower(firstName) like lower( #{examplePerson.firstName} + '%' )</value>
        <value>lower(lastName) like lower( #{examplePerson.lastName} + '%' )</value>
    </framework:restrictions>
</framework:entity-query>
```

Notice the use of an "example" object.

```
<h1>Search for people</h1>
<h:form>
    <div>First name: <h:inputText value="#{examplePerson.firstName}"/></div>
    <div>Last name: <h:inputText value="#{examplePerson.lastName}"/></div>
    <div><h:commandButton value="Search" action="/search.jsp"/></div>
</h:form>

<h:dataTable value="#{people.resultList}" var="person">
    <h:column>
        <s:link view-id="/editPerson.jsp" value="#{person.firstName} #{person.lastName}">
            <f:param name="personId" value="#{person.id}"/>
        </s:link>
    </h:column>
</h:dataTable>
```

The examples in this section have all shown reuse by configuration. However, reuse by extension is equally possible for Query objects.

14.4. Using Hibernate filters

The coolest, and most unique, feature of Hibernate is *filters*. Filters let you provide a restricted view of the data in the database. You can find out more about filters in the Hibernate documentation. But we thought we'd mention an easy way to incorporate filters into a Seam application, one that works especially well with the Seam Application Framework.

Seam-managed persistence contexts may have a list of filters defined, which will be enabled whenever an `EntityManager` or Hibernate `Session` is first created. (Of course, they may only be used when Hibernate is the underlying persistence provider.)

```
<core:filter name="regionFilter">
    <core:name>region</core:name>
    <core:parameters>
```

```
        <key>regionCode</key>
        <value>#{region.code}</value>
    </core:parameters>
</core:filter>

<core:filter name="currentFilter">
    <core:name>current</core:name>
    <core:parameters>
        <key>date</key>
        <value>#{currentDate}</value>
    </core:parameters>
</core:filter>

<core:managed-persistence-context name="personDatabase"
    persistence-unit-jndi-name="java:/EntityManagerFactories/personDatabase">
    <core:filters>
        <value>#{regionFilter}</value>
        <value>#{currentFilter}</value>
    </core:filters>
</core:managed-persistence-context>
```

Chapter 15. Seam Text

Collaboration-oriented websites require a human-friendly markup language for easy entry of formatted text in forum posts, wiki pages, blogs, comments, etc. Seam provides the `<s:formattedText/>` control for display of formatted text that conforms to the *Seam Text* language. Seam Text is implemented using an ANTLR-based parser. You don't need to know anything about ANTLR to use it, however.

15.1. Basic fomatting

Here is a simple example:

```
It's easy to make *bold text*, /italic text/, |monospace|,
-deleted text-, super^scripts^ or _underlines_.
```

If we display this using `<s:formattedText/>`, we will get the following HTML produced:

```
<p>
It's easy to make <b>bold text</b>, <i>italic text</i>, <tt>monospace</tt>
<del>deleted text</del>, super<sup>scripts</sup> or <u>underlines</u>.
</p>
```

We can use a blank line to indicate a new paragraph, and + to indicate a heading:

```
+This is a big heading
You /must/ have some text following a heading!

++This is a smaller heading
This is the first paragraph. We can split it across multiple
lines, but we must end it with a blank line.

This is the second paragraph.
```

This is the HTML that results:

```
<h1>This is a big heading</h1>
<p>
You <i>must</i> have some text following a heading!
</p>

<h2>This is a smaller heading</h2>
<p>
This is the first paragraph. We can split it across multiple
lines, but we must end it with a blank line.
</p>

<p>
This is the second paragraph.
</p>
```

Ordered lists are created using the # character. Unordered lists use the = character:

```
An ordered list:

#first item
#second item
#and even the /third/ item

An unordered list:

=an item
```

```
=another item
```

```
<p>
An ordered list:
</p>

<ol>
<li>first item</li>
<li>second item</li>
<li>and even the <i>third</i> item</li>
</ol>

<p>
An unordered list:
</p>

<ul>
<li>an item</li>
<li>another item</li>
</ul>
```

Quoted sections should be surrounded in double quotes:

```
The other guy said:

"Nyeah nyeah-nee
/nyeah/ nyeah!"

But what do you think he means by "nyeah-nee"?
```

```
<p>
The other guy said:
</p>

<quote>Nyeah nyeah-nee
<i>nyeah</i> nyeah!</quote>

<p>
But what do you think he means by <quote>nyeah-nee</quote>?
</p>
```

15.2. Entering code and text with special characters

Special characters such as `*`, `|` and `#`, along with HTML characters such as `<`, `>` and `&` may be escaped using `\`:

```
You can write down equations like 2\*3\=6 and HTML tags
like \<body\> using the escape character: \\.
```

```
<p>
You can write down equations like 2*3=6 and HTML tags
like &lt;body&gt; using the escape character: \.
</p>
```

And we can quote code blocks using backticks:

```
My code doesn't work:

`for (int i=0; i<100; i--)
{
    doSomething();
}`
```

```
Any ideas?
```

```
<p>
My code doesn't work:
</p>

<pre>for (int i=0; i&lt;100; i--)
{
    doSomething();
}</pre>

<p>
Any ideas?
</p>
```

15.3. Entering HTML

Text may even include a certain limited subset of HTML (don't worry, the subset is chosen to be safe from cross-site scripting attacks). This is useful for creating links:

```
You might want to link to <a href="http://jboss.com/products/seam">something
cool</a>, or even include an image: <img src="/logo.jpg"/>
```

And for creating tables:

```
<table>
    <tr><td>First name:</td><td>Gavin</td></tr>
    <tr><td>Last name:</td><td>King</td></tr>
</table>
```

But you can do much more if you want!

Chapter 16. Seam annotations

When you write a Seam application, you'll use a lot of annotations. Seam lets you use annotations to achieve a declarative style of programming. Most of the annotations you'll use are defined by the EJB 3.0 specification. The annotations for data validation are defined by the Hibernate Validator package. Finally, Seam defines its own set of annotations, which we'll describe in this chapter.

All of these annotations are defined in the package `org.jboss.seam.annotations`.

16.1. Annotations for component definition

The first group of annotations lets you define a Seam component. These annotations appear on the component class.

`@Name`

```
@Name("componentName")
```

Defines the Seam component name for a class. This annotation is required for all Seam components.

`@Scope`

```
@Scope(ScopeType.CONVERSATION)
```

Defines the default context of the component. The possible values are defined by the `ScopeType` enumeration: `EVENT, PAGE, CONVERSATION, SESSION, BUSINESS_PROCESS, APPLICATION, STATELESS`.

When no scope is explicitly specified, the default depends upon the component type. For stateless session beans, the default is `STATELESS`. For entity beans and stateful session beans, the default is `CONVERSATION`. For JavaBeans, the default is `EVENT`.

`@Role`

```
@Role(name="roleName", scope=ScopeType.SESSION)
```

Allows a Seam component to be bound to multiple contexts variables. The `@Name`/`@Scope` annotations define a "default role". Each `@Role` annotation defines an additional role.

- `name` — the context variable name.
- `scope` — the context variable scope. When no scope is explicitly specified, the default depends upon the component type, as above.

`@Roles`

```
@Roles({
        @Role(name="user", scope=ScopeType.CONVERSATION),
        @Role(name="currentUser", scope=ScopeType.SESSION)
    })
```

Allows specification of multiple additional roles.

`@Intercept`

```
@Intercept(InterceptionType.ALWAYS)
```

Determines when Seam interceptors are active. The possible values are defined by the `InterceptionType` enumeration: `ALWAYS`, `AFTER_RESTORE_VIEW`, `AFTER_UPDATE_MODEL_VALUES`, `INVOKE_APPLICATION`, `NEVER`.

When no interception type is explicitly specified, the default depends upon the component type. For entity beans, the default is `NEVER`. For session beans, message driven beans and JavaBeans, the default is `ALWAYS`.

`@JndiName`

```
@JndiName("my/jndi/name")
```

Specifies the JNDI name that Seam will use to look up the EJB component. If no JNDI name is explicitly specified, Seam will use the JNDI pattern specified by `org.jboss.seam.core.init.jndiPattern`.

`@Conversational`

```
@Conversational(ifNotBegunOutcome="error")
```

Specifies that a conversation scope component is conversational, meaning that no method of the component can be called unless a long-running conversation started by this component is active (unless the method would begin a new long-running conversation).

`@Startup`

```
@Startup(depends={"org.jboss.core.jndi", "org.jboss.core.jta"})
```

Specifies that an application scope component is started immediately at initialization time. This is mainly used for certain built-in components that bootstrap critical infrastructure such as JNDI, datasources, etc.

```
@Startup
```

Specifies that a session scope component is started immediately at session creation time.

- `depends` — specifies that the named components must be started first, if they are installed.

`@Install`

```
@Install(false)
```

Specifies whether or not a component should be installed by default. The lack of an @Install annotation indicates a component should be installed.

```
@Install(dependencies="org.jboss.seam.core.jbpm")
```

Specifies that a component should only be stalled if the components listed as dependencies are also installed.

```
@Install(genericDependencies=ManagedQueueSender.class)
```

Specifies that a component should only be installed if a component that is implemented by a certain class is

installed. This is useful when the dependency doesn't have a single well-known name.

```
@Install(precedence=BUILT_IN)
```

Specifies the precedence of the component. If multiple components with the same name exist, the one with the higher precedence will be installed. The defined precendence values are:

- `BUILT_IN` — Precedence of all built-in Seam components
- `FRAMEWORK` — Precedence to use for components of frameworks which extend Seam
- `APPLICATION` — Predence of application components (the default precedence)
- `DEPLOYMENT` — Precedence to use for components which override application components in a particular deployment

`@Synchronized`

```
@Synchronized(timeout=1000)
```

Specifies that a component is accessed concurrently by multiple clients, and that Seam should serialize requests. If a request is not able to obtain its lock on the component in the given timeout period, an exception will be raised.

`@ReadOnly`

```
@ReadOnly
```

Specifies that a JavaBean component or component method does not require state replication at the end of the invocation.

16.2. Annotations for bijection

The next two annotations control bijection. These attributes occur on component instance variables or property accessor methods.

`@In`

```
@In
```

Specifies that a component attribute is to be injected from a context variable at the beginning of each component invocation. If the context variable is null, an exception will be thrown.

```
@In(required=false)
```

Specifies that a component attribute is to be injected from a context variable at the beginning of each component invocation. The context variable may be null.

```
@In(create=true)
```

Specifies that a component attribute is to be injected from a context variable at the beginning of each component invocation. If the context variable is null, an instance of the component is instantiated by Seam.

```
@In(value="contextVariableName")
```

Specifies the name of the context variable explicitly, instead of using the annotated instance variable name.

```
@In(value="#{customer.addresses['shipping']}")
```

Specifies that a component attribute is to be injected by evaluating a JSF EL expression at the beginning of each component invocation.

- `value` — specifies the name of the context variable. Default to the name of the component attribute. Alternatively, specifies a JSF EL expression, surrounded by `#{...}`.
- `create` — specifies that Seam should instantiate the component with the same name as the context variable if the context variable is undefined (null) in all contexts. Default to false.
- `required` — specifies Seam should throw an exception if the context variable is undefined in all contexts.

`@Out`

```
@Out
```

Specifies that a component attribute that is a Seam component is to be outjected to its context variable at the end of the invocation. If the attribute is null, an exception is thrown.

```
@Out(required=false)
```

Specifies that a component attribute that is a Seam component is to be outjected to its context variable at the end of the invocation. The attribute may be null.

```
@Out(scope=ScopeType.SESSION)
```

Specifies that a component attribute that is *not* a Seam component type is to be outjected to a specific scope at the end of the invocation.

Alternatively, if no scope is explicitly specified, the scope of the component with the `@Out` attribute is used (or the `EVENT` scope if the component is stateless).

```
@Out(value="contextVariableName")
```

Specifies the name of the context variable explicitly, instead of using the annotated instance variable name.

- `value` — specifies the name of the context variable. Default to the name of the component attribute.
- `required` — specifies Seam should throw an exception if the component attribute is null during outjection.

Note that it is quite common for these annotations to occur together, for example:

```
@In(create=true) @Out private User currentUser;
```

The next annotation supports the *manager component* pattern, where a Seam component that manages the lifecycle of an instance of some other class that is to be injected. It appears on a component getter method.

`@Unwrap`

```
@Unwrap
```

Specifies that the object returned by the annotated getter method is the thing that is injected instead of the component instance itself.

The next annotation supports the *factory component* pattern, where a Seam component is responsible for initializing the value of a context variable. This is especially useful for initializing any state needed for rendering the response to a non-faces request. It appears on a component method.

`@Factory`

```
@Factory("processInstance")
```

Specifies that the method of the component is used to initialize the value of the named context variable, when the context variable has no value. This style is used with methods that return `void`.

```
@Factory("processInstance", scope=CONVERSATION)
```

Specifies that the method returns a value that Seam should use to initialize the value of the named context variable, when the context variable has no value. This style is used with methods that return a value. If no scope is explicitly specified, the scope of the component with the `@Factory` method is used (unless the component is stateless, in which case the `EVENT` context is used).

- `value` — specifies the name of the context variable. If the method is a getter method, default to the JavaBeans property name.
- `scope` — specifies the scope that Seam should bind the returned value to. Only meaningful for factory methods which return a value.

This annotation lets you inject a `Log`:

`@Logger`

```
@Logger("categoryName")
```

Specifies that a component field is to be injected with an instance of `org.jboss.seam.log.Log`.

- `value` — specifies the name of the log category. Default to the name of the component class.

The last annotation lets you inject a request parameter value:

`@RequestParameter`

```
@RequestParameter("parameterName")
```

Specifies that a component attribute is to be injected with the value of a request parameter. Basic type conversions are performed automatically.

- `value` — specifies the name of the request parameter. Default to the name of the component attribute.

16.3. Annotations for component lifecycle methods

These annotations allow a component to react to its own lifecycle events. They occur on methods of the component. There may be only one of each per component class.

`@Create`

```
@Create
```

Specifies that the method should be called when an instance of the component is instantiated by Seam. Note that create methods are only supported for JavaBeans and stateful session beans.

`@Destroy`

```
@Destroy
```

Specifies that the method should be called when the context ends and its context variables are destroyed. Note that create methods are only supported for JavaBeans and stateful session beans.

Note that all stateful session bean components *must* define a method annotated `@Destroy @Remove` in order to guarantee destruction of the stateful bean when a context ends.

Destroy methods should be used only for cleanup. *Seam catches, logs and swallows any exception that propagates out of a destroy method.*

`@Observer`

```
@Observer("somethingChanged")
```

Specifies that the method should be called when a component-driven event of the specified type occurs.

16.4. Annotations for context demarcation

These annotations provide declarative conversation demarcation. They appear on methods of Seam components, usually action listener methods.

Every web request has a conversation context associated with it. Most of these conversations end at the end of the request. If you want a conversation that span multiple requests, you must "promote" the current conversation to a *long-running conversation* by calling a method marked with `@Begin`.

`@Begin`

```
@Begin
```

Specifies that a long-running conversation begins when this method returns a non-null outcome without exception.

```
@Begin(ifOutcome={"success", "continue"})
```

Specifies that a long-running conversation begins when this action listener method returns with one of the given outcomes.

```
@Begin(join=true)
```

Specifies that if a long-running conversation is already in progress, the conversation context is simply propagated.

```
@Begin(nested=true)
```

Specifies that if a long-running conversation is already in progress, a new *nested* conversation context begins. The nested conversation will end when the next `@End` is encountered, and the outer conversation will resume. It is perfectly legal for multiple nested conversations to exist concurrently in the same outer conversation.

```
@Begin(pageflow="process definition name")
```

Specifies a jBPM process definition name that defines the pageflow for this conversation.

```
@Begin(flushMode=FlushModeType.MANUAL)
```

Specify the flush mode of any Seam-managed persistence contexts. `flushMode=FlushModeType.MANUAL` supports the use of *atomic conversations* where all write operations are queued in the conversation context until an explicit call to `flush()` (which usually occurs at the end of the conversation).

- `ifOutcome` — specifies the JSF outcome or outcomes that result in a new long-running conversation context.
- `join` — determines the behavior when a long-running conversation is already in progress. If `true`, the context is propagated. If `false`, an exception is thrown. Default to `false`. This setting is ignored when `nested=true` is specified
- `nested` — specifies that a nested conversation should be started if a long-running conversation is already in progress.
- `flushMode` — set the flush mode of any Seam-managed Hibernate sessions or JPA persistence contexts that are created during this conversation.
- `pageflow` — a process definition name of a jBPM process definition deployed via `org.jboss.seam.core.jbpm.pageflowDefinitions`.

`@End`

```
@End
```

Specifies that a long-running conversation ends when this method returns a non-null outcome without exception.

```
@End(ifOutcome={"success", "error"}, evenIfException={SomeException.class, OtherException.class})
```

Specifies that a long-running conversation ends when this action listener method returns with one of the given outcomes or throws one of the specified classes of exception.

- `ifOutcome` — specifies the JSF outcome or outcomes that result in the end of the current long-running conversation.
- `beforeRedirect` — by default, the conversation will not actually be destroyed until after any redirect has occurred. Setting `beforeRedirect=true` specifies that the conversation should be destroyed at the end of the current request, and that the redirect will be processed in a new temporary conversation context.

`@StartTask`

```
@StartTask
```

"Starts" a jBPM task. Specifies that a long-running conversation begins when this method returns a non-null outcome without exception. This conversation is associated with the jBPM task specified in the named request parameter. Within the context of this conversation, a business process context is also defined, for the business process instance of the task instance.

The jBPM `TaskInstance` will be available in a request context variable named `taskInstance`. The jPBM `ProcessInstance` will be available in a request context variable named `processInstance`. (Of course, these objects are available for injection via `@In`.)

- `taskIdParameter` — the name of a request parameter which holds the id of the task. Default to `"taskId"`, which is also the default used by the Seam `taskList` JSF component.
- `flushMode` — set the flush mode of any Seam-managed Hibernate sessions or JPA persistence contexts that are created during this conversation.

`@BeginTask`

```
@BeginTask
```

Resumes work on an incomplete jBPM task. Specifies that a long-running conversation begins when this method returns a non-null outcome without exception. This conversation is associated with the jBPM task specified in the named request parameter. Within the context of this conversation, a business process context is also defined, for the business process instance of the task instance.

The jBPM `TaskInstance` will be available in a request context variable named `taskInstance`. The jPBM `ProcessInstance` will be available in a request context variable named `processInstance`.

- `taskIdParameter` — the name of a request parameter which holds the id of the task. Default to `"taskId"`, which is also the default used by the Seam `taskList` JSF component.
- `flushMode` — set the flush mode of any Seam-managed Hibernate sessions or JPA persistence contexts that are created during this conversation.

`@EndTask`

```
@EndTask
```

"Ends" a jBPM task. Specifies that a long-running conversation ends when this method returns a non-null outcome, and that the current task is complete. Triggers a jBPM transition. The actual transition triggered will be the default transition unless the application has called `Transition.setName()` on the built-in component named `transition`.

```
@EndTask(transition="transitionName")
```

Triggers the given jBPM transition.

```
@EndTask(ifOutcome={"success", "continue"})
```

Specifies that the task ends when this method returns one of the listed outcomes.

- `transition` — the name of the jBPM transition to be triggered when ending the task. Defaults to the default transition.
- `ifOutcome` — specifies the JSF outcome or outcomes that result in the end of the task.
- `beforeRedirect` — by default, the conversation will not actually be destroyed until after any redirect has occurred. Setting `beforeRedirect=true` specifies that the conversation should be destroyed at the end of the current request, and that the redirect will be processed in a new temporary conversation context.

`@CreateProcess`

```
@CreateProcess(definition="process definition name")
```

Creates a new jBPM process instance when the method returns a non-null outcome without exception. The `ProcessInstance` object will be available in a context variable named `processInstance`.

- `definition` — the name of the jBPM process definition deployed via `org.jboss.seam.core.jbpm.processDefinitions`.

`@ResumeProcess`

```
@ResumeProcess(processIdParameter="processId")
```

Re-enters the scope of an existing jBPM process instance when the method returns a non-null outcome without exception. The `ProcessInstance` object will be available in a context variable named `processInstance`.

- `processIdParameter` — the name a request parameter holding the process id. Default to `"processId"`.

16.5. Annotations for transaction demarcation

Seam provides an annotation that lets you force a rollback of the JTA transaction for certain action listener outcomes.

`@Rollback`

```
@Rollback(ifOutcome={"failure", "not-found"})
```

If the outcome of the method matches any of the listed outcomes, or if no outcomes are listed, set the transaction to rollback only when the method completes.

- `ifOutcome` — the JSF outcomes that cause a transaction rollback (no outcomes is interpreted to mean any outcome).

`@Transactional`

```
@Transactional
```

Specifies that a JavaBean component should have a similar transactional behavior to the default behavior of a session bean component. ie. method invocations should take place in a transaction, and if no transaction exists when the method is called, a transaction will be started just for that method. This annotation may be applied at either class or method level.

Seam applications usually use the standard EJB3 annotations for all other transaction demarcation needs.

16.6. Annotations for exceptions

These annotations let you specify how Seam should handle an exception that propagates out of a Seam component.

`@Redirect`

```
@Redirect(viewId="error.jsp")
```

Specifies that the annotated exception causes a browser redirect to a specified view id.

- `viewId` — specifies the JSF view id to redirect to.
- `message` — a message to be displayed, default to the exception message.
- `end` — specifies that the long-running conversation should end, default to `false`.

`@Render`

```
@Render(viewId="error.jsp")
```

Specifies that the annotated exception causes immediate rendering of the view. This annotation is ignored unless the exception is thrown during the JSF INVOKE_APPLICATION phase.

- `viewId` — specifies the JSF view id to redirect to.
- `message` — a message to be displayed, default to the exception message.
- `end` — specifies that the long-running conversation should end, default to `false`.

`@HttpError`

```
@HttpError(errorCode=404)
```

Specifies that the annotated exception causes a HTTP error to be sent.

- `errorCode` — the HTTP error code, default to `500`.

- `message` — a message to be sent with the HTTP error, default to the exception message.
- `end` — specifies that the long-running conversation should end, default to `false`.

16.7. Annotations for validation

This annotation triggers Hibernate Validator. It appears on a method of a Seam component, almost always an action listener method.

Please refer to the documentation for the Hibernate Annotations package for information about the annotations defined by the Hibernate Validator framework.

Note that use of `@IfInvalid` is now semi-deprecated and `<s:validateAll>` is now preferred.

`@IfInvalid`

```
@IfInvalid(outcome="invalid", refreshEntities=true)
```

Specifies that Hibernate Validator should validate the component before the method is invoked. If the invocation fails, the specified outcome will be returned, and the validation failure messages returned by Hibernate Validator will be added to the `FacesContext`. Otherwise, the invocation will proceed.

- `outcome` — the JSF outcome when validation fails.
- `refreshEntities` — specifies that any invalid entity in the managed state should be refreshed from the database when validation fails. Default to `false`. (Useful with extended persistence contexts.)

16.8. Annotations for Seam Remoting

Seam Remoting requires that the local interface of a session bean be annotated with the following annotation:

`@WebRemote`

```
@WebRemote(exclude="path.to.exclude")
```

Indicates that the annotated method may be called from client-side JavaScript. The `exclude` property is optional and allows objects to be excluded from the result's object graph (see the Remoting chapter for more details).

16.9. Annotations for Seam interceptors

The following annotations appear on Seam interceptor classes.

Please refer to the documentation for the EJB 3.0 specification for information about the annotations required for EJB interceptor definition.

`@Interceptor`

```
@Interceptor(stateless=true)
```

Specifies that this interceptor is stateless and Seam may optimize replication.

```
@Interceptor(type=CLIENT)
```

Specifies that this interceptor is a "client-side" interceptor that is called before the EJB container.

```
@Interceptor(around={SomeInterceptor.class, OtherInterceptor.class})
```

Specifies that this interceptor is positioned higher in the stack than the given interceptors.

```
@Interceptor(within={SomeInterceptor.class, OtherInterceptor.class})
```

Specifies that this interceptor is positioned deeper in the stack than the given interceptors.

16.10. Annotations for asynchronicity

The following annotations are used to declare an asynchronous method, for example:

```
@Asynchronous public void scheduleAlert(Alert alert, @Expiration Date date) { ... }
```

```
@Asynchronous public Timer scheduleAlerts(Alert alert, @Expiration Date date, @IntervalDuration long i
```

`@Asynchronous`

```
@Asynchronous
```

Specifies that the method call is processed asynchronously.

`@Duration`

```
@Duration
```

Specifies that a parameter of the asynchronous call is the duration before the call is processed (or first processed for recurring calls).

`@Expiration`

```
@Expiration
```

Specifies that a parameter of the asynchronous call is the datetime at which the call is processed (or first processed for recurring calls).

`@IntervalDuration`

```
@IntervalDuration
```

Specifies that an asynchronous method call recurs, and that the annotationed parameter is duration between recurrences.

16.11. Annotations for use with JSF `dataTable`

The following annotations make it easy to implement clickable lists backed by a stateful session bean. They appear on attributes.

`@DataModel`

```
@DataModel("variableName")
```

Exposes an attribute of type `List`, `Map`, `Set` or `Object[]` as a JSF `DataModel` into the scope of the owning component (or the `EVENT` scope if the owning component is `STATELESS`). In the case of `Map`, each row of the `DataModel` is a `Map.Entry`.

- `value` — name of the conversation context variable. Default to the attribute name.
- `scope` — if `scope=ScopeType.PAGE` is explicitly specified, the `DataModel` will be kept in the `PAGE` context.

`@DataModelSelection`

```
@DataModelSelection
```

Injects the selected value from the JSF `DataModel` (this is the element of the underlying collection, or the map value).

- `value` — name of the conversation context variable. Not needed if there is exactly one `@DataModel` in the component.

`@DataModelSelectionIndex`

```
@DataModelSelectionIndex
```

Exposes the selection index of the JSF `DataModel` as an attribute of the component (this is the row number of the underlying collection, or the map key).

- `value` — name of the conversation context variable. Not needed if there is exactly one `@DataModel` in the component.

16.12. Meta-annotations for databinding

These meta-annotations make it possible to implement similar functionality to `@DataModel` and `@DataModelSelection` for other datastructures apart from lists.

`@DataBinderClass`

```
@DataBinderClass(DataModelBinder.class)
```

Specifies that an annotation is a databinding annotation.

`@DataSelectorClass`

```
@DataSelectorClass(DataModelSelector.class)
```

Specifies that an annotation is a dataselection annotation.

16.13. Annotations for packaging

This annotation provides a mechanism for declaring information about a set of components that are packaged together. It can be applied to any Java package.

`@Namespace`

```
@Namespace(value="http://jboss.com/products/seam/example/seampay")
```

Specifies that components in the current package are associated with the given namespace. The declared namespace can be used as an XML namespace in a `components.xml` file to simplify application configuration.

```
@Namespace(value="http://jboss.com/products/seam/core", prefix="org.jboss.seam.core")
```

Specifies a namespace to associate with a given package. Additionally, it specifies a component name prefix to be applied to component names specified in the XML file. For example, an XML element named `microcontainer` that is associated with this namespace would be understood to actually refere to a component named `org.jboss.seam.core.microcontainer`.

Chapter 17. Built-in Seam components

This chapter describes Seam's built-in components, and their configuration properties.

Note that you can replace any of the built in components with your own implementations simply by specifying the name of one of the built in components on your own class using `@Name`.

Note also that even though all the built in components use a qualified name, most of them are aliased to unqualified names by default. These aliases specify `auto-create="true"`, so you do not need to use `create=true` when injecting built-in components by their unqualified name.

17.1. Context injection components

The first set of built in components exist purely to support injection of various contextual objects. For example, the following component instance variable would have the Seam session context object injected:

```
@In private Context sessionContext;
```

`org.jboss.seam.core.eventContext`
: Manager component for the event context object

`org.jboss.seam.core.pageContext`
: Manager component for the page context object

`org.jboss.seam.core.conversationContext`
: Manager component for the conversation context object

`org.jboss.seam.core.sessionContext`
: Manager component for the session context object

`org.jboss.seam.core.applicationContext`
: Manager component for the appication context object

`org.jboss.seam.core.businessProcessContext`
: Manager component for the business process context object

`org.jboss.seam.core.facesContext`
: Manager component for the `FacesContext` context object (not a true Seam context)

All of these components are always installed.

17.2. Utility components

These components are merely useful.

`org.jboss.seam.core.facesMessages`
: Allows faces success messages to propagate across a browser redirect.

- `add(FacesMessage facesMessage)` — add a faces message, which will be displayed during the next render response phase that occurs in the current conversation.

- `add(String messageTemplate)` — add a faces message, rendered from the given message template which may contain EL expressions.
- `add(Severity severity, String messageTemplate)` — add a faces message, rendered from the given message template which may contain EL expressions.
- `addFromResourceBundle(String key)` — add a faces message, rendered from a message template defined in the Seam resource bundle which may contain EL expressions.
- `addFromResourceBundle(Severity severity, String key)` — add a faces message, rendered from a message template defined in the Seam resource bundle which may contain EL expressions.
- `clear()` — clear all messages.

`org.jboss.seam.core.redirect`
A convenient API for performing redirects with parameters (this is especially useful for bookmarkable search results screens).

- `redirect.viewId` — the JSF view id to redirect to.
- `redirect.conversationPropagationEnabled` — determines whether the conversation will propagate across the redirect.
- `redirect.parameters` — a map of request parameter name to value, to be passed in the redirect request.
- `execute()` — perform the redirect immediately.
- `captureCurrentRequest()` — stores the view id and request parameters of the current GET request (in the conversation context), for later use by calling `execute()`.

`org.jboss.seam.core.httpError`
A convenient API for sending HTTP errors.

`org.jboss.seam.core.events`
An API for raising events that can be observed via `@Observer` methods, or method bindings in `WEB-INF/events.xml`.

- `raiseEvent(String type)` — raise an event of a particular type and distribute to all observers.
- `raiseAsynchronousEvent(String type)` — raise an event to be processed asynchronously by the EJB3 timer service.
- `raiseTimedEvent(String type, ....)` — schedule an event to be processed asynchronously by the EJB3 timer service.
- `addListener(String type, String methodBinding)` — add an observer for a particular event type.

`org.jboss.seam.core.interpolator`
An API for interpolating the values of JSF EL expressions in Strings.

- `interpolate(String template)` — scan the template for JSF EL expressions of the form `#{...}` and replace them with their evaluated values.

`org.jboss.seam.core.expressions`
An API for creating value and method bindings.

- `createValueBinding(String expression)` — create a value binding object.
- `createMethodBinding(String expression)` — create a method binding object.

`org.jboss.seam.core.pojoCache`
Manager component for a JBoss Cache `PojoCache` instance.

- `pojoCache.cfgResourceName` — the name of the configuration file. Default to `treecache.xml`.

`org.jboss.seam.core.uiComponent`
Allows access to a JSF `UIComponent` by its id from the EL. For example, we can write `@In("#{uiComponent['myForm:address'].value}")`.

All of these components are always installed.

17.3. Components for internationalization and themes

The next group of components make it easy to build internationalized user interfaces using Seam.

`org.jboss.seam.core.locale`
The Seam locale. The locale is session scoped.

`org.jboss.seam.core.timezone`
The Seam timezone. The timezone is session scoped.

`org.jboss.seam.core.resourceBundle`
The Seam resource bundle. The resource bundle is session scoped. The Seam resource bundle performs a depth-first search for keys in a list of Java resource bundles.

- `resourceBundle.bundleNames` — the names of the Java resource bundles to search. Default to `messages`.

`org.jboss.seam.core.localeSelector`
Supports selection of the locale either at configuration time, or by the user at runtime.

- `select()` — select the specified locale.
- `localeSelector.locale` — the actual `java.util.Locale`.
- `localeSelector.localeString` — the stringified representation of the locale.
- `localeSelector.language` — the language for the specified locale.
- `localeSelector.country` — the country for the specified locale.
- `localeSelector.variant` — the variant for the specified locale.
- `localeSelector.supportedLocales` — a list of `SelectItems` representing the supported locales listed

in `jsf-config.xml`.

- `localeSelector.cookieEnabled` — specifies that the locale selection should be persisted via a cookie.

`org.jboss.seam.core.timezoneSelector`
Supports selection of the timezone either at configuration time, or by the user at runtime.

- `select()` — select the specified locale.
- `timezoneSelector.timezone` — the actual `java.util.TimeZone`.
- `timezoneSelector.timeZoneId` — the stringified representation of the timezone.
- `timezoneSelector.cookieEnabled` — specifies that the timezone selection should be persisted via a cookie.

`org.jboss.seam.core.messages`
A map containing internationalized messages rendered from message templates defined in the Seam resource bundle.

`org.jboss.seam.theme.themeSelector`
Supports selection of the theme either at configuration time, or by the user at runtime.

- `select()` — select the specified theme.
- `theme.availableThemes` — the list of defined themes.
- `themeSelector.theme` — the selected theme.
- `themeSelector.themes` — a list of `SelectItem`s representing the defined themes.
- `themeSelector.cookieEnabled` — specifies that the theme selection should be persisted via a cookie.

`org.jboss.seam.theme.theme`
A map containing theme entries.

All of these components are always installed.

17.4. Components for controlling conversations

The next group of components allow control of conversations by the application or user interface.

`org.jboss.seam.core.conversation`
API for application control of attributes of the current Seam conversation.

- `getId()` — returns the current conversation id
- `isNested()` — is the current conversation a nested conversation?
- `isLongRunning()` — is the current conversation a long-running conversation?
- `getId()` — returns the current conversation id

- `getParentId()` — returns the conversation id of the parent conversation
- `getRootId()` — returns the conversation id of the root conversation
- `setTimeout(int timeout)` — sets the timeout for the current conversation
- `setViewId(String outcome)` — sets the view id to be used when switching back to the current conversation from the conversation switcher, conversation list, or breadcrumbs.
- `setDescription(String description)` — sets the description of the current conversation to be displayed in the conversation switcher, conversation list, or breadcrumbs.
- `redirect()` — redirect to the last well-defined view id for this conversation (useful after login challenges).
- `leave()` — exit the scope of this conversation, without actually ending the conversation.
- `begin()` — begin a long-running conversation (equivalent to `@Begin`).
- `beginPageflow(String pageflowName)` — begin a long-running conversation with a pageflow (equivalent to `@Begin(pageflow="...")`).
- `end()` — end a long-running conversation (equivalent to `@End`).
- `pop()` — pop the conversation stack, returning to the parent conversation.
- `root()` — return to the root conversation of the conversation stack.
- `changeFlushMode(FlushModeType flushMode)` — change the flush mode of the conversation.

`org.jboss.seam.core.conversationList`
: Manager component for the conversation list.

`org.jboss.seam.core.conversationStack`
: Manager component for the conversation stack (breadcrumbs).

`org.jboss.seam.core.switcher`
: The conversation switcher.

All of these components are always installed.

17.5. jBPM-related components

These components are for use with jBPM.

`org.jboss.seam.core.pageflow`
: API control of Seam pageflows.

- `isInProcess()` — returns `true` if there is currently a pageflow in process
- `getProcessInstance()` — returns jBPM `ProcessInstance` for the current pageflow
- `begin(String pageflowName)` — begin a pageflow in the context of the current conversation

- `reposition(String nodeName)` — reposition the current pageflow to a particular node

`org.jboss.seam.core.actor`
API for application control of attributes of the jBPM actor associated with the current session.

- `setId(String actorId)` — sets the jBPM actor id of the current user.
- `getGroupActorIds()` — returns a `Set` to which jBPM actor ids for the current users groups may be added.

`org.jboss.seam.core.transition`
API for application control of the jBPM transition for the current task.

- `setName(String transitionName)` — sets the jBPM transition name to be used when the current task is ended via `@EndTask`.

`org.jboss.seam.core.businessProcess`
API for programmatic control of the association between the conversation and business process.

- `businessProcess.taskId` — the id of the task associated with the current conversation.
- `businessProcess.processId` — the id of the process associated with the current conversation.
- `businessProcess.hasCurrentTask()` — is a task instance associated with the current conversation?
- `businessProcess.hasCurrentProcess()` — is a process instance associated with the current conversation.
- `createProcess(String name)` — create an instance of the named process definition and associate it with the current conversation.
- `startTask()` — start the task associated with the current conversation.
- `endTask(String transitionName)` — end the task associated with the current conversation.
- `resumeTask(Long id)` — associate the task with the given id with the current conversation.
- `resumeProcess(Long id)` — associate the process with the given id with the current conversation.
- `transition(String transitionName)` — trigger the transition.

`org.jboss.seam.core.taskInstance`
Manager component for the jBPM `TaskInstance`.

`org.jboss.seam.core.processInstance`
Manager component for the jBPM `ProcessInstance`.

`org.jboss.seam.core.jbpmContext`
Manager component for an event-scoped `JbpmContext`.

`org.jboss.seam.core.taskInstanceList`
Manager component for the jBPM task list.

`org.jboss.seam.core.pooledTaskInstanceList`

Manager component for the jBPM pooled task list.

`org.jboss.seam.core.taskInstanceListForType`
Manager component for the jBPM task lists.

`org.jboss.seam.core.pooledTask`
Action handler for pooled task assignment.

All of these components are installed whenever the component `org.jboss.seam.core.jbpm` is installed.

17.6. Security-related components

These components relate to web-tier security.

`org.jboss.seam.core.userPrincipal`
Manager component for the current user `Principal`.

`org.jboss.seam.core.isUserInRole`
Allows JSF pages to choose to render a control, depending upon the roles available to the current principal. `<h:commandButton value="edit" rendered="#{isUserInRole['admin']}"/>`.

17.7. JMS-related components

These components are for use with managed `TopicPublishers` and `QueueSenders` (see below).

`org.jboss.seam.jms.queueSession`
Manager component for a JMS `QueueSession` .

`org.jboss.seam.jms.topicSession`
Manager component for a JMS `TopicSession` .

17.8. Infrastructural components

These components provide critical platform infrastructure. You can install a component by including its class name in the `org.jboss.seam.core.init.componentClasses` configuration property.

`org.jboss.seam.core.init`
Initialization settings for Seam. Always installed.

- `org.jboss.seam.core.init.jndiPattern` — the JNDI pattern used for looking up session beans
- `org.jboss.seam.core.init.debug` — enable Seam debug mode
- `org.jboss.seam.core.init.clientSideConversations` — if set to `true`, Seam will save conversation context variables in the client instead of in the `HttpSession`.
- `org.jboss.seam.core.init.userTransactionName` — the JNDI name to use when looking up the JTA `UserTransaction` object.

`org.jboss.seam.core.manager`
Internal component for Seam page and conversation context management. Always installed.

- `org.jboss.seam.core.manager.conversationTimeout` — the conversation context timeout in milliseconds.
- `org.jboss.seam.core.manager.concurrentRequestTimeout` — maximum wait time for a thread attempting to gain a lock on the long-running conversation context.
- `org.jboss.seam.core.manager.conversationIdParameter` — the request parameter used to propagate the conversation id, default to `conversationId`.
- `org.jboss.seam.core.manager.conversationIsLongRunningParameter` — the request parameter used to propagate information about whether the conversation is long-running, default to `conversationIsLongRunning`.

`org.jboss.seam.core.pages`
Internal component for Seam workspace management. Always installed.

- `org.jboss.seam.core.pages.noConversationViewId` — global setting for the view id to redirect to when a conversation entry is not found on the server side.

`org.jboss.seam.core.ejb`
Bootstraps the JBoss Embeddable EJB3 container. Install as class `org.jboss.seam.core.Ejb`. This is useful when using Seam with EJB components outside the context of a Java EE 5 application server.

The basic Embedded EJB configuration is defined in `jboss-embedded-beans.xml`. Additional microcontainer configuration (for example, extra datasources) may be specified by `jboss-beans.xml` or `META-INF/jboss-beans.xml` in the classpath.

`org.jboss.seam.core.microcontainer`
Bootstraps the JBoss microcontainer. Install as class `org.jboss.seam.core.Microcontainer`. This is useful when using Seam with Hibernate and no EJB components outside the context of a Java EE application server. The microcontainer can provide a partial EE environment with JNDI, JTA, a JCA datasource and Hibernate.

The microcontainer configuration may be specified by `jboss-beans.xml` or `META-INF/jboss-beans.xml` in the classpath.

`org.jboss.seam.core.jbpm`
Bootstraps a `JbpmConfiguration`. Install as class `org.jboss.seam.core.Jbpm`.

- `org.jboss.seam.core.jbpm.processDefinitions` — a list of resource names of jPDL files to be used for orchestration of business processes.
- `org.jboss.seam.core.jbpm.pageflowDefinitions` — a list of resource names of jPDL files to be used for orchestration of conversation page flows.

`org.jboss.seam.core.conversationEntries`
Internal session-scoped component recording the active long-running conversations between requests.

`org.jboss.seam.core.facesPage`
Internal page-scoped component recording the conversation context associated with a page.

`org.jboss.seam.core.persistenceContexts`
Internal component recording the persistence contexts which were used in the current conversation.

`org.jboss.seam.jms.queueConnection`
Manages a JMS `QueueConnection`. Installed whenever managed managed `QueueSender` is installed.

- `org.jboss.seam.jms.queueConnection.queueConnectionFactoryJndiName` — the JNDI name of a JMS `QueueConnectionFactory`. Default to `UIL2ConnectionFactory`

`org.jboss.seam.jms.topicConnection`
Manages a JMS `TopicConnection`. Installed whenever managed managed `TopicPublisher` is installed.

- `org.jboss.seam.jms.topicConnection.topicConnectionFactoryJndiName` — the JNDI name of a JMS `TopicConnectionFactory`. Default to `UIL2ConnectionFactory`

`org.jboss.seam.persistence.persistenceProvider`
Abstraction layer for non-standardized features of JPA provider.

`org.jboss.seam.core.validation`
Internal component for Hibernate Validator support.

`org.jboss.seam.debug.introspector`
Support for the Seam Debug Page.

`org.jboss.seam.debug.contexts`
Support for the Seam Debug Page.

17.9. Special components

Certain special Seam component classes are installable multiple times under names specified in the Seam configuration. For example, the following lines in `components.xml` install and configure two Seam components:

```
<component name="bookingDatabase"
          class="org.jboss.seam.core.ManagedPersistenceContext">
    <property name="persistenceUnitJndiName">java:/comp/emf/bookingPersistence</property>
</component>

<component name="userDatabase"
          class="org.jboss.seam.core.ManagedPersistenceContext">
    <property name="persistenceUnitJndiName">java:/comp/emf/userPersistence</property>
</component>
```

The Seam component names are `bookingDatabase` and `userDatabase`.

`<entityManager>`, `org.jboss.seam.core.ManagedPersistenceContext`
Manager component for a conversation scoped managed `EntityManager` with an extended persistence context.

- `<entityManager>.entityManagerFactory` — a value binding expression that evaluates to an instance of `EntityManagerFactory`.

  `<entityManager>.persistenceUnitJndiName` — the JNDI name of the entity manager factory, default to `java:/<managedPersistenceContext>`.

<entityManagerFactory>, `org.jboss.seam.core.EntityManagerFactory`
Manages a JPA `EntityManagerFactory`. This is most useful when using JPA outside of an EJB 3.0 supporting environment.

- `entityManagerFactory.persistenceUnitName` — the name of the persistence unit.

See the API JavaDoc for further configuration properties.

<session>, `org.jboss.seam.core.ManagedSession`
Manager component for a conversation scoped managed Hibernate `Session`.

- *<session>*`.sessionFactory` — a value binding expression that evaluates to an instance of `SessionFactory`.

  <session>`.sessionFactoryJndiName` — the JNDI name of the session factory, default to `java:/`*<managedSession>*.

<sessionFactory>, `org.jboss.seam.core.HibernateSessionFactory`
Manages a Hibernate `SessionFactory`.

- `<sessionFactory>.cfgResourceName` — the path to the configuration file. Default to `hibernate.cfg.xml`.

See the API JavaDoc for further configuration properties.

<managedQueueSender>, `org.jboss.seam.jms.ManagedQueueSender`
Manager component for an event scoped managed JMS `QueueSender`.

- *<managedQueueSender>*`.queueJndiName` — the JNDI name of the JMS queue.

<managedTopicPublisher>, `org.jboss.seam.jms.ManagedTopicPublisher`
Manager component for an event scoped managed JMS `TopicPublisher`.

- *<managedTopicPublisher>*`.topicJndiName` — the JNDI name of the JMS topic.

<managedWorkingMemory>, `org.jboss.seam.drools.ManagedWorkingMemory`
Manager component for a conversation scoped managed Drools `WorkingMemory`.

- *<managedWorkingMemory>*`.ruleBase` — a value expression that evaluates to an instance of `RuleBase`.

<ruleBase>, `org.jboss.seam.drools.RuleBase`
Manager component for an application scoped Drools `RuleBase`. *Note that this is not really intended for production usage, since it does not support dynamic installation of new rules.*

- *<ruleBase>*`.ruleFiles` — a list of files containing Drools rules.

  <ruleBase>`.dslFile` — a Drools DSL definition.

<entityHome>, `org.jboss.seam.framework.EntityHome`

<hibernateEntityHome>, `org.jboss.seam.framework.HibernateEntityHome`

<*entityQuery*>, `org.jboss.seam.framework.EntityQuery`

<*hibernateEntityQuery*>, `org.jboss.seam.framework.HibernateEntityQuery`

Chapter 18. Seam JSF controls

Seam includes a number of JSF controls that are useful for working with Seam. These are intended to complement the built-in JSF controls, and controls from other third-party libraries. We recommend the Ajax4JSF and ADF faces (now Trinidad) tag libraries for use with Seam. We do not recommend the use of the Tomahawk tag library.

`<s:validate>`
Validate a JSF input field against the bound property using Hibernate Validator.

`<s:validateAll>`
Validate all child JSF input fields against the bound propertys using Hibernate Validator.

`<s:formattedText>`
Output *Seam Text*.

`<s:convertDateTime>`
Perform date or time conversions in the Seam timezone.

`<s:convertEnum>`
Assigns an enum converter to the current component. This is primarily useful for radio button and drop-down controls.

`<s:enumItem>`
Creates a `SelectItem` from an enum value.

- `enumValue` — the string representation of the enum value.
- `label` — the label to be used when rendering the `SelectItem`.

`<s:decorate>`
"Decorate" a JSF input field when validation fails.

`<s:message>`
"Decorate" a JSF input field with the validation error message.

`<s:span>`
Render a HTML `<span>`.

`<s:div>`
Render a HTML `<div>`.

`<s:cache>`
Cache the rendered page fragment using JBoss Cache. Note that `<s:cache>` actually uses the instance of JBoss Cache managed by the built-in `pojoCache` component.

- `key` — the key to cache rendered content, often a value expression. For example, if we were caching a page fragment that displays a document, we might use `key="Document-#{document.id}"`.
- `enabled` — a value expression that determines if the cache should be used.
- `region` — a JBoss Cache node to use (different nodes can have different expiry policies).

`<s:link>`
A link that supports invocation of an action with control over conversation propagation. Does not submit the form.

- `value` — the label.
- `action` — a method binding that specified the action listener.
- `view` — the JSF view id to link to.
- `fragment` — the fragment identifier to link to.
- `disabled` — is the link disabled?
- `propagation` — determines the conversation propagation style: `begin`, `join`, `nest`, `none` or `end`.
- `pageflow` — a pageflow definition to begin. (This is only useful when `propagation="begin"` or `propagation="join"`.)

`<s:button>`
A button that supports invocation of an action with control over conversation propagation. Does not submit the form.

- `value` — the label.
- `action` — a method binding that specified the action listener.
- `view` — the JSF view id to link to.
- `fragment` — the fragment identifier to link to.
- `disabled` — is the link disabled?
- `propagation` — determines the conversation propagation style: `begin`, `join`, `nest`, `none` or `end`.
- `pageflow` — a pageflow definition to begin. (This is only useful when `propagation="begin"` or `propagation="join"`.)

`<s:selectDate>`
Displays a dynamic date picker component that selects a date for the specified input field. The body of the selectDate element should contain HTML elements, such as text or an image, that prompt the user to click to display the date picker.

- `for` — The id of the input field that the date picker will insert the selected date into.

`<s:conversationPropagation>`
Customize the conversation propagation for a command link or button (or similar JSF control). *Facelets only.*

- `propagation` — determines the conversation propagation style: `begin`, `join`, `nest`, `none` or `end`.
- `pageflow` — a pageflow definition to begin. (This is only useful when `propagation="begin"` or `propagation="join"`.)

`<s:conversationId>`
: Add the conversation id to an output link (or similar JSF control). *Facelets only.*

`<s:taskId>`
: Add the task id to an output link (or similar JSF control), when the task is available via `#{task}`. *Facelets only.*

Chapter 19. iText PDF generation

Seam now includes an optional component set for generating documents using iText. The primary focus of Seam's iText document support is for the generation of PDF doucuments, but Seam also offers basic support for RTF document generation.

19.1. Using PDF Support

iText support is provided by jboss-seam-pdf.jar. This JAR contains the iText JSF controls, which are used to construct views that can render to PDF, and the DocumentStore component, which serves the rendered documents to the user. To include PDF support in your application, included jboss-seam-pdf.jar in your WEB-INF/lib directory along with the iText JAR file. There is no further configuration needed to use Seam's iText support.

The Seam iText module requires the use of Facelets as the view technology. Future versions of the library may also support the use of JSP. Additionally, it requires the use of the seam-ui package.

The examples/pdf project contains an example of the PDF support in action. It demonstrates proper deployment packaging, and it contains a number examples that demonstrate the key PDF generation features current supported.

19.2. Creating a document

Documents are generated by facelets documents using tags in the `http://jboss.com/products/seam/pdf` namespace. Documents should always have the `document` tag at the root of the document. The document tag prepares Seam to generate a document into the DocumentStore and renders an HTML redirect to that stored content. The following is a a small PDF document consisting only a single line of text:

```
<p:document xmlns:p="http://jboss.com/products/seam/pdf">
    A very tiny PDF
</p:document>
```

Documents should be composed entirely using Seam document components. Controls meant for HTML rendering are generally not supported. However, pure control structures can be used. The following document uses the `ui:repeat` tag to to display a list of values retrieved from a Seam component.

```
<p:document xmlns:ui="http://java.sun.com/jsf/facelets"
            xmlns:p="http://jboss.com/products/seam/pdf">

    <p:list>
        <ui:repeat value="#{lists.numbers}" var="item">
            <p:listItem><p:font style="bold">#{item}</p:font>:
            This is some information...</p:listItem>
        </ui:repeat>
    </p:list>
</p:document>
```

19.2.1. p:document

The p:document tag supports the following attributes:

`type`
: The type of the document to be produced. Valid values are `PDF`, `RTF` and `HTML` modes. Seam defaults to PDF generation, and many of the features only work correctly when generating PDF documents.

`pageSize`
: The size of the page to be generate. The most commonly used values would be `LETTER` and `A4`. A full list of supported pages sizes can be found in `com.lowagie.text.PageSize` class.

`margins`
: The left, right, top and bottom margin values.

`marginMirroring`
: Indicates that margin settings should be reversed an alternating pages.

Document metadata is also set as attributes of the document tag. The following metadata fields are supported:

`title`

`subject`

`keywords`

`author`

`creator`

19.3. Headers and Footers

19.3.1. p:header and p:footer

The `p:header` and `p:footer` components provide the ability to place header and footer text on each page of a generated document, with the exception of the first page. Header and footer declarations should appear near the top of a document.

`alignment`
: The alignment of the header/footer box section. (see Section 19.8.2, “Alignment Values” for alignment values)

`backgroundColor`
: The background color of the header/footer box. (see Section 19.8.1, “Color Values” for color values)

`borderColor`
: The border color of the header/footer box. Individual border sides can be set using `borderColorLeft`, `borderColorRight`, `borderColorTop` and `borderColorBottom`.(see Section 19.8.1, “Color Values” for color values)

`borderWidth`
: The width of the border. Inidvidual border sides can be specified using `borderWidthLeft`, `borderWidthRight`, `borderWidthTop` and `borderWidthBottom`.

19.3.2. p:pageNumber

The current page number can be placed inside of a header or footer using the `p:pageNumber` tag. The page number tag can only be used in the context of a header or footer and can only be used once.

19.4. Chapters and Sections

If the generated document follows a book/article structure, the `p:chapter` and `p:section` tags can be used to provide the necessary structure. Sections can only be used inside of chapters, but they may be nested arbitrarily deep.

19.4.1. p:chapter and p:section

`number`
: The chapter number. Every chapter should be assigned a chapter number.

`numberDepth`
: The depth of numbering for section. All sections are numbered relative to their surrounding chapter/sections. The fourth section of of the first section of chapter three would be section 3.1.4, if displayed at the default number depth of three. To omit the chapter number, a number depth of 2 should be used. In that case, the section number would be displayed as 1.4.

19.4.2. p:title

Any chapter or section can contain a `p:title`. The title will be displayed next to the chapter/section number. The body of the title may contain text or may be a `p:paragraph`.

19.5. Lists

List structures can be displayed using the `p:list` and `p:listItem` tags. Lists may contain arbitrarily-nested sublists. List items may not be used outside of a list.

19.5.1. p:list

`p:list` supports the following attributes:

`style`
: The ordering/bulleting style of list. One of: `NUMBERED`, `LETTERED`, `GREEK`, `ROMAN`, `ZAPFDINGBATS`, `ZAPFDINGBATS_NUMBER`. If no style is given, the list items are bulleted.

`listSymbol`
: For bulleted lists, specifies the bullet symbol.

`indent`
: The indentation level of the list.

`lowerCase`
: For list styles using letters, indicates whether the letters should be lower case.

`charNumber`
For ZAPFDINGBATS, indicates the character code of the bullet character.

`numberType`
For ZAPFDINGBATS_NUMBER, indicates the numbering style.

19.5.2. p:listItem

`p:listItem` supports the following attributes:

`alignment`
The alignment of the list item. (See Section 19.8.2, “Alignment Values” for possible values)

`indentationLeft`
The left indentation amount.

`indentationRight`
The right indentation amount.

`listSymbol`
Overrides the default list symbol for this list item.

19.6. Tables

Table structures can be created using the `p:table` and `p:cell` tags. Unlike many table structures, there is no explicit row declaration. If a table has 3 columns, then every 3 cells will automatically form a row.

19.6.1. p:table

`p:table` supports the following attributes.

`columns`
The number of columns (cells) that make up a table row.

`widths`
The relative widths of each column. There should be one value for each column. For example: widths="2 1 1" would indicate that there are 3 columns and the first column should be twice the size of the second and third column.

`headerRows`
The initial number of rows which are considered to be headers or footer rows and should be repeated if the table spans multiple pages.

`footerRows`
The number of rows that are considered to be footer rows. This value is subtracted from the `headerRows` value. If document has 2 rows which make up the header and one row that makes up the footer, `headerRows` should be set to 3 and `footerRows` should be set to 1

`widthPercentage`
The percentage of the page width that the table spans.

`horizontalAlignment`
The horizontal alignment of the table. (See Section 19.8.2, “Alignment Values” for possible values)

`skipFirstHeader`

`runDirection`

`lockedWidth`

`splitRows`

`spacingBefore`
The blank space to be inserted before the element.

`spacingAfter`
The blank space to be inserted after the element.

`extendLastRow`

`headersInEvent`

`splitLate`

`keepTogether`

19.6.2. p:cell>

`p:cell` supports the following attributes.

`colspan`
Cells can span more than one column by declaring a `colspan` greater than 1. Tables do not have the ability to span across multiple rows.

`horizontalAlignment`
The horizontal alignment of the cell. (see Section 19.8.2, “Alignment Values” for possible values)

`verticalAlignment`
The vertical alignment of the cell. (see Section 19.8.2, “Alignment Values” for possible values)

`padding`
Padding on a given side can also be specified using `paddingLeft`, `paddingRight`, `paddingTop` and `paddingBottom`.

`useBorderPadding`

`leading`

`multipliedLeading`

`indent`

`verticalAlignment`

`extraParagraphSpace`

`fixedHeight`

`noWrap`

`minimumHeight`

`followingIndent`

`rightIndent`

`spaceCharRatio`

`runDirection`

`arabicOptions`

`useAscender`

`grayFill`

`rotation`

19.7. Basic Text Elements

19.7.1. p:paragraph

`firstLineIndent`

`extraParagraphSpace`

`leading`

`multipliedLeading`

`spacingBefore`
The blank space to be inserted before the element.

`spacingAfter`
The blank space to be inserted after the element.

`indentationLeft`

`indentationRight`

`keepTogether`

19.7.2. p:font

`familyName`

`size`

`style`

19.7.3. p:newPage

`p:newPage` inserts a page break.

19.7.4. p:image

`p:image` inserts an image into the document.

`resource`
: The location of the image resource to be included. Resources should be relative to the document root of the web application.

`rotation`
: The rotation of the image in degrees.

`height`
: The height of the image.

`width`
: The width of the image.

`alignment`
: The alignment of the image. (see Section 19.8.2, “Alignment Values” for possible values)

`alt`

`indentationLeft`

`indentationRight`

`spacingBefore`
: The blank space to be inserted before the element.

`spacingAfter`
: The blank space to be inserted after the element.

`widthPercentage`

`initialRotation`

`dpi`

`wrap`

`underlying`

19.7.5. p:anchor>

`p:anchor` defines clickable links from a document. It supports the following attributes:

`name`
: The name of an in-document anchor destination.

`reference`
: The destination the link refers to. Links to other points in the document should begin with a "#". For example, "#link1" to refer to an anchor postion with a `name` of `link1`. Links may also be a full URL to point to a resource outside of the document.

19.8. Document Constants

19.8.1. Color Values

Seam documents do not yet support a full color specification. Currently, only named colors are supported. They are: `white`, `gray`, `lightgray`, `darkgray`, `black`, `red`, `pink`, `yellow`, `green`, `magenta`, `cyan` and `blue`.

19.8.2. Alignment Values

`left`, `right`, `center`, `justify`, `justifyall`, `top`, `middle`, `bottom`, `baseline`

19.9. iText links

For further information on iText, see:

- iText Home Page [http://www.lowagie.com/iText/]
- iText in Action [http://www.manning.com/lowagie/]

Chapter 20. Expression language enhancements

The standard Unified Expression Language (EL) assumes that any parameters to a method expression will be provided by Java code. This means that a method with parameters cannot be used as a JSF method binding. Seam provides an enhancement to the EL that allows parameters to be included in a method expression itself. This applies to *any* Seam method expression, including any JSF method binding, for example:

```
<s:commandButton action="#{hotelBooking.bookHotel(hotel)}" value="Book Hotel"/>
```

20.1. Configuration

To use this feature in Facelets, you will need to declare a special view handler, `SeamFaceletViewHandler` in `faces-config.xml`.

```
<faces-config>
    <application>
        <view-handler>org.jboss.seam.ui.facelet.SeamFaceletViewHandler</view-handler>
    </application>
</faces-config>
```

20.2. Usage

Parameters are surrounded by parentheses, and separated by commas:

```
<h:commandButton action="#{hotelBooking.bookHotel(hotel, user)}" value="Book Hotel"/>
```

The parameters `hotel` and `user` will be evaluated as value expressions and passed to the `bookHotel()` method of the component. This gives you an alternative to the use of `@In`.

Any value expression may be used as a parameter:

```
<h:commandButton action="#{hotelBooking.bookHotel(hotel.id, user.username)}" value="Book Hotel"/>
```

You may even pass literal strings using single or double quotes:

```
<h:commandLink action="#{printer.println( 'Hello world!' )}" value="Hello"/>
```

```
<h:commandLink action='#{printer.println( "Hello again" )}' value='Hello'/>
```

You might even want to use this notation for all your action methods, even when you don't have parameters to pass. This improves readability by making it clear that the expression is a method expression and not a value expression:

```
<s:link value="Cancel" action="#{hotelBooking.cancel()}"/>
```

20.3. Limitations

Please be aware of the following limitations:

20.3.1. Incompatibility with JSP 2.1

This extension is not currently compatible with JSP 2.1. So if you want to use this extension with JSF 1.2, you will need to use Facelets. The extension works correctly with JSP 2.0.

20.3.2. Calling a `MethodExpression` from Java code

Normally, when a `MethodExpression` or `MethodBinding` is created, the parameter types are passed in by JSF. In the case of a method binding, JSF assumes that there are no parameters to pass. With this extension, we can't know the parameter types until after the expression has been evaluated. This has two minor consequences:

- When you invoke a `MethodExpression` in Java code, parameters you pass may be ignored. Parameters defined in the expression will take precedence.
- Ordinarily, it is safe to call `methodExpression.getMethodInfo().getParamTypes()` at any time. For an expression with parameters, you must first invoke the `MethodExpression` before calling `getParamTypes()`.

Both of these cases are exceedingly rare and only apply when you want to invoke the `MethodExpression` by hand in Java code.

Chapter 21. Testing Seam applications

Most Seam applications will need at least two kinds of automated tests: *unit tests*, which test a particular Seam component in isolation, and scripted *integration tests* which exercise all Java layers of the application (that is, everything except the view pages).

Both kinds of tests are very easy to write.

21.1. Unit testing Seam components

All Seam components are POJOs. This is a great place to start if you want easy unit testing. And since Seam emphasises the use of bijection for inter-component interactions and access to contextual objects, it's very easy to test a Seam component outside of its normal runtime environment.

Consider the following Seam component:

```
@Stateless
@Scope(EVENT)
@Name("register")
public class RegisterAction implements Register
{
   private User user;
   private EntityManager em;

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

   @PersistenceContext
   public void setBookingDatabase(EntityManager em) {
       this.em = em;
   }

   public String register()
   {
      List existing = em.createQuery("select username from User where username=:username")
         .setParameter("username", user.getUsername())
         .getResultList();
      if (existing.size()==0)
      {
         em.persist(user);
         return "success";
      }
      else
      {
         return null;
      }
   }

}
```

We could write a TestNG test for this component as follows:

```
public class RegisterActionTest
{

    @Test
    public testRegisterAction()
    {
        EntityManager em = getEntityManagerFactory().createEntityManager();
        em.getTransaction().begin();
```

```
        User gavin = new User();
        gavin.setName("Gavin King");
        gavin.setUserName("1ovthafew");
        gavin.setPassword("secret");

        RegisterAction action = new RegisterAction();
        action.setUser(gavin);
        action.setBookingDatabase(em);

        assert "success".equals( action.register() );

        em.getTransaction().commit();
        em.close();
    }


    private EntityManagerFactory emf;

    public EntityManagerFactory getEntityManagerFactory()
    {
        return emf;
    }

    @Configuration(beforeTestClass=true)
    public void init()
    {
        emf = Persistence.createEntityManagerFactory("myResourceLocalEntityManager");
    }

    @Configuration(afterTestClass=true)
    public void destroy()
    {
        emf.close();
    }

}
```

Seam components don't usually depend directly upon container infrastructure, so most unit testing as as easy as that!

21.2. Integration testing Seam applications

Integration testing is slightly more difficult. In this case, we can't eliminate the container infrastructure; indeed, that is part of what is being tested! At the same time, we don't want to be forced to deploy our application to an application server to run the automated tests. We need to be able to reproduce just enough of the container infrastructure inside our testing environment to be able to exercise the whole application, without hurting performance too much.

A second problem is emulating user interactions. A third problem is where to put our assertions. Some test frameworks let us test the whole application by reproducing user interactions with the web browser. These frameworks have their place, but they are not appropriate for use at development time.

The approach taken by Seam is to let you write tests that script your components while running inside a pruned down container environment (Seam, together with the JBoss Embeddable EJB container). The role of the test script is basically to reproduce the interaction between the view and the Seam components. In other words, you get to pretend you are the JSF implementation!

This approach tests everything except the view.

Let's consider a JSP view for the component we unit tested above:

```
<html>
 <head>
  <title>Register New User</title>
 </head>
 <body>
  <f:view>
   <h:form>
     <table border="0">
       <tr>
         <td>Username</td>
         <td><h:inputText value="#{user.username}"/></td>
       </tr>
       <tr>
         <td>Real Name</td>
         <td><h:inputText value="#{user.name}"/></td>
       </tr>
       <tr>
         <td>Password</td>
         <td><h:inputSecret value="#{user.password}"/></td>
       </tr>
     </table>
     <h:messages/>
     <h:commandButton type="submit" value="Register" action="#{register.register}"/>
   </h:form>
  </f:view>
 </body>
</html>
```

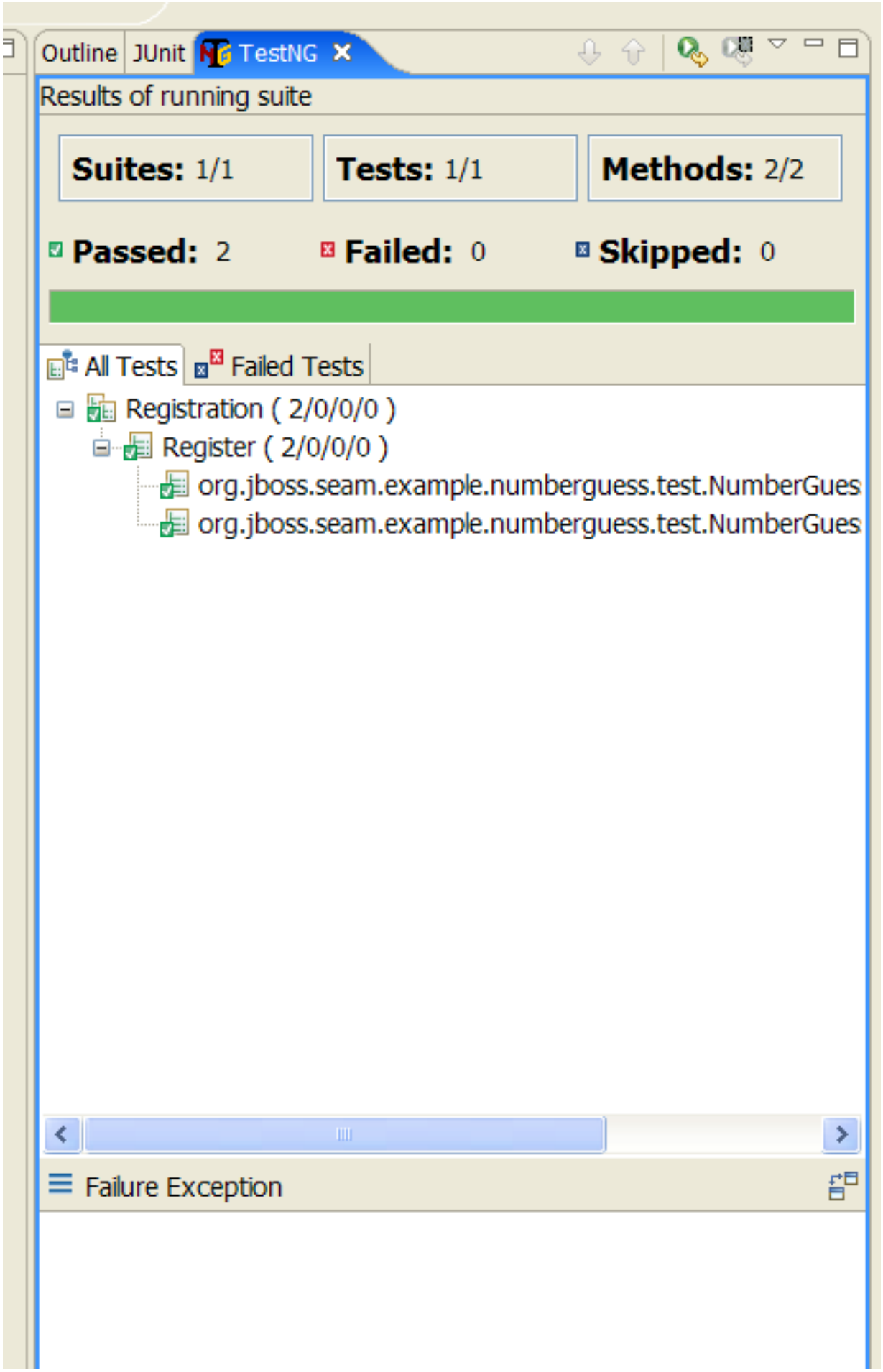
We want to test the registration functionality of our application (the stuff that happens when the user clicks the Register button). We'll reproduce the JSF request lifecycle in an automated TestNG test:

```
public class RegisterTest extends SeamTest
{

   @Test
   public void testRegister() throws Exception
   {

      new FacesRequest() {

         @Override
         protected void processValidations() throws Exception
         {
            validateValue("#{user.username}", "1ovthafew");
            validateValue("#{user.name}", "Gavin King");
            validateValue("#{user.password}", "secret");
            assert !isValidationFailure();
         }

         @Override
         protected void updateModelValues() throws Exception
         {
            setValue("#{user.username}", "1ovthafew");
            setValue("#{user.name}", "Gavin King");
            setValue("#{user.password}", "secret");
         }

         @Override
         protected void invokeApplication()
         {
            assert invokeMethod("#{register.register}").equals("success");
         }

         @Override
         protected void renderResponse()
         {
            assert getValue("#{user.username}").equals("1ovthafew");
            assert getValue("#{user.name}").equals("Gavin King");
            assert getValue("#{user.password}").equals("secret");
```

```
         }

      }.run();

   }

   ...
}
```

Notice that we've extended `SeamTest`, which provides a Seam environment for our components, and written our test script as an anonymous class that extends `SeamTest.FacesRequest`, which provides an emulated JSF request lifecycle. (There is also a `SeamTest.NonFacesRequest` for testing GET requests.) We've written our code in methods which are named for the various JSF phases, to emulate the calls that JSF would make to our components. Then we've thrown in various assertions.

You'll find plenty of integration tests for the Seam example applications which demonstrate more complex cases. There are instructions for running these tests using Ant, or using the TestNG plugin for eclipse:


Outline
JUnit
TestNG
Results of running suite
Suites: 1/1
Tests: 1/1
Methods: 2/2
Passed: 2
Failed: 0
Skipped: 0
All Tests
Failed Tests
Registration (2/0/0/0)
Register (2/0/0/0)
org.jboss.seam.example.numberguess.test.NumberGues
org.jboss.seam.example.numberguess.test.NumberGues
Failure Exception

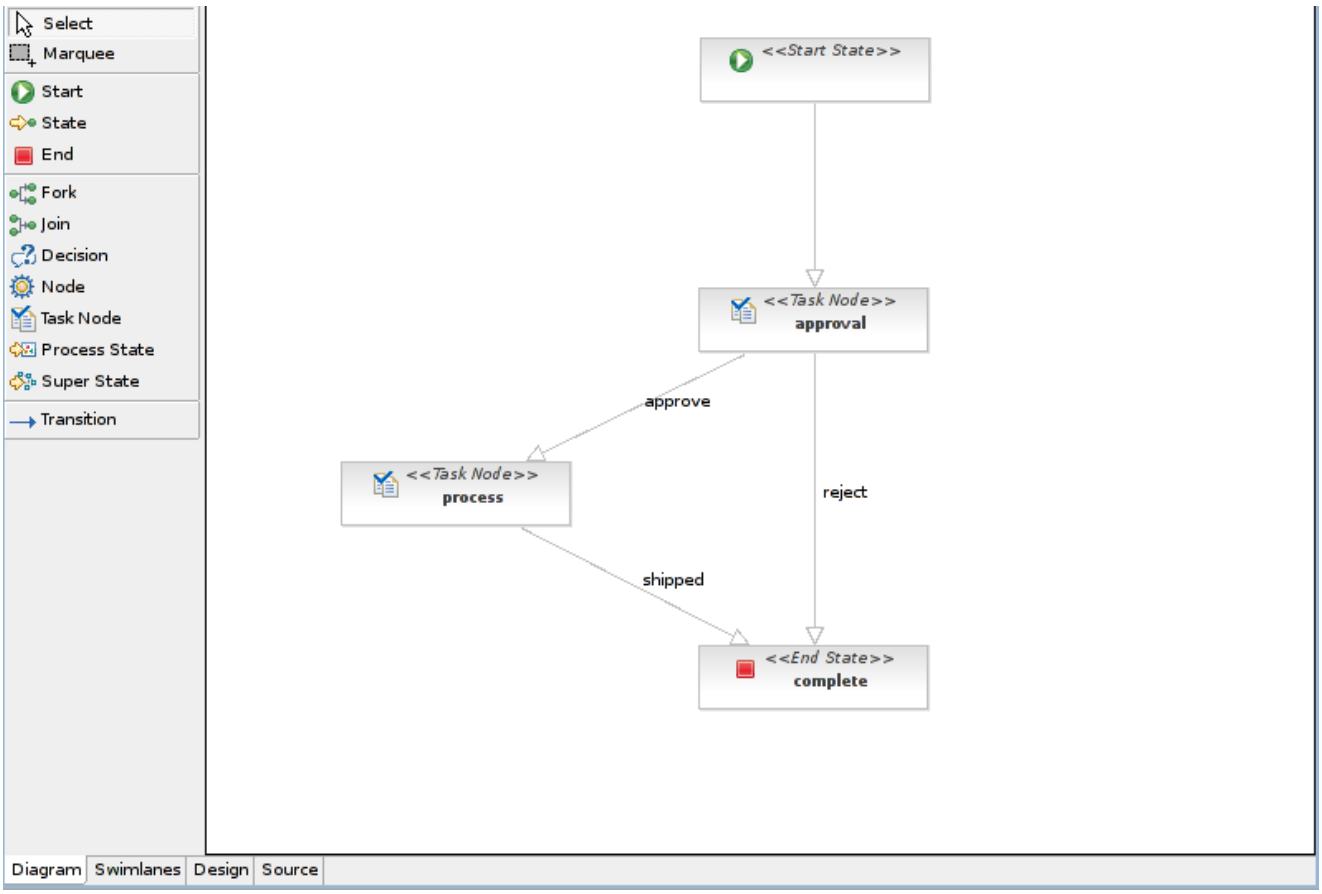
Chapter 22. Seam tools

22.1. jBPM designer and viewer

The jBPM designer and viewer will let you design and view in a nice way your business processes and your pageflows. This convenient tool is part of JBoss Eclipse IDE and more details can be found in the jBPM's documentation (http://docs.jboss.com/jbpm/v3/gpd/)

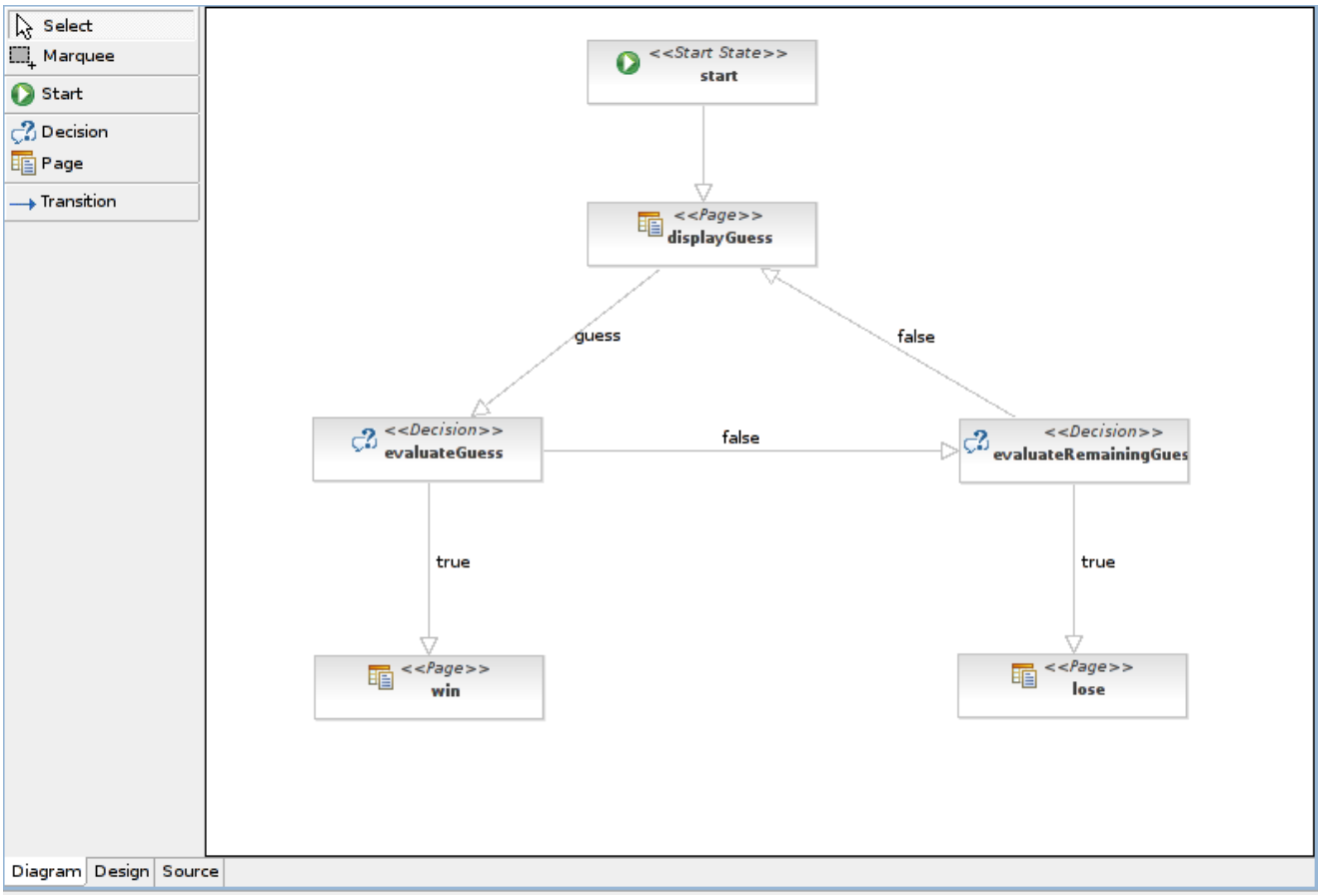
22.1.1. Business process designer

This tool lets you design your own business process in a graphical way.



22.1.2. Pageflow viewer

This tool let you design to some extend your pageflows and let you build graphical views of them so you can easily share and compare ideas on how it should be designed.



22.2. CRUD-application generator

This chapter, will give you a short overview of the support for Seam that is available in the Hibernate Tools. Hibernate Tools is a set of tools for working with Hibernate and related technologies, such as JBoss Seam and EJB3. The tools are available as a set of eclipse plugins and Ant tasks. You can download the Hibernate Tools from the JBoss Eclipse IDE or Hibernate Tools websites.

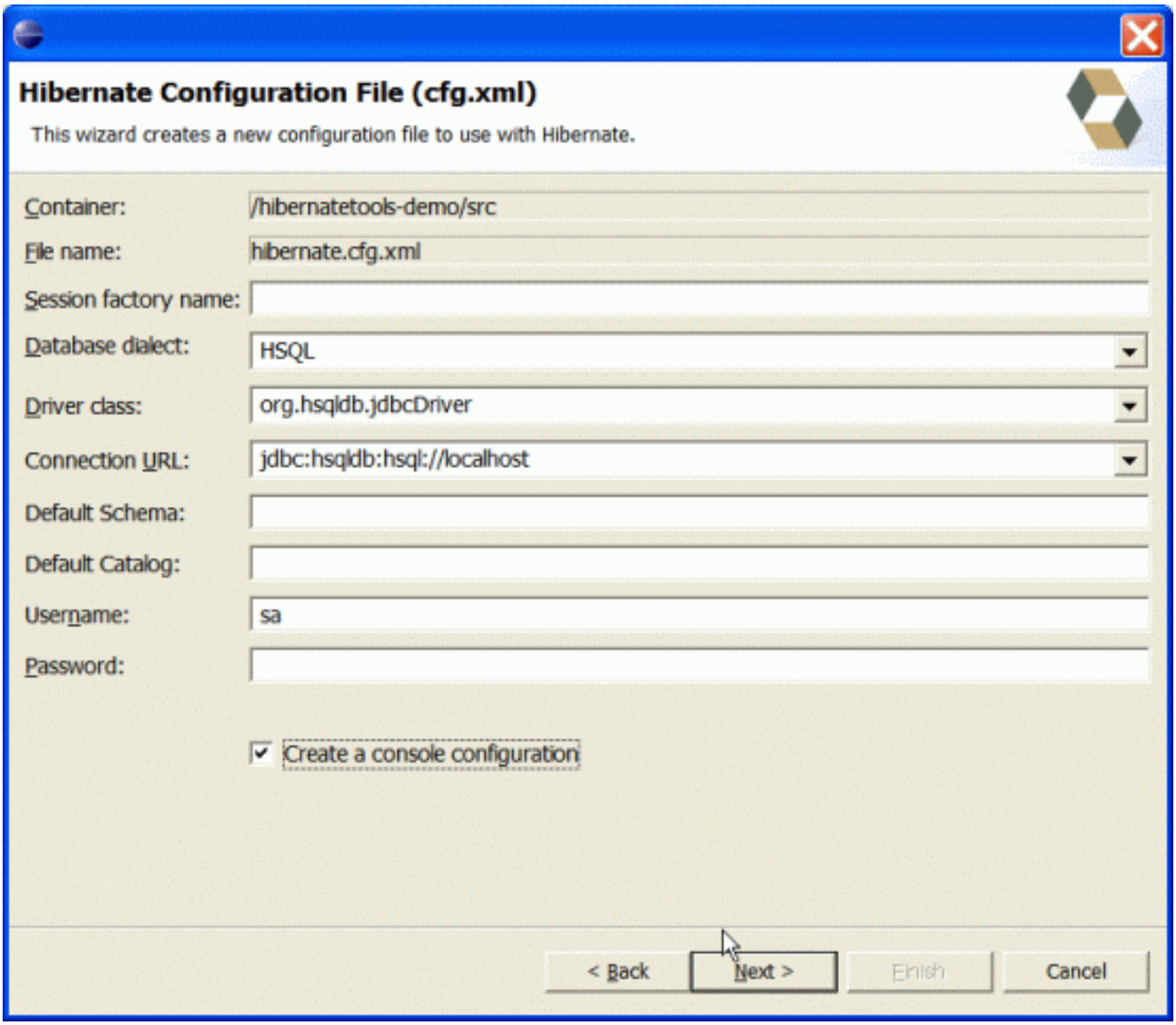
The specific support for Seam that is currently available is generation of a fully functional Seam based CRUD-application. The CRUD-application can be generated based on your existing Hibernate mapping files or EJB3 annotated POJO's or even fully reverse engineered from your existing database schema.

The following sections is focused on the features required to understand for usage with Seam. The content is derived from the the Hibernate Tools reference documentation. Thus if you need more detailed information please refer to the Hibernate Tools documentation.

22.2.1. Creating a Hibernate configuration file

To be able to reverse engineer and generate code a hibernate.properties or hibernate.cfg.xml file is needed. The Hibernate Tools provide a wizard for generating the hibernate.cfg.xml file if you do not already have such file.

Start the wizard by clicking "New Wizard" (Ctrl+N), select the Hibernate/Hibernate Configuration file (cfg.xml) wizard and press "Next". After selecting the wanted location for the hibernate.cfg.xml file, you will see the following page:



Tip: The contents in the combo boxes for the JDBC driver class and JDBC URL change automatically, depending on the Dialect and actual driver you have chosen.

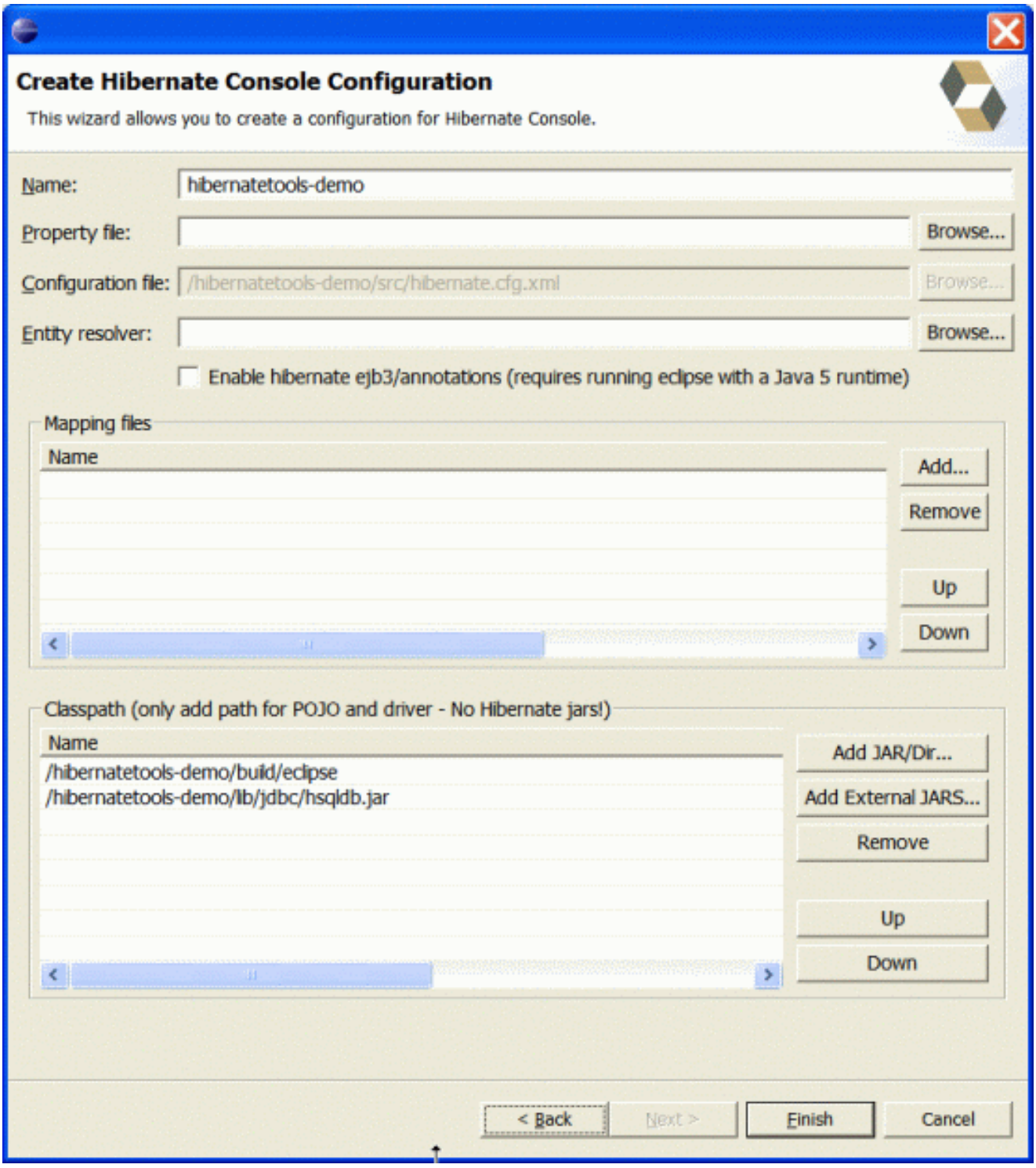
Enter your configuration information in this dialog. Details about the configuration options can be found in Hibernate reference documentation.

Press "Finish" to create the configuration file, after optionally creating a Console onfiguration, the hibernate.cfg.xml will be automatically opened in an editor. The last option "Create Console Configuration" is enabled by default and when enabled i will automatically use the hibernate.cfg.xml for the basis of a "Console Configuration"

22.2.2. Creating a Hibernate Console configuration

A Console Configuration describes to the Hibernate plugin which configuration files should be used to configure hibernate, including which classpath is needed to load the POJO's, JDBC drivers etc. It is required to make usage of query prototyping, reverse engineering and code generation. You can have multiple named console configurations. Normally you would just need one per project, but more (or less) is definitly possible.

You create a console configuration by running the Console Configuration wizard, shown in the following screenshot. The same wizard will also be used if you are coming from the hibernate.cfg.xml wizard and had enabled "Create Console Configuration".



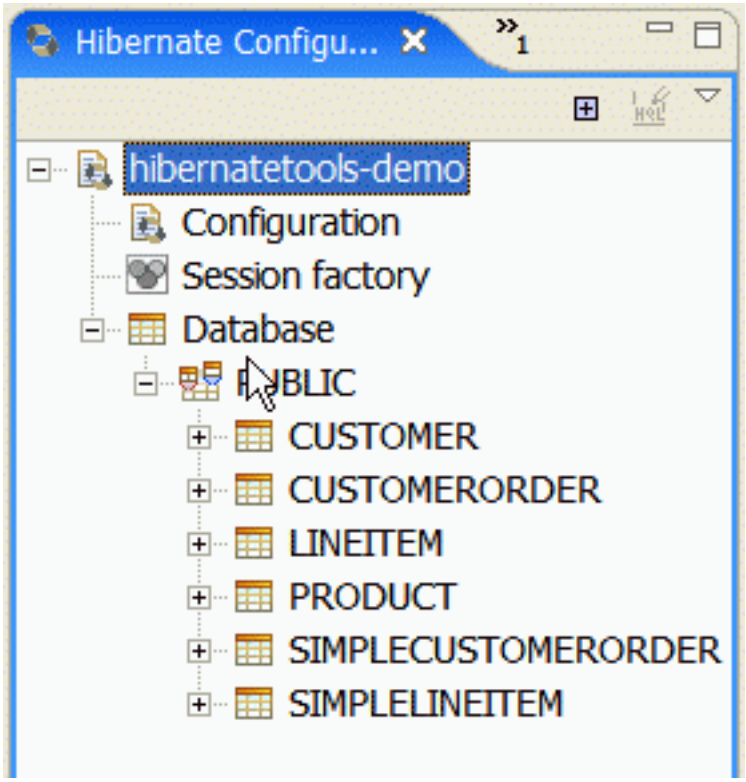
The following table describes the relevant settings. The wizard can automatically detect default values for most of these if you started the Wizard with the relevant java project selected

Table 22.1. Hibernate Console Configuration Parameters

Parameter	Description	Auto detected value
Name	The unique name of the configuration	Name of the selected project

Parameter	Description	Auto detected value
Property file	Path to a hibernate.properties file	First hibernate.properties file found in the selected project
Configuration file	Path to a hibernate.cfg.xml file	First hibernate.cfg.xml file found in the selected project
Enable Hibernate ejb3/annotations	Selecting this option enables usage of annotated classes. hbm.xml files are of course still possible to use too. This feature requires running the Eclipse IDE with a JDK 5 runtime, otherwise you will get classloading and/or version errors.	Not enabled
Mapping files	List of additional mapping files that should be loaded. Note: A hibernate.cfg.xml can also contain mappings. Thus if these a duplicated here, you will get "Duplicate mapping" errors when using the console configuration.	If no hibernate.cfg.xml file is found, all hbm.xml files found in the selected project
Classpath	The classpath for loading POJO and JDBC drivers. Do not add Hibernate core libraries or dependencies, they are already included. If you get ClassNotFound errors then check this list for possible missing or redundant directories/jars.	The default build output directory and any JARs with a class implementing java.sql.Driver in the selected project

Clicking "Finish" creates the configuration and shows it in the "Hibernate Configurations" view



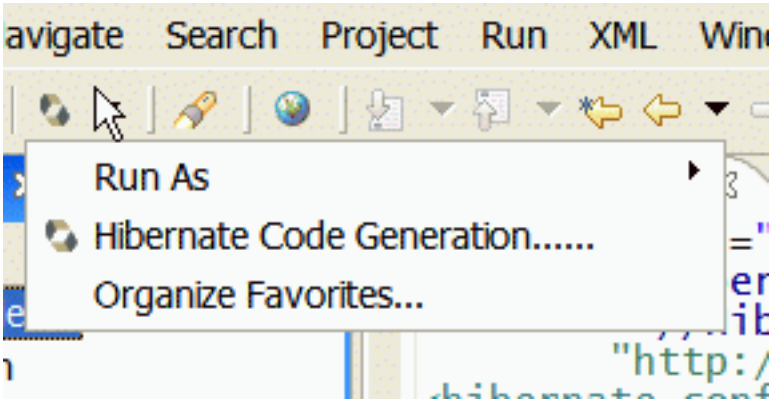
22.2.3. Reverse engineering and code generation

A very simple "click-and-generate" reverse engineering and code generation facility is available. It is this facility that allows you to generate the skeleton for a full Seam CRUD application.

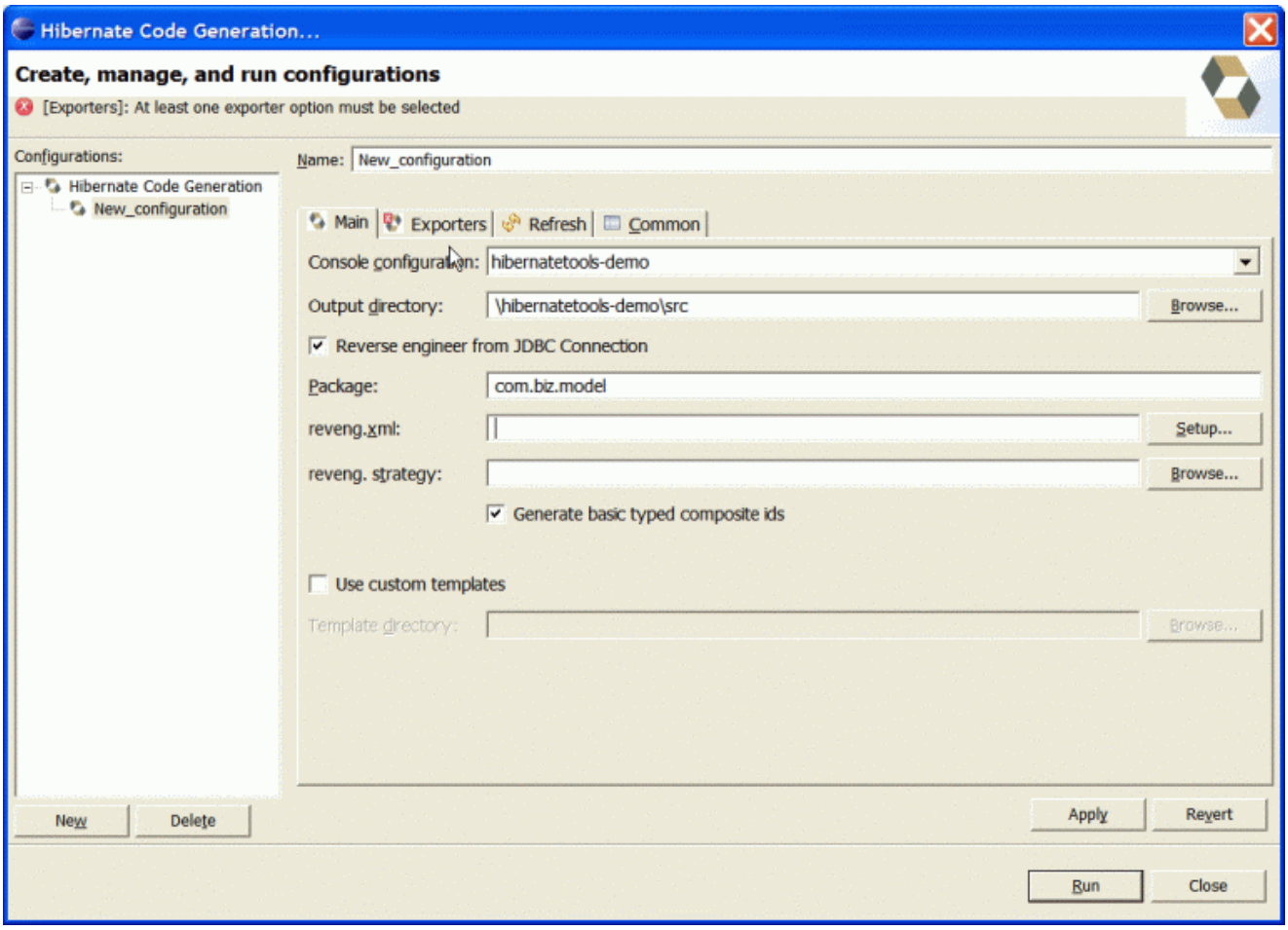
To start working with this process, start the "Hibernate Code Generation" which is available in the toolbar via the Hibernate icon or via the "Run/Hibernate Code Generation" menu item.

22.2.3.1. Code Generation Launcher

When you click on "Hibernate Code Generation" the standard Eclipse launcher dialog will appear. In this dialog you can create, edit and delete named Hibernate code generation "launchers".



The dialog has the standard tabs "Refresh" and "Common" that can be used to configure which directories should be automatically refreshed and various general settings launchers, such as saving them in a project for sharing the launcher within a team.



The first time you create a code generation launcher you should give it a meaningfull name, otherwise the default prefix "New_Generation" will be used.

Note: The "At least one exporter option must be selected" is just a warning stating that for this launch to work you need to select an exporter on the Exporter tab. When an exporter has been selected the warning will disappear.

On the "Main" tab you the following fields:

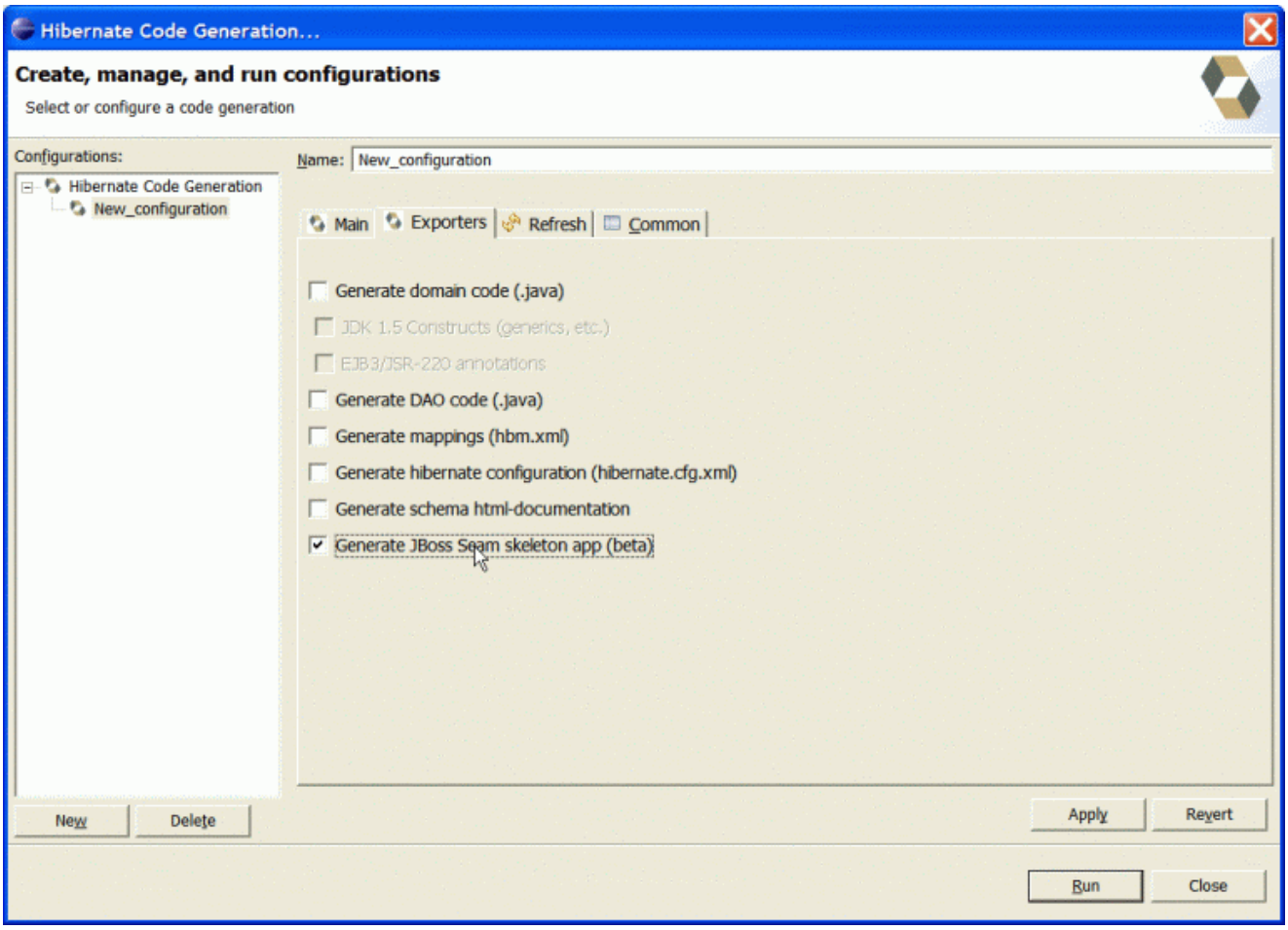
Table 22.2. Code generation "Main" tab fields

Field	Description
Console Configuration	The name of the console configuration which should be used when code generating.
Output directory	Path to a directory into where all output will be written by default. Be aware that existing files will be overwritten, so be sure to specify the correct directory.
Reverse engineer from JDBC Connection	If enabled the tools will reverse engineer the database available via the connection information in the selected Hibernate Console Configuration and generate code based on the database schema. If not enabled the code generation will just be based on the mappings already specified in the Hibernate Console configuration.

Field	Description
Package	The package name here is used as the default package name for any entities found when reverse engineering.
reveng.xml	Path to a reveng.xml file. A reveng.xml file allows you to control certain aspects of the reverse engineering. e.g. how jdbc types are mapped to hibernate types and especially important which tables are included/excluded from the process. Clicking "setup" allows you to select an existing reveng.xml file or create a new one..
reveng. strategy	If reveng.xml does not provide enough customization you can provide your own implementation of an ReverseEngineeringStrategy. The class need to be in the claspath of the Console Configuration, otherwise you will get class not found exceptions.
Generate basic typed composite ids	This field should always be enabled when generating the Seam CRUD application. A table that has a multi-colum primary key a <composite-id> mapping will always be created. If this option is enabled and there are matching foreign-keys each key column is still considered a 'basic' scalar (string, long, etc.) instead of a reference to an entity. If you disable this option a <key-many-to-one> instead. Note: a <many-to-one> property is still created, but is simply marked as non-updatable and non-insertable.
Use custom templates	If enabled, the Template directory will be searched first when looking up the velocity templates, allowing you to redefine how the individual templates process the hibernate mapping model.
Template directory	A path to a directory with custom velocity templates.

22.2.3.2. Exporters

The exporters tab is used to specify which type of code that should be generated. Each selection represents an "Exporter" that are responsible for generating the code, hence the name.



The following table describes in short the various exporters. The most relevant for Seam is of course the "JBoss Seam Skeleton app".

Table 22.3. Code generation "Exporter" tab fields

Field	Description
Generate domain code	Generates POJO's for all the persistent classes and components found in the given Hibernate configuration.
JDK 1.5 constructs	When enabled the POJO's will use JDK 1.5 constructs.
EJB3/JSR-220 annotations	When enabled the POJO's will be annotated according to the EJB3/JSR-220 persistency specification.
Generate DAO code	Generates a set of DAO's for each entity found.
Generate Mappings	Generate mapping (hbm.xml) files for each entity
Generate hibernate configuration file	Generate a hibernate.cfg.xml file. Used to keep the hibernate.cfg.xml uptodate with any new found mapping files.
Generate schema html-	Generates set of html pages that documents the database schema and some of the

Field	Description
documentation	mappings.
Generate JBoss Seam skeleton app (beta)	Generates a complete JBoss Seam skeleton app. The generation will include annotated POJO's, Seam controller beans and a JSP for the presentation layer. See the generated readme.txt for how to use it. Note: this exporter generates a full application, including a build.xml thus you will get the best results if you use an output directory which is the root of your project.

22.2.3.3. Generating and using the code

When you have finished filling out the settings, simply press "Run" to start the generation of code. This might take a little while if you are reverse engineering from a database.

When the generation have finished you should now have a complete skeleton Seam application in the output directory. In the output directory there is a `readme.txt` file describing the steps needed to deploy and run the example.

If you want to regenerate/update the skeleton code then simply run the code generation again by selecting the "Hibernate Code Generation" in the toolbar or "Run" menu. Enjoy.

Chapter 23. Security

The Seam Security API is an optional Seam module that provides authentication and authorization features for securing both domain and page resources within your Seam project. It supports multiple levels of security *granularity*, making it capable of performing either simple role-based security checks, or at the other end of the scale complex rule-based permission checks on domain objects using JBoss Rules.

23.1. Overview

The Seam security API provides the following features for securing a Seam application.

23.1.1. JAAS-based Authentication

The authentication component of the security API is based on JAAS, allowing authentication to be carried out against one or more configurable login modules. The security API provides a login module that delegates authentication to a Seam component, making it easy to authenticate a user using the application's domain objects.

23.1.2. Page Security

This feature allows direct access to page resources to be controlled based on the set of roles granted to the requesting user. Using a servlet filter, each request to a protected resource is validated to ensure that the user has the necessary roles to access the resource.

23.1.3. EL Integration

The `s:hasRole()` and `s:hasPermission()` EL functions can be used within the pages of an application to control which parts of a page are rendered based on the security level of the user. On the model side, component classes and/or their methods can also be annotated with the very same EL expressions to restrict the execution of methods based on the current security context.

23.1.4. Rule-based Authorization

By integrating with JBoss Rules, the security API is able to make security decisions by evaluating a set of user-defined security rules defined at deployment time. This feature allows any security checks to carry out a decision-making process which can be as simple or as complex as required, while taking into account the state of any objects asserted into the security context for the check.

23.2. Configuration

The first step in configuring Seam Security is to install the `SeamSecurityManager` and `SecurityConfiguration` components, as they are not installed by default. The `SeamSecurityManager` component is the core of the Seam Security API, and is responsible for loading security rules, and performing role and permission checks. `SecurityConfiguration` is used to load role and page security configuration from a configuration file (or can be extended to load this configuration from an alternative source, such as a database). The following entries in `components.xml` will install these two components:

```
<!DOCTYPE components PUBLIC "-//JBoss/Seam Component Configuration DTD 1.1//EN"
  "http://jboss.com/products/seam/components-1.1.dtd">

<components>
  <!-- Install a Security Configuration -->
  <component class="org.jboss.seam.security.config.SecurityConfiguration"/>

  <!-- Install the Seam Security Manager -->
  <component class="org.jboss.seam.security.SeamSecurityManager"/>

</components>
```

23.2.1. security-config.xml

The next step is to configure roles. While this step is optional (it is possible for a user to belong to a role even if it is not configured here), it is necessary if either 1) explicit permissions are required by your application, or 2) roles require membership of other roles. Create a file called `security-config.xml`, which goes in the `META-INF` directory of the Seam application's jar file. Here's an example `security-config.xml` file with a few roles defined.

```
<security-config>

  <roles>
    <role name="admin">
      <memberships>superuser</memberships>

      <permissions>
        <permission name="user" action="create"/>
        <permission name="user" action="modify"/>
        <permission name="user" action="delete"/>
      </permissions>
    </role>

    <role name="superuser">
      <memberships>user</memberships>

      <permissions>
        <permission name="account" action="create"/>
        <permission name="account" action="delete"/>
      </permissions>
    </role>

    <role name="user">
      <permissions>
        <permission name="customer" action="create"/>
        <permission name="customer" action="delete"/>
      </permissions>
    </role>

    <role name="guest">

    </role>

  </roles>

</security-config>
```

In the above example, we can see that there are four distinct roles: `admin`, `superuser`, `user` and `guest`. Strictly speaking, the `guest` entry is redundant as it declares no permissions nor contains any memberships of other

roles. What is most of interest here are the permissions and role memberships, which are explained in further detail as follows.

Explicit Permissions

These are permissions which are explicitly granted to members of a role. Explicit permissions are used to address simple security concerns, such as *"may this user create a new customer record?"*, or *"may this user view customer details?"*. While it is possible to perform a more complex contextual-based decision to grant a specific permission or not, explicit permissions allow simple decision-less permissions to be easily granted to certain groups of users. How these permissions are used will be explained further on.

Role memberships

Role memberships are a simple inheritence feature designed to make it easier to build a security model from the ground up. In the above example, it can be seen that the `admin` role contains a membership of the `superuser` role. What this means, is that any explicit permissions that the `superuser` has, whether they were granted to the `superuser` role directly or inherited from further up, are automatically granted to the `admin` user also, as a result of its membership.

The following table illustrates this concept by showing the permissions granted to each role based on the above `security-config.xml` example.

Table 23.1. Permissions granted to example roles

Role	Permission	Source
user	customer:create	Assigned directly
user	customer:delete	Assigned directly
superuser	customer:create	Inherited from user
superuser	customer:delete	Inherited from user
superuser	account:create	Assigned directly
superuser	account:delete	Assigned directly
admin	customer:create	Inherited from superuser
admin	customer:delete	Inherited from superuser
admin	account:create	Inherited from superuser
admin	account:delete	Inherited from superuser
admin	user:create	Assigned directly

Role	Permission	Source
admin	user:modify	Assigned directly
admin	user:delete	Assigned directly

Page security

Page security is used to restrict direct access to web resources based on the user's roles. It requires the Seam security filter (a servlet filter - see the next section) to be installed. To configure page security, `security-constraint` entries must be created within `security-config.xml` to control which web resources are accessible to which roles. The restricted web resources are defined using a url-pattern expression. The following example demonstrates how all of the `*.seam` resources in the `/secure` directory are restricted to users in the `admin` role.

```
<security-config>

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Secure Page</web-resource-name>
      <url-pattern>/secure/*</url-pattern>
    </web-resource-collection>

    <auth-constraint>
      <role-name>admin</role-name>
    </auth-constraint>
  </security-constraint>

</security-config>
```

This configuration may look familiar, because it is the same format defined by the servlet specification, and as such follows the same wildcard conventions. I.e. at the end of a pattern, `/*` matches any sequence of characters from that point forward, and the pattern `*.extension` matches any resources ending with `.extension`. An asterisk in any other position is not a wildcard.

So what happens when a user tries to access a page that they don't have the necessary rights for? By default, the security filter redirects these requests to the `/securityError.seam` page. This page can be overridden by setting the `securityErrorPage` property of the `SecurityConfiguration` component in `components.xml`:

```
  <component class="org.jboss.seam.security.config.SecurityConfiguration">
    <property name="securityErrorPage">/security/generalSecurityError.seam</property>
  </component>
```

23.2.2. Seam Security Filter

The Seam security filter is a servlet filter that provides page security features. It is an optional component in that if your application does not require page security, then there is no requirement to install the security filter. To configure which pages are secure, see the previous section on page security. To configure the security filter, add the following entries to `web.xml`:

```
  <!-- Seam security filter -->

  <filter>
    <filter-name>Seam Security Filter</filter-name>
    <filter-class>org.jboss.seam.security.filter.SeamSecurityFilter</filter-class>
  </filter>

  <filter-mapping>
    <filter-name>Seam Security Filter</filter-name>
    <url-pattern>*.seam</url-pattern>
  </filter-mapping>
```

23.2.3. security-rules.drl

This file also goes into the Seam application's `META-INF` directory. It contains all the rules that make up an application's authorization policy, and is described further down in the Authorization section.

23.3. Authentication

It is a relatively straight forward process to set up authentication. The first step is to configure the login modules that are to be used within the project by adding one or more `application-policy` entries to `security-config.xml`.

```
<application-policy>
  <authentication>
    <login-module code="org.jboss.seam.security.spi.SeamLoginModule" flag="required">
      <module-option name="authMethod">#{login.authenticate}</module-option>
    </login-module>
  </authentication>
</application-policy>
```

An `application-policy` without a specified name will be given a default name. It is possible to create multiple application policies if required, with each one having its own set of login modules.

```
<application-policy>  <!-- default policy -->
  <authentication>
    <login-module ...
  </authentication>
</application-policy>

<application-policy name="special">
  <authentication>
    <login-module ...
  </authentication>
</application-policy>
```

The login module configuration should look familiar if you've ever used JAAS before. Each login module should have its own `login-module` entry, specifying the fully qualified class name of the login module class,

plus the flag for the login module. It is also possible to configure additional options for each login module, by including `module-option` entries as children of the `login-module` entry. Flag values are found in the JSE API documentation for `javax.security.auth.login.Configuration`, but are repeated here for convenience:

Table 23.2. JAAS Configuration Flags

Flag	Description
Required	The LoginModule is required to succeed. If it succeeds or fails, authentication still continues to proceed down the LoginModule list.
Requisite	The LoginModule is required to succeed. If it succeeds, authentication continues down the LoginModule list. If it fails, control immediately returns to the application (authentication does not proceed down the LoginModule list).
Sufficient	The LoginModule is not required to succeed. If it does succeed, control immediately returns to the application (authentication does not proceed down the LoginModule list). If it fails, authentication continues down the LoginModule list.
Optional	The LoginModule is not required to succeed. If it succeeds or fails, authentication still continues to proceed down the LoginModule list.

23.3.1. Using `SeamLoginModule` to authenticate

One of the simplest ways to authenticate a user is to delegate the authentication to a Seam component. There is a special login module, `SeamLoginModule` that allows this. Configure this login module in `security-config.xml`, and in its `authMethod` option specify an EL method expression. By default the method that you specify should accept the parameters [`java.lang.String, java.lang.String, java.util.Set`], however this can be customised (see section further down). The third `java.util.Set` parameter is a set to which the authentication method should add any roles that the authenticating user is a member of. Here's a complete example:

```
@Name("authenticator")
public class AuthenticatorAction
{
  @In(create=true)
    private EntityManager entityManager;

  public boolean authenticate(String username, String password, Set<String> roles)
  {
    try
    {
       User user = (User) entityManager.createQuery(
          "from User where username = :username and password = :password")
          .setParameter("username", username)
```

```
          .setParameter("password", password)
          .getSingleResult();

       for (UserRole r : user.getRoles())
          roles.add(r.getName());

       return true;
    }
    catch (NoResultException ex)
    {
       log.warn("Invalid username/password");
       return false;
    }
  }
}
```

Based on the above example, the EL expression that would need to be specified in the `authMethod` option for the login module would be `#{authenticator.authenticate}`.

23.3.2. Logging in the user

Now that the login module has been configured, and an authentication method written, all that is left is to perform a JAAS login. This is the easiest step, as shown by the following example:

```
   public void login()
   {
      try
      {
         CallbackHandler cbh = SeamSecurityManager.instance().createCallbackHandler(
               user.getUsername(), user.getPassword());

         LoginContext lc = SeamSecurityManager.instance().createLoginContext(null, cbh);
         lc.login();
      }
      catch (LoginException ex)
      {
         FacesMessages.instance().add("Invalid login");
      }
   }
```

The first thing that happens in the above code is the creation of a `CallbackHandler` which is going to be responsible for providing the user's credentials (their username and password) to the login module. The `CallbackHandler` is created by a convenience method in `SeamSecurityManager`, and knows how to handle `NameCallback` and `PasswordCallback` callback types.

The next thing that happens is the creation of a `LoginContext`. There is a factory method in `SeamSecurityManager` for creating this, as the login context doesn't use the standard configuration (it uses an application-specific configuration). If the application policy isn't configured with a name, then it will have a default name and the String parameter passed to `createLoginContext()` can be null. If your application has multiple application policies configured, then you can specify which one to use by providing the policy name to `createLoginContext()`:

```
  LoginContext lc = SeamSecurityManager.instance().createLoginContext("special", cbh);
```

```
```

The final step is the call to `lc.login()`, which calls each of the configured login modules in turn, passing in the callback handler to each one and performing an authentication based on the configured login module flags.

Once the user is successfully authenticated, calls to `Identity.loggedIn()` will return true for the duration of the session.

23.3.3. Customising the Authentication process

In some situations it may be necessary to perform authentication that isn't based on a simple username/password combination. The good news is that this is not very difficult, however it requires a few additional steps. This section will walk through an example where authentication requires a Company ID in addition to a username and password.

The first step is to subclass the `SeamLoginModule` class and override the `getLoginParams()` method to return the appropriate parameters for the custom authentication method.

```
public class CustomLoginModule extends SeamLoginModule
{
   @Override public Object[] getLoginParams()
      throws Exception
   {
      CompanyCallback cbCompany = new CompanyCallback("Enter company ID");
      NameCallback cbName = new NameCallback("Enter username");
      PasswordCallback cbPassword = new PasswordCallback("Enter password", false);

      // Get the username and password from the callback handler
      callbackHandler.handle(new Callback[] { cbCompany, cbName, cbPassword });
      username = cbName.getName();

      return new Object[] { cbCompany.getCompanyId(), username,
                            new String(cbPassword.getPassword()), roles };
   }
}
```

Since `CompanyCallback` is a custom callback it needs to be created also:

```
public class CompanyCallback implements Callback, Serializable {
  private String prompt;
  private int companyId;
  public CompanyCallback(String prompt) {
    if (prompt == null || "".equals(prompt))
      throw new IllegalArgumentException();
    this.prompt = prompt;
  }
  public String getPrompt() {
    return prompt;
  }
  public int getCompanyId() {
    return companyId;
  }
  public void setCompanyId(int companyId) {
    this.companyId = companyId;
  }
}
```

```
```

So far so good, now it's time to configure the custom login module in `security-config.xml`:

```
  <loginmodules>
    <loginmodule class="com.acme.security.CustomLoginModule" flag="required">
      <option name="paramTypes">java.lang.Integer,java.lang.String,java.lang.String,java.util.Set</opt
      <option name="authMethod">#{authenticator.authenticate}</option>
    </loginmodule>
  </loginmodules>
```

The important thing to note above is that the `paramTypes` option is specified, based on the customised parameters that are going to be sent to the authentication method. Now that they have been configured, it is time to write the authentication method:

```
@Name("authenticator")
public class CustomAuthenticator {
  @In(create=true)
  private EntityManager entityManager;

  public boolean authenticate(int companyId, String username, String password, Set<String> roles)
  {
    try
    {
       User user = (User) entityManager.createQuery(
          "from User where companyId = :companyId and username = :username and password = :password")
          .setParameter("companyId", companyId)
          .setParameter("username", username)
          .setParameter("password", password)
          .getSingleResult();

       for (UserRole r : user.getRoles())
          roles.add(r.getName());

       return true;
    }
    catch (NoResultException ex)
    {
       log.warn("Invalid username/password");
       return false;
    }
  }
}
```

As can be seen in the above example, the customised authentication method contains the additional `companyId` parameter. The last thing to do is write a login method:

```
@Name("loginAction")
public class LoginAction implements LoginLocal {
   @In(create = true) EntityManager entityManager;
   @In(required = false) User user;

   public void login()
   {
      try
      {
```

```
         CallbackHandler cbh = createCallbackHandler(user.getCompanyId(),
               user.getUsername(), user.getPassword());

         LoginContext lc = SeamSecurityManager.instance().createLoginContext(cbh);
         lc.login();
      }
      catch (LoginException ex)
      {
         FacesMessages.instance().add("Invalid login");
      }
   }

   private CallbackHandler createCallbackHandler(final int companyId,
         final String username, final String password)
   {
      return new CallbackHandler() {
         public void handle(Callback[] callbacks)
            throws IOException, UnsupportedCallbackException
         {
            for (int i = 0; i < callbacks.length; i++)
            {
               if (callbacks[i] instanceof CompanyCallback)
                  ((CompanyCallback) callbacks[i]).setCompanyId(companyId);
               else if (callbacks[i] instanceof NameCallback)
                  ((NameCallback) callbacks[i]).setName(username);
               else if (callbacks[i] instanceof PasswordCallback)
                  ((PasswordCallback) callbacks[i]).setPassword(password.toCharArray());
               else
                  throw new UnsupportedCallbackException(callbacks[i],
                        "Unsupported callback");
            }

         }
      };
   }
```

The most significant thing to note in the above code is the `createCallbackHandler()` method. It returns a callback handler that knows how to handle a `CompanyCallback`, in addition to the standard `NameCallback` and `PasswordCallback`.

That wraps up the creation of a customised authentication process. Based on the above steps, it should be possible to create a customised authentication based on any combination of credentials.

23.4. Authorization

The authorization features of the Seam security API make it possible to restrict access to a Seam component based on the roles and permissions granted to the authenticated user. Security restrictions are defined using EL expressions, and configured by annotating either a Seam component method, or the component class itself, with the `@Restrict` annotation.

23.4.1. Types of authorization checks

The Seam security API provides two types of authorization checks; role checks and permission checks. Role checks are simple checks to determine if a user is a member of a specific role. They are equivalent in function to the `isUserInRole()` method found within the servlet specification. Role checks can be performed by using the `s:hasRole()` EL function. Here's a few examples of role checks.

This first example demonstrates how to restrict access to the Seam component `secureAction` to all users be-

sides those in the `admin` role.

```
@Name("secureAction")
@Restrict("#{s:hasRole('admin')}")
public class SecureAction {
   // ...
}
```

This example demonstrates how to restrict access to a method of a Seam component to users in the `superuser` role.

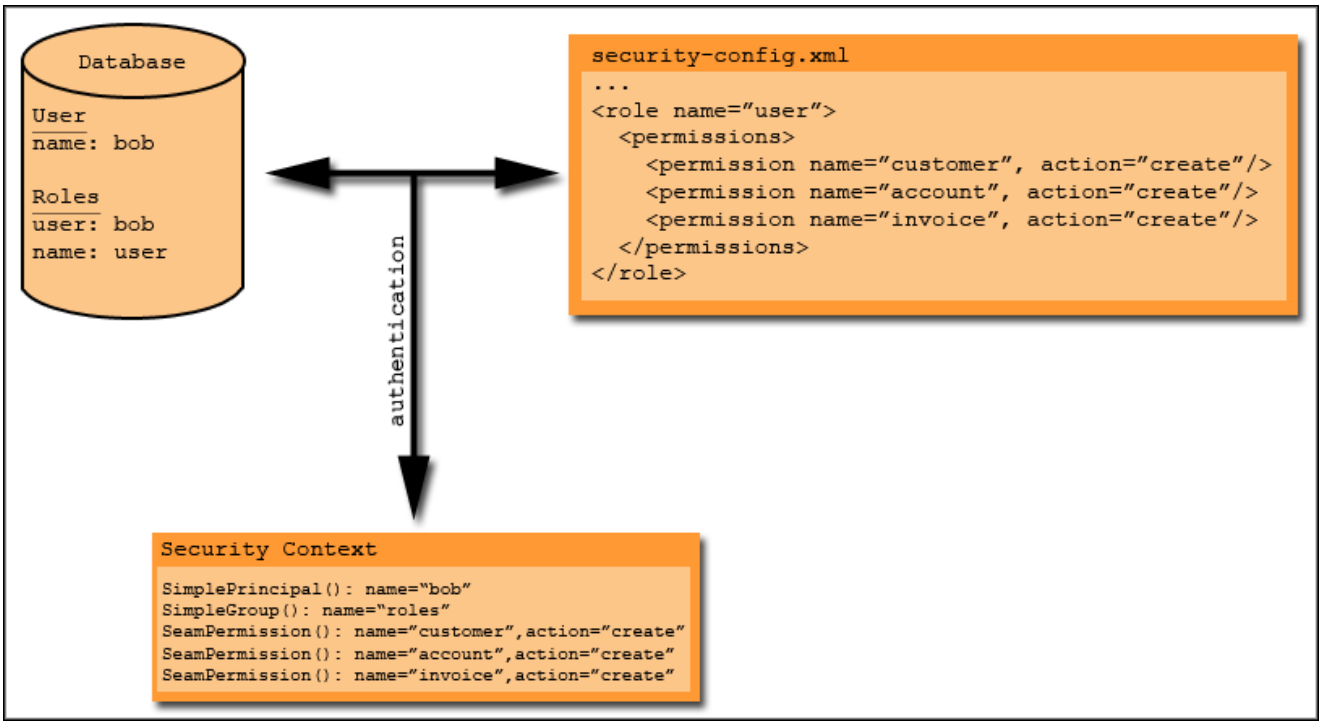
```
  @Restrict("#{s:hasRole('superuser')}")
  public void secureMethod {
    // ...
  }
```

This last example shows how an inline security check can be performed within the body of a method, rather than using the `@Restrict` annotation.

```
  public String changeUserPassword() {
    // code here
    Identity.instance().checkRestriction("#{s:hasRole('superuser')}");
    // code here
  }
```

23.4.2. The Security Context

The first time that a permission check is performed after a user is authenticated, a security context is created for that user and placed into the user's session context. This security context (which in fact is an instance of `org.drools.WorkingMemory`) is populated with all of the `Principals` for the authenticated user, plus all of the explicitly granted permissions granted to the user's role/s, as instances of`org.jboss.seam.security.SeamPermission`. To make this a little clearer, here's a diagram:



In this diagram, the database contains a user "bob", who is a member of the "user" role. The security configuration file, `security-config.xml` defines a role called "user" that has the explicitly assigned permissions `customer:create`, `account:create` and `invoice:create`. After bob authenticates, his Security Context will contain all of the permissions granted to him through his role memberships, as well as any `Principals` created as a result of the authentication process. This includes his roles, which exist in a `SimpleGroup` principal with the name of "roles".

23.4.3. How do permission checks work?

23.4.4. Establishing a default security policy

As mentioned previously, permissions can either be explicitly granted, or they can be granted as the result of a rule-based decision. Strictly speaking though, *all* permissions must be granted via a security rule; this includes explicit permissions also. By having all permissioning under the control of the rule engine, it is easier to configure any custom security requirements for your application. So what does this mean? For starters, granting explicit permissions without specifying a minimal security policy will mean that any security checks for those permissions will fail. What does a minimal security policy look like? Here's an example `security-rules.drl` file with a default security policy defined:

```
package MyProjectPermissions;

import org.jboss.seam.security.Identity;
import org.jboss.seam.security.rules.PermissionCheck;
import org.jboss.seam.security.SeamPermission;

rule DefaultPolicy
  salience -10
  activation-group "permissions"
when
  c: PermissionCheck(granted == false)
  p: SeamPermission()
  eval( p.getName().equals(c.getName()) && p.getAction().equals(c.getAction()))
then
```

```
  c.grant();
end;
```

If you're familiar with JBoss Rules then this example should make at least some sense already. In a nutshell, what this rule does is "catch" any permission checks that haven't already been granted by any higher priority rules, and grant the permission if a `SeamPermission` instance having the same name and action as the `PermissionCheck` exists within the security context.

A couple of notes about this rule; first of all, the `salience -10` line means that this rule will have a lower priority than any other rules with a higher salience (the default salience if not specified is 0). This assigns rules for any dynamic permissions (that don't have an explicit `SeamPermission` in the security context) a higher firing priority. Secondly, the `activation-group` value means that once the first rule within the activation group fires, no other rules within the same activation group will fire.