Seam - Les composants contextuels

Un Serveur d'application pour Java Enterprise

2.2.3-SNAPSHOT

par Gavin King, Pete Muir, Norman Richards, Shane Bryzak, Michael Yuan, Mike Youngstrom, Christian Bauer, Jay Balunas, Dan Allen, Max Rydahl Andersen, Emmanuel Bernard, Nicklas Karlsson, Daniel Roth, Matt Drees, Jacob Orshalick, Denis Forveille, Marek Novotny, et Jozef Hartinger

edited by Samson Kittoli

and thanks to James Cobb (Graphic Design), Cheyenne Weaver (Graphic Design), Mark Newton, Steve Ebersole, Michael Courcy (French Translation), Nicola Benaglia (Italian Translation), Stefano Travelli (Italian Translation), Francesco Milesi (Italian Translation), et Japan JBoss User Group (Japanese Translation)

Introduction à JBoss Seam

Seam est un serveur d'application pour Java EE5. Il a été inspiré par les principes suivants:

Une seule sorte de "truc"

Seam défini un modèle de composant uniforme pour toute la logique métier de votre application. Un composant Seam peut être avec état, avec état associé avec au choix un des contextes bien définie, ce qui inclus *le contexte de processus métier* à exécution longue et *le contexte conversationnel* qui est préservé au travers de plusieurs requêtes web dans les intéractions utilisateur.

Il n'y a pas de distinction entre les composants tierces de présentations et les composants de logique métier dans Seam. Vous pouvez mettre en plusieurs couches votre application en accord avec l'architecture de votre choix, au lieu d'être forcé de tordre votre logique d'application dans un schéma avec des couches anormales en étant forcé par une combinaisons de tuyaux des serveurs d'applications que vous utilisez aujourd'hui.

A la différence des composants pur Java EE ou J2EE, les composants Seam peuvent accèder *simultanément* aux états associés avec la requête web et avec l'état conservé dans les ressources transactionnelles (sans avoir besoin de propager l'état de la requête web avec des paramètres de la méthode). Vous pouvez objecter que la conception en couche de l'application qui vous est imposée par la vieille plateforme J2EE est une Bonne Chose. Et bien, rien ne vous empêche de ne pas créer une architecture en couche équivalent en utilisant Seam — la différence est que *vous* pouvez convevoir votre application et décider quel mis en couche vous avez et comment elles vont fonctionner ensemble.

Intégrer JSF avec EJB3.0

JSF et EJB 3.0 sont deux des meilleures fonctionnalités récentes de Java EE5. EJB3 est un modèle de composant tout nouveau pour l'applicatif métier côté serveur et la logique de persistance. Principalement, JSF est un bon modèle par composants pour la partie présentation. Malheureusement, le modèle de composant n'est pas capable de résoudre tous les problèmes dans la programmation elle-même. En fait, JSF et EJB3 fonctionnent bien ensemble. Mais les spécifications Java EE 5 ne fournissent pas un moyen standard pour intégrer les 2 modèles de composants. Heureusement, les créateurs de ces 2 modèles ont prévu cette situation et fournissent des points d'extensions standard pour permettre l'extension et l'intégration d'autres serveurs d'applications.

Seam unifie les modèles de composants de JSF et d'EJB3, élimant le code glue, et laissant le développeur se concentrer sur les problèmes du métier.

Il est possible d'écrire des applications Seam avec "tout" est un EJB. Cela peut être une grosse surprise si vous avez l'habitude de pensez aux EJBs comme des gros grains, des objets "super-lourd". Cependant, la version 3.0 a complètement changé la nature de l'EJB depuis le point de vu du développeur. Un EJB est un objet bien dimensionné — rien de bien complexe qu'un JavaBean annoté. Seam vous encourage même à utiliser les beans de sessions comme des écouteurs d'actions JSF!

D'un autre côté, si vous préférez ne pas adopter EJB 3.0 pour l'instant, vous n'avez pas à le faire. Virtuellement toute classe Java peut être un composant de Seam et Seam fourni toute les fonctionnalité que vous attendez pour un container "poid-mouche" et bien plus, pour chaque composant, EJB ou autre.

Integration AJAX

Seam utilise deux excelents solutions open source AJAX basé sur JSF: JBoss RichFace et ICEfaces. Ces solutions permettent d'ajouter les capacités d'AJAX à vos interfaces utilisateur sans avoir besoin d'écrire du code Javascript.

Autrement, Seam fourni aussi un couche distante développée en Javascript vous permettant d'appeler les composants de manière assynchrone depuis le côté client JavaScript sans avoir besoin d'une couche d'action intermédiaire. Vous pouvez même sousrire à des sujets JMS côté serveur et recevoir les messages via un push AJAX.

Aucune de ces approches ne fonctionnerai bien si il n'y a pas la concurence livrée par Seam et un gestionnaire d'état qui assure que toute les requêtes concurentes assynchrones et à fines granularités ne sont pas gérées de amnière sure et éfficace du côté serveur.

Processus métier intégré comme un premier constructeur de classe

Optionnellement, Seam intègre la gestion des processus métier de façon transparent via jBPM. Vous n'allez pas y croire qu' il est vraiment facile d'implémenter des enchaînements de taches complexes en utilisant jBPM et Seam.

Seam permet même de définir des enchainements de pages tierces de présentation en utilisant le même langage (jPDL) que jBPM utilise pour la définition de processus métier.

JSF fourni un modèle incroyablement riche d'évènement pour la couche présentation tierce. Seam améliore ce modèle en exposant les processus métier jBPM liant les évènement par un mécanisme de gestion de ces même évènements, fournissant un modèle d'évènement uniforme pour le modèle uniforme de composant de Seam.

Gestionnaire d'état déclaratif

Nous somme habitué au concept de gestionnaire de transaction déclaratif et à la sécurité déclarative depuis les tout premier jours des EJB. EJB 3.0 introduit même un gestionnaire de persistance contextuelle déclarative. Ceci sont les trois exemples du grand problème du gestionnaire d'état qui est associé avec un *contexte* particulie, tout tant s'assurant que tout les besoins en nettoyage interviennent quand le contexte s'arrête. Seam rends le concept de gestionnaire d'état déclaratif bien meilleur et l'applique à *l'état applicatif*. Traditionnellement, les applications J2EE doivent presque toujours implémenter manuellement un gestionnaire d'état, en affectant et en récupérant les attributs de requête et de session de servlet. Cette approche du gestionnaire d'état est la source de nombreux bugs et de fuite de mémoire quand les applications n'arrivent pas à nettoyer les attributs de la session, ou quand les données de session associées avec différentes enchaînements de tâches cohabitent dans une application multi fenêtrée. Seam dispose du potentiel pour éliminer presque complètement ce genre de bugs.

Une application gestionnaire d'état déclaratif est devenue possible par la richesse du *modèle contextuel* définis par Seam. Seam étends le modèle de contexte définie par les spécifications servlet c — request, session, application c — avec deux nouveaux contextes : conversation et processus métier c — ces derniers prennent tous leur sens du points de vue de la logique métier.

Vous allez être impréssionné comment les choses deviennent plus simple une fois que vous commencerez à utiliser les conversations. Avez vous déjà connu la douleur dans la gestion de correspondance d'association chargées à la demande avec une solution ORM comme Hibernate ou JPA? Les contextes de persistance d'étendue conversationnelle de Seam font que vous allez rarement voir une `LazyInitializationException`. Avez-vous déjà eu des problèmes avec le bouton rafraichir ? Le bouton précédent? Avec une soumission de formulaire dupliquée ? Avec des messages propagées au travers d'un post-ensuite-redirection? Le gestionnaire de conversation de Seam résoud tous ces problèmes sans que vous n'aillez besoin de réellement y penser. Ils sont tous des symptomes d'une architecture de gestions d'état morcelée qui a prévalue depuis les premiers temps du web.

La bijection

La notion d'*Inversion de contrôle* ou *d'injection dépendante* existe à la fois dans JSF et dans EJB3, tout autant que "les containeurs légers" aux multiples dénominations. La plus part de ces containers parlent d'injection de composants en implémentant *des services sans état*. Même quand l'injection de composants avec état est disponible (comme dans JSF), il est virtuellement inutile pour la gestion de l'état de l'application car la durée de vie du composants avec état ne peut être définie avec suffisamment de flexibilité et parce que l'appartenance des composants à une étendue plus grande ne peuvent pas être injecté dans des composants appartenant à des étendues plus réduites.

La *Bijection*diffère de l'inversion de contrôle (IoC) dans le fait qu'il est *dynamique*, *contextuel* et *bidirectionnel*. Vous pouvez penser à ce mécanisme comme des variables de contextes avec alias (les noms dans les différents contextes correspondant au thread courrant) pour des attributs de composants. La bijection autorise un assemblage automatisé de composants avec état par le containeur. Il autorise le composant de manière sécurisante et facile de manipuler les valeurs des variables de contexte, juste par assignation d'un attribut d'un composant.

Gestionnaire d'espace de travail et navigation multi-fenétrée

Les applications de Seam permettent à l'utilisateur de librement basculer entre les multiples onglets du navigateur, chacun associé avec une conversation différentes, isolé de manière sécurisée. Les applications peuvent même profiter du *gestionnaire d'espace de travail* autorisant l'utilisateur à basculer entre des conversations (les espaces de travail) dans un seul onglet du navigateur. Seam ne fournit pas seulement des opération correcte multi-fenétrée mais aussi des opération simulant le multi-fenetré dans une seule fenètre!

Préfère les annotations au XML

Traditionnellement, la communauté Java a été dans un état de grande confusion à propos de ce qui précisément compte comme méta-information dans une configuration. J2EE et les

containeurs "légers" ont fourni des descripteurs de déploiement basé sur XML à la fois pour les choses qui sont clairement configurable entre les différents déploiement du système et pour tout autre chose comme déclaration qui ne peut pas être facilement exprimé en Java. Les annotations de Java 5 ont changé tout cela.

EJB 3.0 utilise les annotations et la "configuration par exception" est la façon la plus facile de fournir de l'informations au containeur dans une forme déclarative. Malheureusement, JSF est encore lourdement dépendant de fichiers de configuration XML verbeux. Seam étend les annotations fournies par EJB3.0 avec un groupe de d'annotations pour le gestionnaire d'état déclaratif et une démarcation du contexte déclarative. Ceci vous permet d'éliminer les complexes déclaration de gestion des beans de JSF et de reduire le XML nécéssaire à ce qui est du ressort réellement du XML (les règles de navigation de JSF).

L'intégration des tests est facile

Les composants Seam, en étant de pures classes Java, sont par nature testable unitairement. Mais pour des applications complexes, le fait de tester unitairement n'est pas suffisant. Tester unitairement est traditionnellement une tâche fatigante et complexe pour les applications web Java. Cependant, Seam fourni pour le test des applications Seam un fonctionnalité au coeur du serveur d'applidcation Vous pouvez facilement écrire des tests JUnit ou TestNG pour reproduire une parfaite interaction avec un utilisateur, sollicitant tous les composants du système indépendamment de vues (la page JSP ou Facelets). Vous pouvez exécuter ces tests directement depuis votre IDE, là où Seam va automatiquement déployer les composants EJB en utilisant Jboss Embeddable.

La spécification n'est pas perfaite

Vous pensez que la dernière incarnation de Java EE est géniale. Mais nous savons qu'elle n'est pas parfaite. Là où il y a des trous dans la spécification (par exemple les limitations dans le cycle de vie de JSF pour les requêtes GET), Seam les règles lui-même. Et les auteurs de Seam travaille avec le groupe d'experts JCP pour faire en sorte que ces corrections seront conservé dans les prochaines révisions des standards.

Il y a bien plus pour une application web que de confectionner des pages HTML

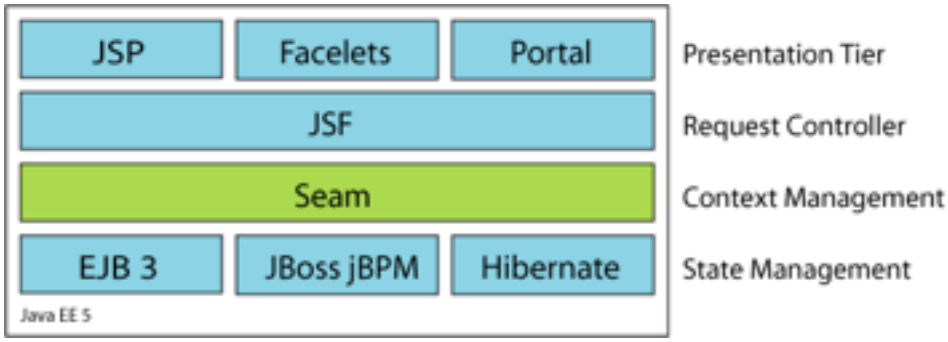
Aujourd'huis les serveurs d'applications pensent trop petit. Ils vous permettent d'obtenir les entrées d'un formulaire utilisateur dans vos objets Java. Et ensuite ils vous laissent vous débrouiller. Un serveur d'application web vraiment complet devrait régler les problèmes comme la persistance, la concurence, l'assynchronisme, la gestion d'état, la sécurité, l'email, les messages, le PDF, la génération de diagrame, l'enchainement de tâche, le rendu du formatage wiki, les services web, la mise en cache et bien plus encore. Une fois que vous aurez plogeés sous la surface de Seam, vous allez être impréssionné par comment les problèmes deviennent plus simple....

Seam intègre JPA et Hibernate3 pour la persistance, le EJB Timer Service et Quartz pour l'assynchronisme version allégée, jBPM pour l'enchainement de tâches, JBoss Rules pour les règles métiers, Meldware Mail pour les emails, Hibernate Search et Lucene pour la recherche plein texte, JMS pour les messages et JBoss Cache pour la mise en cache de fragments. Les couches de Seam son un serveur d'application de sécurité innovant basés sur des règles au

desus de JAAS et de JBoss Rules. Il ya aussi des bibliothèques de tag JSF pour le rendu de PDF, l'envoie des emails, les diagrammes et le texte wiki. Les composants de Seam peuvent même être appelés de mmanière synchrones comme des services web, asynchronisme depuis le JavaScript côté client ou avec Google Web Toolkit ou bien sur directement depuis JSF.

Prêt au départ maintenant!

Seam fonctionne dans n'importe quel serveur d'application Java EE et fonctionne même dans Tomcat. Si votre environement supporte EJB3.0, génial!Si cen'est pas le cas, pas de problème, vous pouvez utiliser le gestionnaire de transaction livré dans Seam avec JPA ou Hibernate3 pour la persistance. Ou, vous pouvez deployer JBoss Embedded dans Tomcat, et avoir le support plein et entier d'EJB 3.0.



Il est clair que la combinaison de Seam, JSF et EJB3 est *la* façon la plus simple d'écrire une application web complexe en Java. Vous ne réalisez pas le peut de code qu'il nécéssite!

1. Contribuez à Seam

Visitez *SeamFramework.org* [http://www.seamframework.org/Community/Contribute] pour découvrir comment contribuer à Seam!

Seam Tutorial

1.1. Using the Seam examples

Seam provides a number of example applications demonstrating how to use the various features of Seam. This tutorial will guide you through a few of those examples to help you get started learning Seam. The Seam examples are located in the `examples` subdirectory of the Seam distribution. The registration example, which will be the first example we look at, is in the `examples/registration` directory.

Each example has the same directory structure:

- The `view` directory contains view-related files such as web page templates, images and stylesheets.
- The `resources` directory contains deployment descriptors and other configuration files.
- The `src` directory contains the application source code.

The example applications run both on JBoss AS and Tomcat with no additional configuration. The following sections will explain the procedure in both cases. Note that all the examples are built and run from the Ant `build.xml`, so you'll need a recent version of Ant installed before you get started.

1.1.1. Running the examples on JBoss AS

The examples are configured for use on JBoss AS 4.2 or 5.0. You'll need to set `jboss.home`, in the shared `build.properties` file in the root folder of your Seam installation, to the location of your JBoss AS installation.

Once you've set the location of JBoss AS and started the application server, you can build and deploy any example by typing `ant explode` in the the directory for that example. Any example that is packaged as an EAR deploys to a URL like `/seam-`*`example`*, where *`example`* is the name of the example folder, with one exception. If the example folder begins with seam, the prefix "seam" is ommitted. For instance, if JBoss AS is running on port 8080, the URL for the registration example is *`http://localhost:8080/seam-registration/`* [http://localhost:8080/seam-registration/], whereas the URL for the seamspace example is *`http://localhost:8080/seam-space/`* [http://localhost:8080/seam-space/].

If, on the other hand, the example gets packaged as a WAR, then it deploys to a URL like `/jboss-seam-`*`example`*. Most of the examples can be deployed as a WAR to Tomcat with Embedded JBoss by typing `ant tomcat.deploy`. Several of the examples can only be deployed as a WAR. Those examples are groovybooking, hibernate, jpa, and spring.

1.1.2. Running the examples on Tomcat

The examples are also configured for use on Tomcat 6.0. You will need to follow the instructions in *Section 30.6.1, « Installing Embedded JBoss »* for installing JBoss Embedded on Tomcat 6.0.

JBoss Embedded is only required to run the Seam demos that use EJB3 components on Tomcat. There are also examples of non-EJB3 applications that can be run on Tomcat without the use of JBoss Embedded.

You'll need to set `tomcat.home`, in the shared `build.properties` file in the root folder of your Seam installation, to the location of your Tomcat installation. make sure you set the location of your Tomcat.

You'll need to use a different Ant target when using Tomcat. Use `ant tomcat.deploy` in example subdirectory to build and deploy any example for Tomcat.

On Tomcat, the examples deploy to URLs like `/jboss-seam-example`, so for the registration example the URL would be *http://localhost:8080/jboss-seam-registration/* [http://localhost:8080/jboss-seam-registration/]. The same is true for examples that deploy as a WAR, as mentioned in the previous section.

1.1.3. Running the example tests

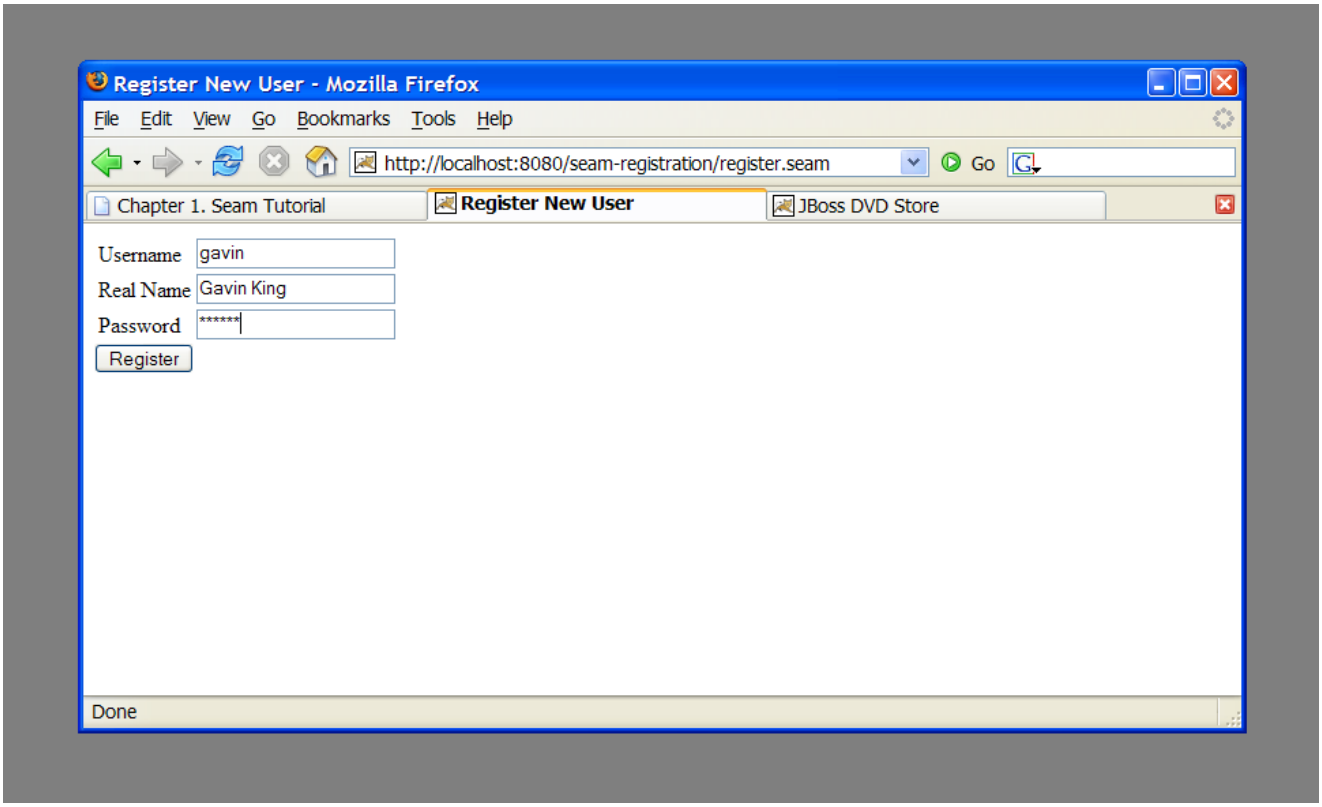
Most of the examples come with a suite of TestNG integration tests. The easiest way to run the tests is to run `ant test`. It is also possible to run the tests inside your IDE using the TestNG plugin. Consult the readme.txt in the examples directory of the Seam distribution for more information.

1.2. Your first Seam application: the registration example

The registration example is a simple 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 Facelets templates, 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.

Exemple 1.1. User.java

```
@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 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!

Exemple 1.2. RegisterAction.java

```
@Stateless                                                    (1)
@Name("register")
public class RegisterAction implements Register
{
   @In
   private User user;                                         (2)

   @PersistenceContext
   private EntityManager em;                                  (3)

   @Logger
   private Log log;                                           (4)

   public String register()
   {                                                          (5)
      List existing = em.createQuery(
         "select username from User where username = #{user.username}")
         .getResultList();                                    (6)

      if (existing.size()==0)
      {
         em.persist(user);
         log.info("Registered new user #{user.username}");
         return "/registered.xhtml";                          (7)
      }                                                       (8)
      else
      {
         FacesMessages.instance().add("User #{user.username} already exists");
         return null;                                         (9)
      }
   }

}
```

1. The EJB `@Stateless` annotation marks this class as a 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 session bean, a transaction is automatically begun when the `register()` method is called, and committed when it completes.
6. Notice that Seam lets you use a JSF EL expression inside EJB-QL. Under the covers, this results in an ordinary JPA `setParameter()` call on the standard JPA `Query` object. Nice, huh?
7. The `Log` API lets us easily display templated log messages which can also make use of JSF EL expressions.
8. 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.*
9. 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. (As of Seam 2.1, you can use `StatusMessages` instead to remove the semantic dependency on JSF). Built-in Seam components may be obtained by injection, or by calling the `instance()` method on the class of the built-in component.

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, which is the only sensible value.

Our session bean action listener performs the business and persistence logic for our mini-application. In more complex applications, we might need require a separate service layer. This is easy to achieve with Seam, but it's overkill for most web applications. Seam does not force you into any particular strategy for application layering, allowing your application to be as simple, or as complex, as you want.

Note that in this simple application, we've actually made it far more complex than it needs to be. If we had used the Seam application framework controllers, we would have eliminated all of our application code. However, then we wouldn't have had much of an application to explain.

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

Naturally, our session bean needs a local interface.

Exemple 1.3. Register.java

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

That's the end of the Java code. Now we'll look at the view.

1.2.1.4. The view: `register.xhtml` and `registered.xhtml`

The view pages for a Seam application could be implemented using any technology that supports JSF. In this example we use Facelets, because we think it's better than JSP.

Exemple 1.4. register.xhtml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:s="http://jboss.com/products/seam/taglib"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:f="http://java.sun.com/jsf/core">

  <head>
    <title>Register New User</title>
  </head>
  <body>
    <f:view>
      <h:form>
        <s:validateAll>
          <h:panelGrid columns="2">
            Username: <h:inputText value="#{user.username}" required="true"/>
            Real Name: <h:inputText value="#{user.name}" required="true"/>
            Password: <h:inputSecret value="#{user.password}" required="true"/>
          </h:panelGrid>
        </s:validateAll>
        <h:messages/>
        <h:commandButton 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.

Exemple 1.5. registered.xhtml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:f="http://java.sun.com/jsf/core">

   <head>
      <title>Successfully Registered New User</title>
   </head>
   <body>
      <f:view>
         Welcome, #{user.name}, you are successfully registered as #{user.username}.
      </f:view>
   </body>

</html>
```

This is a simple Facelets page using some inline EL. There's nothing specific to Seam here.

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

Since this is the first Seam app we've seen, we'll take a look at the deployment descriptors. Before we get into them, it is worth noting that Seam strongly values minimal configuration. These configuration files will be created for you when you create a Seam application. You'll never need to touch most of these files. We're presenting them now only to help you understand what all the pieces in the example are doing.

If you've used many Java frameworks before, you'll be used to having to declare 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:

Exemple 1.6. components.xml

```
<?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-2.2.xsd
      http://jboss.com/products/seam/components
      http://jboss.com/products/seam/components-2.2.xsd">

   <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`. The funny `@` symbols are there because our Ant build script puts the correct JNDI pattern in when we deploy the application, which it reads from the components.properties file. You learn more about how this process works in *Section 5.2, « Configuring components via `components.xml` »*.

1.2.1.6. 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.

Exemple 1.7. web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   version="2.5">

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

```
    </listener>

    <context-param>
        <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
        <param-value>.xhtml</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>

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

    <session-config>
        <session-timeout>10</session-timeout>
    </session-config>

</web-app>
```

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

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

Most Seam applications use JSF views as the presentation layer. So usually we'll need `faces-config.xml`. In our case, we are going to use Facelets for defining our views, so we need to tell JSF to use Facelets as its templating engine.

Exemple 1.8. faces-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config 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-facesconfig_1_2.xsd"
    version="1.2">

    <application>
```

```
    <view-handler>com.sun.facelets.FaceletViewHandler</view-handler>
  </application>

</faces-config>
```

Note that we don't need any JSF managed bean declarations! Our managed beans are annotated Seam components. In Seam applications, the `faces-config.xml` is used much less often than in plain JSF. Here, we are simply using it to enable Facelets as the view handler instead of JSP.

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 orchestration: navigation rules or jBPM process definitions. Seam's stand is 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.

1.2.1.8. 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.

```
<?xml version="1.0" encoding="UTF-8"?>
<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-jar_3_0.xsd"
  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>
```

1.2.1.9. 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/persistence/persistence_1_0.xsd"
   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>
```

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

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

Exemple 1.9. registration application

```
<?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/application_5.xsd"
   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>
      <ejb>jboss-seam.jar</ejb>
   </module>
   <module>
      <java>jboss-el.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 uses the JNDI pattern mentioned earlier to locate the stateless session bean, wraps it as a Seam component, and returns it. Seam then presents this component to JSF and JSF invokes the `register()` action listener method.

But Seam is not done yet. Seam intercepts the method call and injects the `User` entity from the Seam session context, before allowing the invocation to continue.

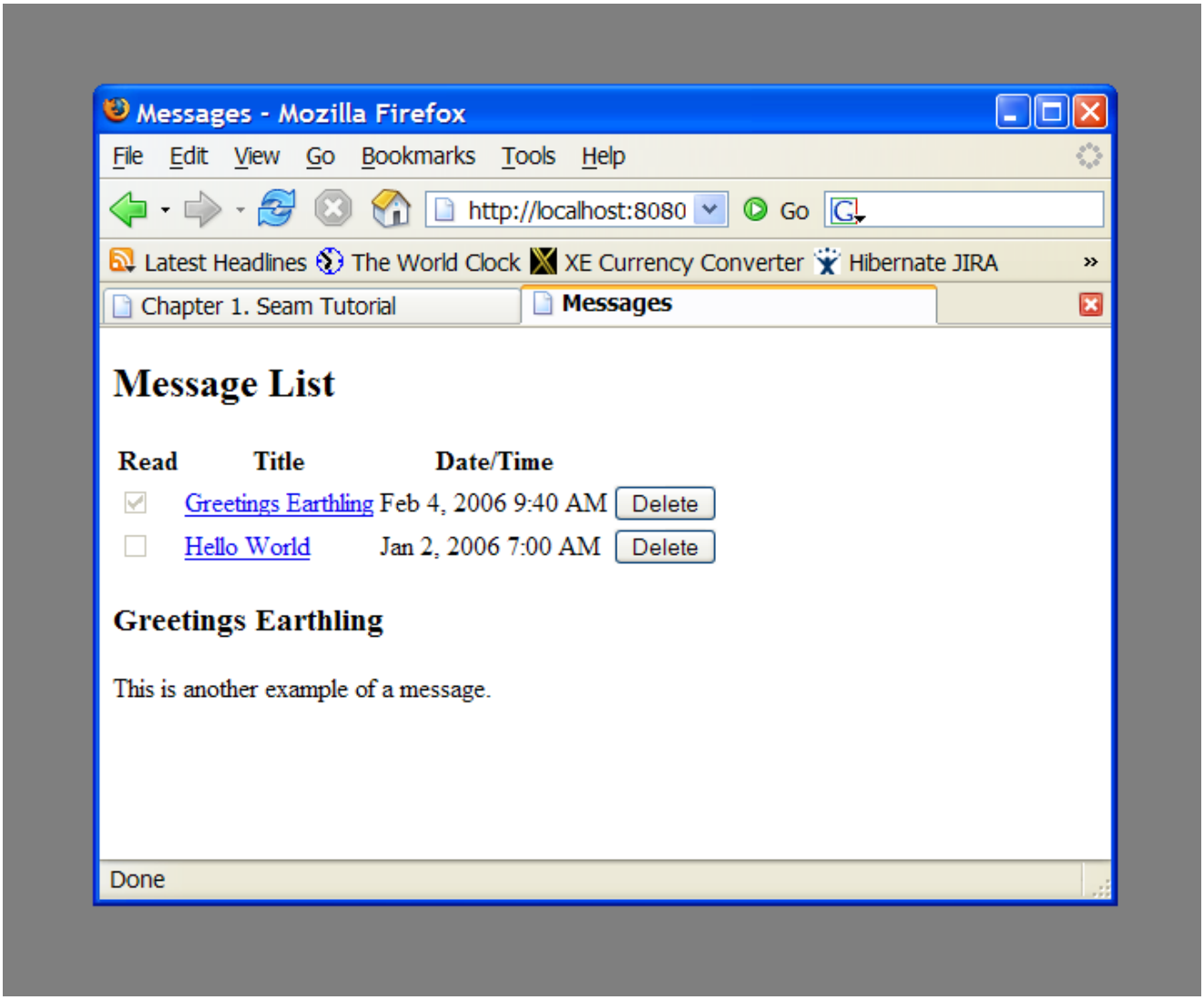
The `register()` method checks if a user with the entered username already exists. If so, an error message is queued with the `FacesMessages` component, and a null outcome is returned, causing a page redisplay. The `FacesMessages` component interpolates the JSF expression embedded in the message string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

If no user with that username exists, the `"/registered.xhtml"` outcome triggers a browser redirect to the `registered.xhtml` 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:

Exemple 1.10. Message.java

```
@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.

Exemple 1.11. MessageManagerBean.java

```
@Stateful
@Scope(SESSION)
@Name("messageManager")
public class MessageManagerBean implements Serializable, MessageManager
{
   @DataModel
   private List<Message> messageList;                                    (1)

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

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

   @Factory("messageList")
   public void findMessages()                                            (5)
   {
     messageList = em.createQuery("select msg 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
   public void destroy() {}                                              (8)
```

```
}
```

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 every 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 with no parameters marked `@Remove` that Seam uses to 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.

Exemple 1.12. MessageManager.java

```
@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.

Exemple 1.13. messages.jsp

```
<%@ 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

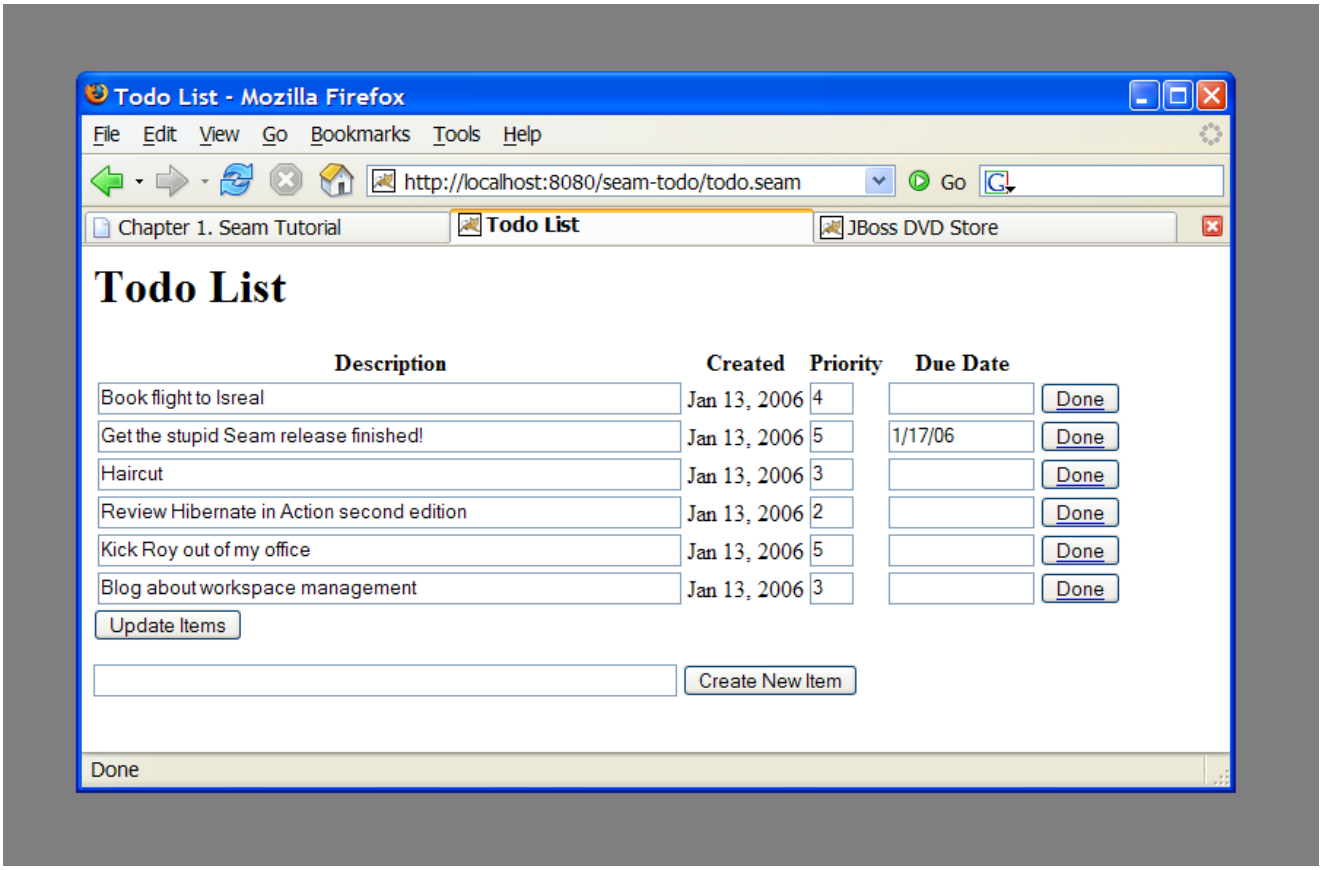
The first time we navigate to the `messages.jsp` page, the page will try to resolve the `messageList` context variable. Since this context variable is not initialized, Seam will call the factory method `findMessages()`, which performs a query against the database and results in a `DataModel` being outjected. 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:

Exemple 1.14. todo.jpdl.xml

```
<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 description 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.

Exemple 1.15. Login.java

```
@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:

Exemple 1.16. login.jsp

```
<%@ 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.

Exemple 1.17. TodoList.java

```
@Name("todoList")
public class TodoList
{
  private String description;


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

  public void setDescription(String description)
  {
    this.description = description;
  }
                                                                    (2)
  @CreateProcess(definition="todo")
  public void createTodo() {}
                                                                    (3)
  @StartTask @EndTask
  public void done() {}
```

```
}
```

1. The description property accepts user input from 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 core of the application is in `todo.jsp`:

Exemple 1.18. todo.jsp

```
<%@ 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.

Exemple 1.19. todo.jsp

```
<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>
```

Note

Seam provides a default JSF date converter for converting a string to a date (no time). Thus, the converter is not necessary for the field bound to `#{task.dueDate}`.

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>
```

1.4.2. How it works

After logging in, todo.jsp uses the `taskInstanceList` component to display a table of outstanding todo items for a the current user. Initially there are none. It also presents a form to enter a new entry. When the user types the todo item and hits the "Create New Item" button, `#{todoList.createTodo}` is called. This starts the todo process, as defined in `todo.jpdl.xml`.

The process instance is created, starting in the start state and immediately transition to the `todo` state, where a new task is created. The task description is set based on the user's input, which

was saved to `#{todoList.description}`. Then, the task is assigned to the current user, which was stored in the seam actor component. Note that in this example, the process has no extra process state. All the state in this example is stored in the task definition. The process and task information is stored in the database at the end of the request.

When `todo.jsp` is redisplayed, `taskInstanceList` now finds the task that was just created. The task is shown in an `h:dataTable`. The internal state of the task is displayed in each column: `#{task.description}`, `#{task.priority}`, `#{task.dueDate}`, etc... These fields can all be edited and saved back to the database.

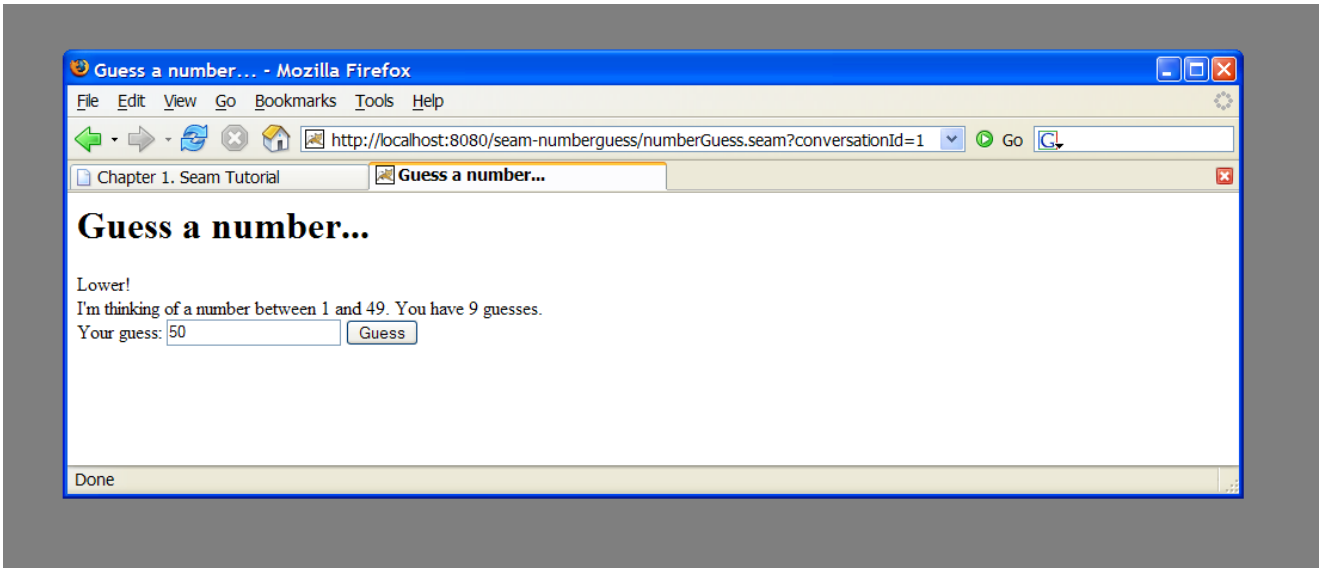
Each todo item also has "Done" button, which calls `#{todoList.done}`. The `todoList` component knows which task the button is for because each s:button specificies `taskInstance="#{task}"`, referring to the task for that particular line of of the table. The `@StartTast` and `@EndTask` annotations cause seam to make the task active and immediately complete the task. The original process then transitions into the `done` state, according to the process definition, where it ends. The state of the task and process are both updated in the database.

When `todo.jsp` is displayed again, the now-completed task is no longer displayed in the `taskInstanceList`, since that component only display active tasks for the user.

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:

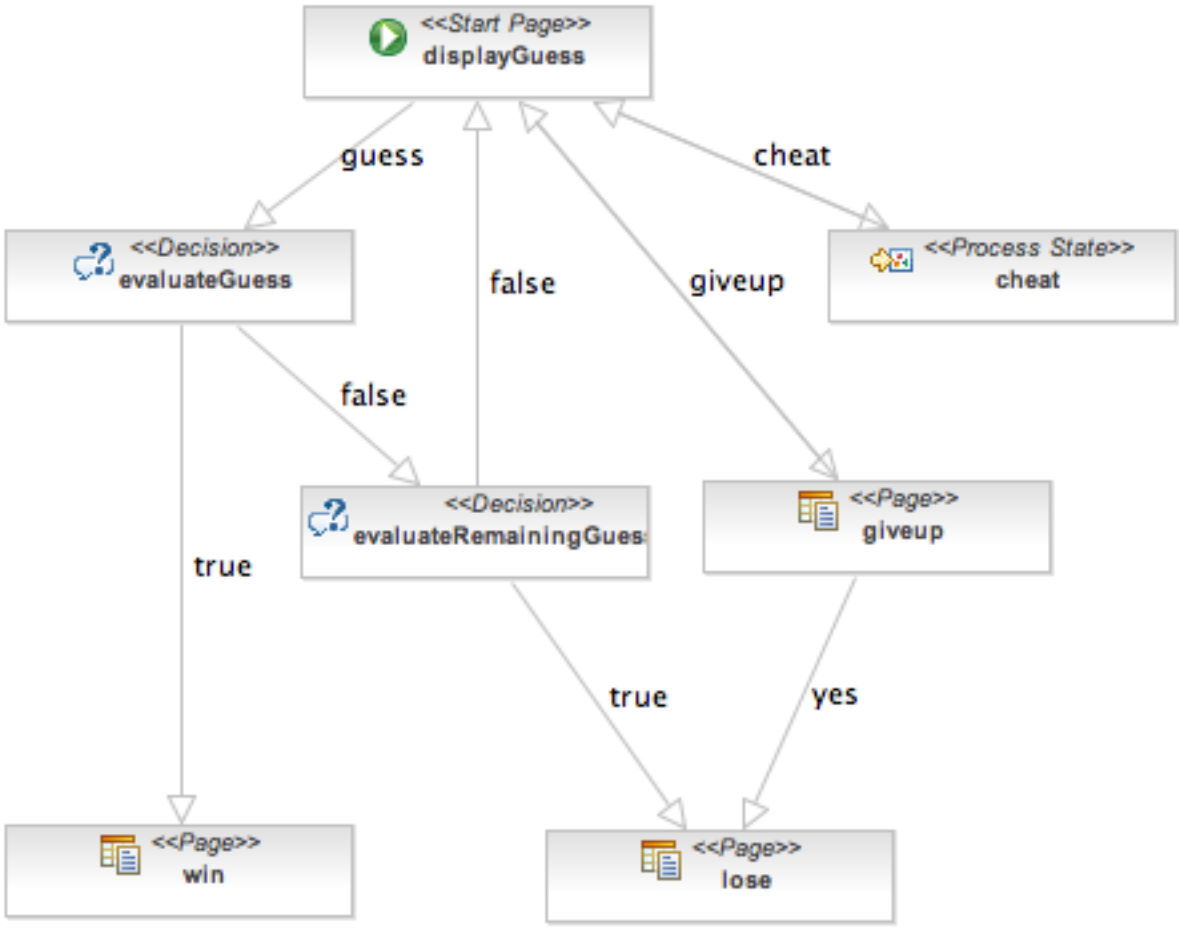
Exemple 1.20. pageflow.jpdl.xml

```
<pageflow-definition
      xmlns="http://jboss.com/products/seam/pageflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://jboss.com/products/seam/pageflow
                  http://jboss.com/products/seam/pageflow-2.2.xsd"
      name="numberGuess">


  <start-page name="displayGuess" view-id="/numberGuess.jspx">                1
    <redirect/>

    <transition name="guess" to="evaluateGuess">                              2

      <action expression="#{numberGuess.guess}"/>                             3
    </transition>
    <transition name="giveup" to="giveup"/>
    <transition name="cheat" to="cheat"/>
  </start-page>

                                                  4
  <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="giveup" view-id="/giveup.jspx">
    <redirect/>
    <transition name="yes" to="lose"/>
    <transition name="no" to="displayGuess"/>
  </page>

  <process-state name="cheat">
    <sub-process name="cheat"/>
```

```
    <transition to="displayGuess"/>
  </process-state>

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

  <page name="lose" view-id="/lose.jspx">
    <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 JBoss Developer Studio 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.jspx`:

Exemple 1.21. numberGuess.jspx

```
<<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
        xmlns:h="http://java.sun.com/jsf/html"
        xmlns:f="http://java.sun.com/jsf/core"
        xmlns:s="http://jboss.com/products/seam/taglib"
        xmlns="http://www.w3.org/1999/xhtml"
        version="2.0">
  <jsp:output doctype-root-element="html"
            doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
            doctype-system="http://www.w3c.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"/>
  <jsp:directive.page contentType="text/html"/>
```

```
<html>
<head>
  <title>Guess a number...</title>
  <link href="niceforms.css" rel="stylesheet" type="text/css" />
  <script language="javascript" type="text/javascript" src="niceforms.js" />
</head>
<body>
  <h1>Guess a number...</h1>
  <f:view>
    <h:form styleClass="niceform">

      <div>
      <h:messages globalOnly="true"/>
      <h:outputText value="Higher!"
            rendered="#{numberGuess.randomNumber gt numberGuess.currentGuess}"/>
      <h:outputText value="Lower!"
            rendered="#{numberGuess.randomNumber lt numberGuess.currentGuess}"/>
      </div>

      <div>
      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.
      </div>

      <div>
      Your guess:
      <h:inputText value="#{numberGuess.currentGuess}" id="inputGuess"
                required="true" size="3"
                rendered="#{(numberGuess.biggest-numberGuess.smallest) gt 20}">
        <f:validateLongRange maximum="#{numberGuess.biggest}"
                        minimum="#{numberGuess.smallest}"/>
      </h:inputText>
      <h:selectOneMenu value="#{numberGuess.currentGuess}"
                    id="selectGuessMenu" required="true"
                    rendered="#{(numberGuess.biggest-numberGuess.smallest) le 20 and
                              (numberGuess.biggest-numberGuess.smallest) gt 4}">
        <s:selectItems value="#{numberGuess.possibilities}" var="i" label="#{i}"/>
      </h:selectOneMenu>
      <h:selectOneRadio value="#{numberGuess.currentGuess}" id="selectGuessRadio"
                     required="true"
                     rendered="#{(numberGuess.biggest-numberGuess.smallest) le 4}">
        <s:selectItems value="#{numberGuess.possibilities}" var="i" label="#{i}"/>
```

```
      </h:selectOneRadio>
      <h:commandButton value="Guess" action="guess"/>
      <s:button value="Cheat" view="/confirm.jspx"/>
      <s:button value="Give up" action="giveup"/>
      </div>

      <div>
      <h:message for="inputGuess" style="color: red"/>
      </div>

    </h:form>
  </f:view>
 </body>
 </html>
</jsp:root>
```

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

The `win.jspx` page is predictable:

Exemple 1.22. win.jspx

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
        xmlns:h="http://java.sun.com/jsf/html"
        xmlns:f="http://java.sun.com/jsf/core"
        xmlns="http://www.w3.org/1999/xhtml"
        version="2.0">
 <jsp:output doctype-root-element="html"
           doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
           doctype-system="http://www.w3c.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"/>
 <jsp:directive.page contentType="text/html"/>
 <html>
 <head>
  <title>You won!</title>
  <link href="niceforms.css" rel="stylesheet" type="text/css" />
 </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.
   <h:outputText value="But you cheated, so it doesn't count!"
              rendered="#{numberGuess.cheat}"/>
   Would you like to <a href="numberGuess.seam">play again</a>?
```

```
    </f:view>
  </body>
  </html>
</jsp:root>
```

The `lose.jspx` looks roughly the same, so we'll skip over it.

Finally, we'll look at the actual application code:

Exemple 1.23. NumberGuess.java

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

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


   @Create                                                    (1)
   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;
  }

  public void cheated()
  {
    cheated = true;
  }

  public boolean isCheat() {
    return cheated;
  }

  public List<Integer> getPossibilities()
  {
    List<Integer> result = new ArrayList<Integer>();
    for(int i=smallest; i<=biggest; i++) result.add(i);
    return result;
  }

}
```

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.

The `pages.xml` file starts a Seam *conversation* (much more about that later), and specifies the pageflow definition to use for the conversation's page flow.

Exemple 1.24. pages.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<pages xmlns="http://jboss.com/products/seam/pages"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://jboss.com/products/seam/pages  http://jboss.com/products/seam/pages-2.2.xsd">

  <page view-id="/numberGuess.jspx">
    <begin-conversation join="true" pageflow="numberGuess"/>
  </page>
```

```
</pages>
```

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

We'll step through basic flow of the application. The game starts with the `numberGuess.jspx` view. When the page is first displayed, the `pages.xml` configuration causes conversation to begin and associates the `numberGuess` pageflow with that conversation. The pageflow starts with a `start-page` tag, which is a wait state, so the `numberGuess.xhtml` is rendered.

The view references the `numberGuess` component, causing a new instance to be created and stored in the conversation. The `@Create` method is called, initializing the state of the game. The view displays an `h:form` that allows the user to edit `#{numberGuess.currentGuess}`.

The "Guess" button triggers the `guess` action. Seam defers to the pageflow to handle the action, which says that the pageflow should transition to the `evaluateGuess` state, first invoking `#{numberGuess.guess}`, which updates the guess count and highest/lowest suggestions in the `numberGuess` component.

The `evaluateGuess` state checks the value of `#{numberGuess.correctGuess}` and transitions either to the `win` or `evaluatingRemainingGuesses` state. We'll assume the number was incorrect, in which case the pageflow transitions to `evaluatingRemainingGuesses`. That is also a decision state, which tests the `#{numberGuess.lastGuess}` state to determine whether or not the user has more guesses. If there are more guesses (`lastGuess` is `false`), we transition back to the original `displayGuess` state. Finally we've reached a page state, so the associated page `/numberGuess.jspx` is displayed. Since the page has a redirect element, Seam sends a redirect to the the user's browser, starting the process over.

We won't follow the state any more except to note that if on a future request either the `win` or the `lose` transition were taken, the user would be taken to either the `/win.jspx` or `/lose.jspx`. Both states specify that Seam should end the conversation, tossing away all the game state and pageflow state, before redirecting the user to the final page.

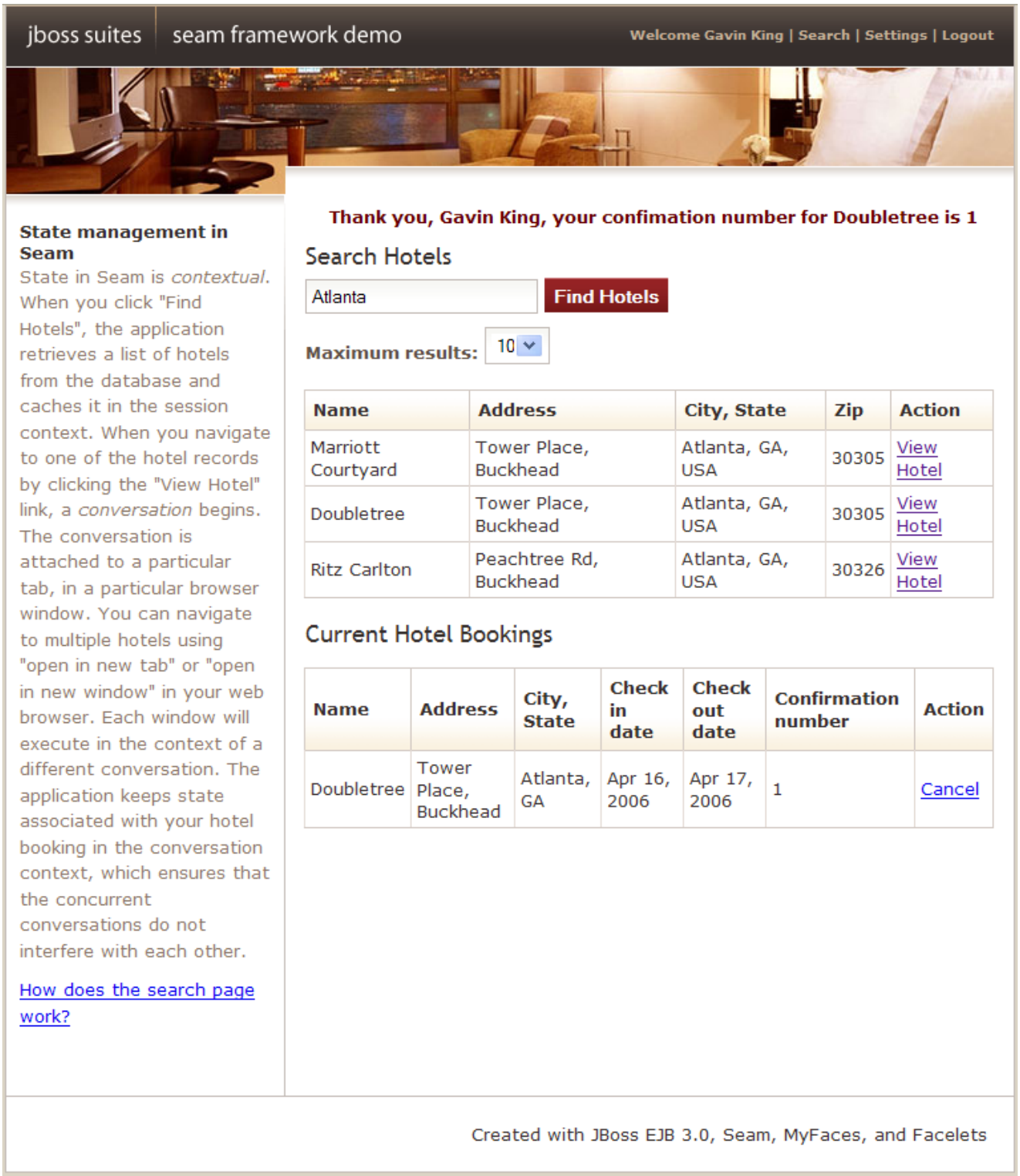
The numberguess example also contains Giveup and Cheat buttons. You should be able to trace the pageflow state for both actions relatively easily. Pay particular attention to the `cheat` transition, which loads a sub-process to handle that flow. Although it's overkill for this application, it does demonstrate how complex pageflows can be broken down into smaller parts to make them easier to understand.

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, « Using the Seam examples »*. Once you've successfully started the application, you can access it by pointing your browser to `http://localhost:8080/seam-booking/` [http://localhost:8080/seam-booking/]

The application uses six session beans for to implement the business logic for the listed features.

- `AuthenticatorAction` provides the login authentication logic.
- `BookingListAction` retrieves existing bookings for the currently logged in user.
- `ChangePasswordAction` updates the password of the currently logged in user.
- `HotelBookingAction` implements booking and confirmation functionality. This functionality is implemented as a *conversation*, so this is one of the most interesting classes in the application.
- `HotelSearchingAction` implements the hotel search functionality.
- `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

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 conversational state. Usually, Java web applications use a combination of several techniques. Some state can be transfered in the URL. What can't is either

thrown into the `HttpSession` or 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`. The HttpSession is great place for true session data, data that is common to all requests that the user has with the application. However, it's a bad place to store data related to individual series of requests. Using the session of conversational quickly breaks down when dealing with the back button and multiple windows. On top of that, without careful programming, data in the HTTP Session can grow quite large, making the HTTP session difficult to cluster. 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. Fortunately, with Seam, you don't have to worry about that.

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.

In this application, we'll use the conversation context to store stateful session beans. There is an ancient canard in the Java community that stateful session beans are a scalability killer. This may have been true in the early days of enterprise Java, but it is no longer true today. Modern application servers have extremely sophisticated mechanisms for stateful session bean state replication. JBoss AS, for example, 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. If you remain unconvinced, Seam allows the use of POJOs instead of stateful session beans. With Seam, the choice is yours.

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 RichFaces Ajax 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.

Exemple 1.25. HotelSearchingAction.java

```
@Stateful                                                    (1)
@Name("hotelSearch")
@Scope(ScopeType.SESSION)

@Restrict("#{identity.loggedIn}")                            (2)
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 void find()
  {
    page = 0;
    queryHotels();
  }
  public void nextPage()
  {
    page++;
    queryHotels();
  }

  private void queryHotels()
  {
    hotels =
        em.createQuery("select h from Hotel h where lower(h.name) like #{pattern} " +
                  "or lower(h.city) like #{pattern} " +
                  "or lower(h.zip) like #{pattern} " +
```

```
                    "or lower(h.address) like #{pattern}")
            .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;
    }

    @Factory(value="pattern", scope=ScopeType.EVENT)
    public String getSearchPattern()
    {
      return searchString==null ?
            "%" : '%' + searchString.toLowerCase().replace('*', '%') + '%';
    }

    public String getSearchString()
    {
      return searchString;
    }

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

                                                    4

    @Remove
    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 `@Restrict` annotation applies a security restriction to the component. It restricts access to the component allowing only logged-in users. The security chapter explains more about security in Seam.
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 must define a method with no parameters marked `@Remove`. This method will be called when Seam destroys the session context.

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

Exemple 1.26. main.xhtml

```
<div class="section">

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

  <h1>Search Hotels</h1>

  <h:form id="searchCriteria">
  <fieldset>
    <h:inputText id="searchString" value="#{hotelSearch.searchString}"
                 style="width: 165px;">
      <a:support event="onkeyup" actionListener="#{hotelSearch.find}"
                 reRender="searchResults" />                          (1)
    </h:inputText>
     
    <a:commandButton id="findHotels" value="Find Hotels" action="#{hotelSearch.find}"
                     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 id="hotels" 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 id="viewHotel" value="View Hotel"                     4
              action="#{hotelBooking.selectHotel(hot)}"/>
      </h:column>
    </h:dataTable>
    <s:link value="More results" action="#{hotelSearch.nextPage}"
          rendered="#{hotelSearch.nextPageAvailable}"/>
  </div>
</a:outputPanel>
```

1. The RichFaces Ajax `<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 RichFaces Ajax `<a:status>` tag lets us display an animated image while we wait for asynchronous requests to return.
3. The RichFaces Ajax `<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.

   If you're wondering how navigation occurs, you can find all the rules in `WEB-INF/pages.xml`; this is discussed in *Section 6.7, « La navigation »*.

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 let's 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.

Exemple 1.27. HotelBookingAction.java

```
@Stateful
@Name("hotelBooking")
@Restrict("#{identity.loggedIn}")
public class HotelBookingAction implements HotelBooking
{


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

  @In
  private User user;

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

```
@In(required=false)

@Out(required=false)                                    2
private Booking booking;

@In
private FacesMessages facesMessages;

@In
private Events events;

@Logger
private Log log;

private boolean bookingValid;


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

public void 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() );
}

public void setBookingDetails()
{
  Calendar calendar = Calendar.getInstance();
  calendar.add(Calendar.DAY_OF_MONTH, -1);
  if ( booking.getCheckinDate().before( calendar.getTime() ) )
  {
    facesMessages.addToControl("checkinDate", "Check in date must be a future date");
    bookingValid=false;
  }
  else if ( !booking.getCheckinDate().before( booking.getCheckoutDate() ) )
  {
    facesMessages.addToControl("checkoutDate",
```

```
                        "Check out date must be later than check in date");
      bookingValid=false;
    }
    else
    {
      bookingValid=true;
    }
  }

  public boolean isBookingValid()
  {
    return bookingValid;
  }


  @End                                                    4
  public void confirm()
  {
    em.persist(booking);
    facesMessages.add("Thank you, #{user.name}, your confimation number " +
                      " for #{hotel.name} is #{booki g.id}");
    log.info("New booking: #{booking.id} for #{user.username}");
    events.raiseTransactionSuccessEvent("bookingConfirmed");
  }

  @End
  public void cancel() {}


  @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 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 Debug Page

The WAR also includes `seam-debug.jar`. The Seam debug page will be available if this jar is deployed in `WEB-INF/lib`, along with the Facelets, and if you set the debug property of the `init` component:

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

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`* [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

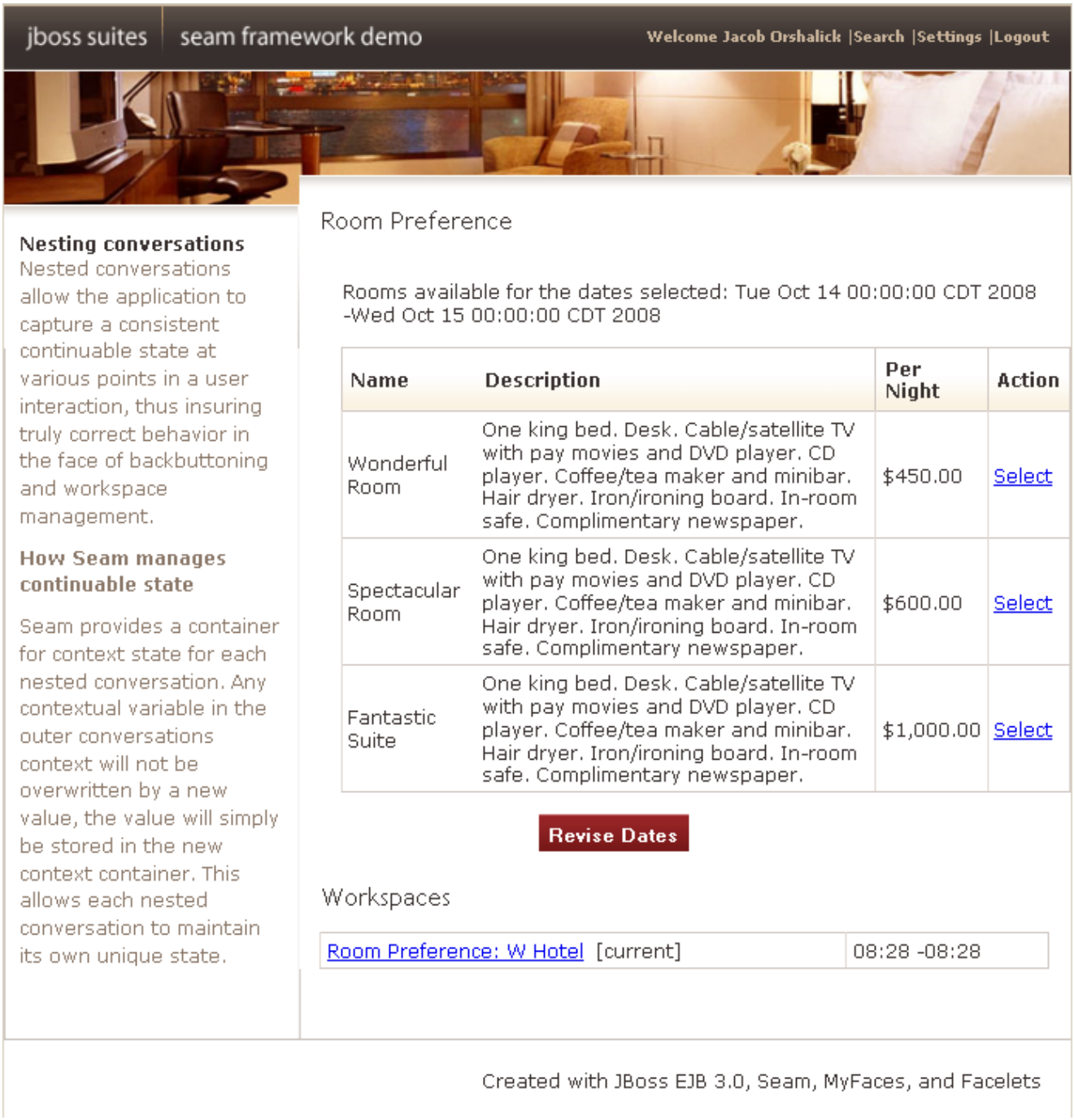
+ Application Context

1.7. Nested conversations: extending the Hotel Booking example

1.7.1. Introduction

Long-running conversations make it simple to maintain consistency of state in an application even in the face of multi-window operation and back-buttoning. Unfortunately, simply beginning and ending a long-running conversation is not always enough. Depending on the requirements of the application, inconsistencies between what the user's expectations and the reality of the application's state can still result.

The nested booking application extends the features of the hotel booking application to incorporate the selection of rooms. Each hotel has available rooms with descriptions for a user to select from. This requires the addition of a room selection page in the hotel reservation flow.



The user now has the option to select any available room to be included in the booking. As with the hotel booking application we saw previously, this can lead to issues with state consistency. As with storing state in the `HTTPSession`, if a conversation variable changes it affects all windows operating within the same conversation context.

To demonstrate this, let's suppose the user clones the room selection screen in a new window. The user then selects the *Wonderful Room* and proceeds to the confirmation screen. To see just how much it would cost to live the high-life, the user returns to the original window, selects the *Fantastic Suite* for booking, and again proceeds to confirmation. After reviewing the total cost, the user decides that practicality wins out and returns to the window showing *Wonderful Room* to confirm.

In this scenario, if we simply store all state in the conversation, we are not protected from multi-window operation within the same conversation. Nested conversations allow us to achieve correct behavior even when context can vary within the same conversation.

1.7.2. Understanding Nested Conversations

Now let's see how the nested booking example extends the behavior of the hotel booking application through use of nested conversations. Again, we can read the class from top to bottom, as if it were a story.

Exemple 1.28. RoomPreferenceAction.java

```
@Stateful
@Name("roomPreference")
@Restrict("#{identity.loggedIn}")
public class RoomPreferenceAction implements RoomPreference
{

   @Logger
   private Log log;

   @In private Hotel hotel;

   @In private Booking booking;

   @DataModel(value="availableRooms")
   private List<Room> availableRooms;

   @DataModelSelection(value="availableRooms")
   private Room roomSelection;

   @In(required=false, value="roomSelection")
   @Out(required=false, value="roomSelection")
   private Room room;

   @Factory("availableRooms")

   public void loadAvailableRooms()
```

```
{
                availableRooms = hotel.getAvailableRooms(booking.getCheckinDate(),
booking.getCheckoutDate());
  log.info("Retrieved #0 available rooms", availableRooms.size());
}

public BigDecimal getExpectedPrice()
{
  log.info("Retrieving price for room #0", roomSelection.getName());

  return booking.getTotal(roomSelection);
}


@Begin(nested=true)                                                    2
public String selectPreference()
{
  log.info("Room selected");


  this.room = this.roomSelection;                                      3

  return "payment";
}

public String requestConfirmation()
{
  // all validations are performed through the s:validateAll, so checks are already
  // performed
  log.info("Request confirmation from user");

  return "confirm";
}

@End(beforeRedirect=true)
                                                                       4
public String cancel()
{
  log.info("ending conversation");

  return "cancel";
}

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

```
}
```

1. The `hotel` instance is injected from the conversation context. The hotel is loaded through an *extended persistence context* so that the entity remains managed throughout the conversation. This allows us to lazily load the `availableRooms` through an `@Factory` method by simply walking the association.
2. When `@Begin(nested=true)` is encountered, a nested conversation is pushed onto the conversation stack. When executing within a nested conversation, components still have access to all outer conversation state, but setting any values in the nested conversation's state container does not affect the outer conversation. In addition, nested conversations can exist concurrently stacked on the same outer conversation, allowing independent state for each.
3. The `roomSelection` is outjected to the conversation based on the `@DataModelSelection`. Note that because the nested conversation has an independent context, the `roomSelection` is only set into the new nested conversation. Should the user select a different preference in another window or tab a new nested conversation would be started.
4. The `@End` annotation pops the conversation stack and resumes the outer conversation. The `roomSelection` is destroyed along with the conversation context.

When we begin a nested conversation it is pushed onto the conversation stack. In the `nestedbooking` example, the conversation stack consists of the outer long-running conversation (the booking) and each of the nested conversations (room selections).

Exemple 1.29. rooms.xhtml

```
<div class="section">
   <h1>Room Preference</h1>
</div>

<div class="section">
   <h:form id="room_selections_form">
      <div class="section">
         <h:outputText styleClass="output"
            value="No rooms available for the dates selected: "
            rendered="#{availableRooms != null and availableRooms.rowCount == 0}"/>
         <h:outputText styleClass="output"
            value="Rooms available for the dates selected: "
            rendered="#{availableRooms != null and availableRooms.rowCount > 0}"/>

         <h:outputText styleClass="output" value="#{booking.checkinDate}"/> -
         <h:outputText styleClass="output" value="#{booking.checkoutDate}"/>

         <br/><br/>
```

```
                                                    (1)
        <h:dataTable value="#{availableRooms}" var="room"
                rendered="#{availableRooms.rowCount > 0}">
          <h:column>
            <f:facet name="header">Name</f:facet>
            #{room.name}
          </h:column>
          <h:column>
            <f:facet name="header">Description</f:facet>
            #{room.description}
          </h:column>
          <h:column>
            <f:facet name="header">Per Night</f:facet>
            <h:outputText value="#{room.price}">
              <f:convertNumber type="currency" currencySymbol="$"/>
            </h:outputText>
          </h:column>

          <h:column>                                (2)
            <f:facet name="header">Action</f:facet>
            <h:commandLink id="selectRoomPreference"
              action="#{roomPreference.selectPreference}">Select</h:commandLink>
          </h:column>
        </h:dataTable>
      </div>
      <div class="entry">
        <div class="label"> </div>

        <div class="input">                         (3)
          <s:button id="cancel" value="Revise Dates" view="/book.xhtml"/>
        </div>
      </div>
    </h:form>
</div>
```

(1) When requested from EL, the `#{availableRooms}` are loaded by the `@Factory` method defined in `RoomPreferenceAction`. The `@Factory` method will only be executed once to load the values into the current context as a *`@DataModel`* instance.

(2) Invoking the `#{roomPreference.selectPreference}` action results in the row being selected and set into the `@DataModelSelection`. This value is then outjected to the nested conversation context.

(3) Revising the dates simply returns to the `/book.xhtml`. Note that we have not yet nested a conversation (no room preference has been selected), so the current conversation can simply

be resumed. The `<s:button >` component simply propagates the current conversation when displaying the `/book.xhtml` view.

Now that we have seen how to nest a conversation, let's see how we can confirm the booking once a room has been selected. This can be achieved by simply extending the behavior of the `HotelBookingAction`.

Exemple 1.30. HotelBookingAction.java

```
@Stateful
@Name("hotelBooking")
@Restrict("#{identity.loggedIn}")
public class HotelBookingAction implements HotelBooking
{

  @PersistenceContext(type=EXTENDED)
  private EntityManager em;

  @In
  private User user;

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

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

  @In(required=false)
  private Room roomSelection;

  @In
  private FacesMessages facesMessages;

  @In
  private Events events;

  @Logger
  private Log log;

  @Begin
  public void selectHotel(Hotel selectedHotel)
  {
    log.info("Selected hotel #0", selectedHotel.getName());
```

```
    hotel = em.merge(selectedHotel);
  }

  public String setBookingDates()
  {
    // the result will indicate whether or not to begin the nested conversation
    // as well as the navigation.  if a null result is returned, the nested
    // conversation will not begin, and the user will be returned to the current
    // page to fix validation issues
    String result = null;

    Calendar calendar = Calendar.getInstance();
    calendar.add(Calendar.DAY_OF_MONTH, -1);

    // validate what we have received from the user so far
    if ( booking.getCheckinDate().before( calendar.getTime() ) )
    {
      facesMessages.addToControl("checkinDate", "Check in date must be a future date");
    }
    else if ( !booking.getCheckinDate().before( booking.getCheckoutDate() ) )
    {
       facesMessages.addToControl("checkoutDate", "Check out date must be later than check
in date");
    }
    else
    {
      result = "rooms";
    }

    return result;
  }

  public void 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() );
  }

  @End(root=true)
                                                                    (1)
  public void confirm()
  {
```

```
    // on confirmation we set the room preference in the booking.  the room preference
    // will be injected based on the nested conversation we are in.
    booking.setRoomPreference(roomSelection);
                                                                    (2)
    em.persist(booking);
     facesMessages.add("Thank you, #{user.name}, your confimation number for #{hotel.name}
 is #{booking.id}");
    log.info("New booking: #{booking.id} for #{user.username}");
    events.raiseTransactionSuccessEvent("bookingConfirmed");
  }


  @End(root=true, beforeRedirect=true)                              (3)
  public void cancel() {}

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

(1) Annotating an action with `@End(root=true)` ends the root conversation which effectively destroys the entire conversation stack. When any conversation is ended, its nested conversations are ended as well. As the root is the conversation that started it all, this is a simple way to destroy and release all state associated with a workspace once the booking is confirmed.

(2) The `roomSelection` is only associated with the `booking` on user confirmation. While outjecting values to the nested conversation context will not impact the outer conversation, any objects injected from the outer conversation are injected by reference. This means that any changing to these objects will be reflected in the parent conversation as well as other concurrent nested conversations.

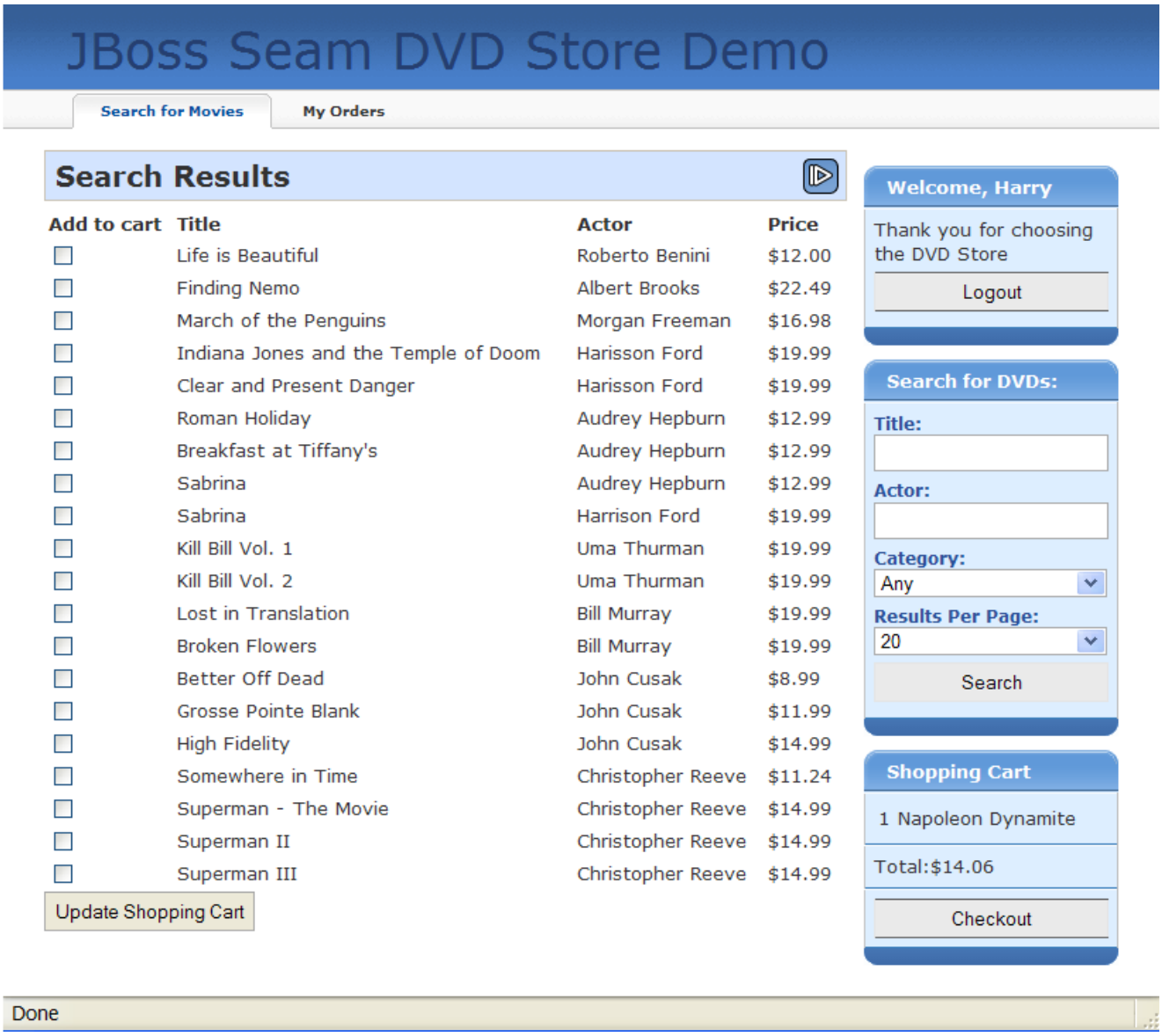
(3) By simply annotating the cancellation action with `@End(root=true, beforeRedirect=true)` we can easily destroy and release all state associated with the workspace prior to redirecting the user back to the hotel selection view.

Feel free to deploy the application, open many windows or tabs and attempt combinations of various hotels with various room preferences. Confirming a booking always results in the correct hotel and room preference thanks to the nested conversation model.

1.8. 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.

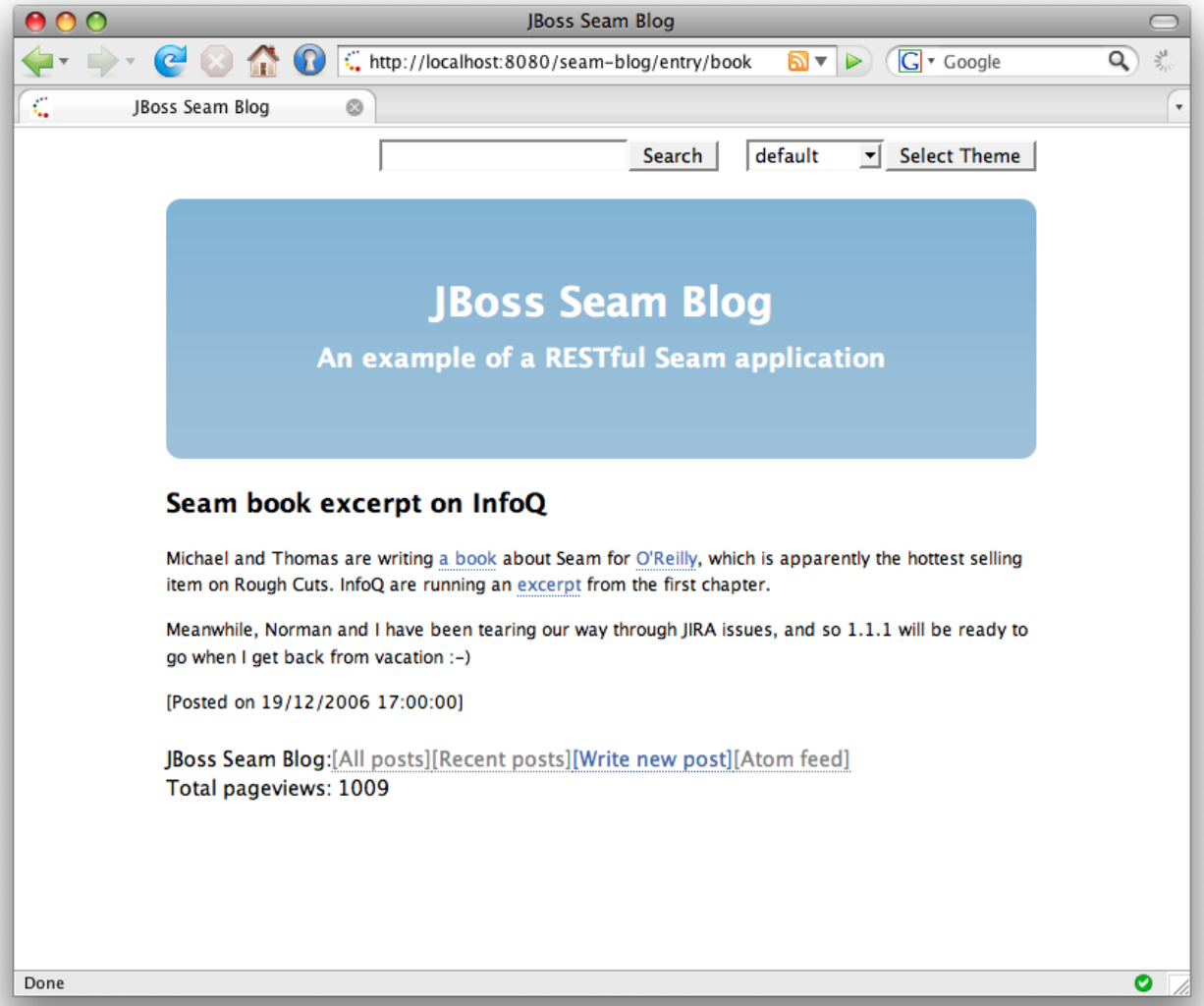


The administration screens take use jBPM to manage the approval and shipping cycle for orders. The business process may even be changed dynamically, by selecting a different process definition!

The Seam DVD Store demo can be run from `dvdstore` directory, just like the other demo applications.

1.9. Bookmarkable URLs with 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 want to keep application state in the URL so that any page can be accessed at any time through a bookmark. The blog example shows how to a implement an application that supports bookmarking throughout, even on the search results page. This example demonstrates how Seam can manage application state in the URL as well as how Seam can rewrite those URLs to be even



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.

1.9.1. Using "pull"-style MVC

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

Exemple 1.31.

```
<h:dataTable value="#{blog.recentBlogEntries}" var="blogEntry" rows="3">
 <h:column>
   <div class="blogEntry">
     <h3>#{blogEntry.title}</h3>
     <div>
```

```
      <s:formattedText value="#{blogEntry.excerpt==null ? blogEntry.body : blogEntry.excerpt}"/>
    </div>
    <p>
      <s:link view="/entry.xhtml" rendered="#{blogEntry.excerpt!=null}" propagation="none"
          value="Read more...">
        <f:param name="blogEntryId" value="#{blogEntry.id}"/>
      </s:link>
    </p>
    <p>
      [Posted on 
      <h:outputText value="#{blogEntry.date}">
        <f:convertDateTime timeZone="#{blog.timeZone}" locale="#{blog.locale}" type="both"/>
      </h:outputText>]
       
      <s:link view="/entry.xhtml" propagation="none" value="[Link]">
        <f:param name="blogEntryId" value="#{blogEntry.id}"/>
      </s:link>
    </p>
  </div>
 </h:column>
</h:dataTable>
```

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

Exemple 1.32.

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

  @In EntityManager entityManager;                                    (1)

  @Unwrap                                                             (2)
  public Blog getBlog()
  {
      return (Blog) entityManager.createQuery("select distinct b 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?

1.9.2. 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`:

Exemple 1.33.

```
<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:

```
<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:

```
<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 in the URL to get a bookmarkable URL like `http://localhost:8080/seam-blog/search/`. JSF does not provide an easy way to do this, but Seam does. We use two Seam features to accomplish this: *page parameters* and *URL rewriting*. Both are defined in `WEB-INF/pages.xml`:

Exemple 1.34.

```
<pages>
  <page view-id="/search.xhtml">
    <rewrite pattern="/search/{searchPattern}"/>
    <rewrite pattern="/search"/>

    <param name="searchPattern" value="#{searchService.searchPattern}"/>

  </page>
  ...
</pages>
```

The page parameter instructs Seam to link the request parameter named `searchPattern` to the value of `#{searchService.searchPattern}`, both whenever a request for the Search page comes in and whenever a link to the search page is generated. Seam takes responsibility for maintaining the link between URL state and application state, and you, the developer, don't have to worry about it.

Without URL rewriting, the URL for a search on the term `book` would be `http://localhost:8080/seam-blog/seam/search.xhtml?searchPattern=book`. This is nice, but Seam can make the URL even simpler using a rewrite rule. The first rewrite rule, for the pattern `/search/{searchPattern}`, says that any time we have a URL for search.xhtml with a searchPattern request parameter, we can fold that URL into the simpler URL. So,the URL we saw earlier, `http://localhost:8080/seam-blog/seam/search.xhtml?searchPattern=book` can be written instead as `http://localhost:8080/seam-blog/search/book`.

Just like with page parameters, URL rewriting is bi-directional. That means that Seam forwards requests for the simpler URL to the the right view, and it also automatically generates the simpler view for you. You never need to worry about constructing URLs. It's all handled transparently behind the scenes. The only requirement is that to use URL rewriting, the rewrite filter needs to be enabled in `components.xml`.

```
<web:rewrite-filter view-mapping="/seam/*" />
```

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

```
<h:dataTable value="#{searchResults}" var="blogEntry">
  <h:column>
    <div>
      <s:link view="/entry.xhtml" propagation="none" value="#{blogEntry.title}">
        <f:param name="blogEntryId" value="#{blogEntry.id}"/>
      </s:link>
      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 using Hibernate Search.

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

   @In
   private FullTextEntityManager entityManager;

   private String searchPattern;

   @Factory("searchResults")
   public List<BlogEntry> getSearchResults()
   {
      if (searchPattern==null || "".equals(searchPattern) ) {
         searchPattern = null;
```

```
            return entityManager.createQuery("select be from BlogEntry be order by date
desc").getResultList();
      }
      else
      {
        Map<String,Float> boostPerField = new HashMap<String,Float>();
        boostPerField.put( "title", 4f );
        boostPerField.put( "body", 1f );
        String[] productFields = {"title", "body"};
        QueryParser parser = new MultiFieldQueryParser(productFields, new StandardAnalyzer(),
boostPerField);
        parser.setAllowLeadingWildcard(true);
        org.apache.lucene.search.Query luceneQuery;
        try
        {
          luceneQuery = parser.parse(searchPattern);
        }
        catch (ParseException e)
        {
          return null;
        }

        return entityManager.createFullTextQuery(luceneQuery, BlogEntry.class)
            .setMaxResults(100)
            .getResultList();
      }
    }

    public String getSearchPattern()
    {
      return searchPattern;
    }

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

}
```

1.9.3. 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:

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

  @Out 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`:

```
<pages>
  ...

  <page view-id="/entry.xhtml">
    <rewrite pattern="/entry/{blogEntryId}" />
    <rewrite pattern="/entry" />

    <param name="blogEntryId"
         value="#{blogEntry.id}"/>

    <action execute="#{entryAction.loadBlogEntry(blogEntry.id)}"/>
  </page>

  <page view-id="/post.xhtml" login-required="true">
    <rewrite pattern="/post" />

    <action execute="#{postAction.post}"
          if="#{validation.succeeded}"/>
```

```
      <action execute="#{postAction.invalid}"
            if="#{validation.failed}"/>

      <navigation from-action="#{postAction.post}">
         <redirect view-id="/index.xhtml"/>
      </navigation>
   </page>

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

</pages>
```

Notice that the example is using page actions for post validation 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. Keep in mind that because of the URL rewriting, the blogEntryId parameter name won't show up in the URL. Seam 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:

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

```
@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:

```
@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.

The blog demo also demonstrates very simple password authentication, posting to the blog, page fragment caching and atom feed generation.

Démarrage avec Seam en utilisant seam-gen

La distribution Seam inclus un utilitaire en ligne de commande qui permet facilement de configurer un projet Eclipse, de générer un squelette de code simple Seam et de réaliser une ingénierie inverse sur une base de données pré existante.

C'est une façon simple de garder les pieds au sec avec Seam, et de vous donnez des munitions pour la prochaine fois que vous vous trouverez enfermer dans un ascenseur avec un de ces gars tordus de Ruby vous taquinant sur le fait que leur jouet est génial et merveilleux pour construire des applications triviales qui mettent des choses dans les base de données.

Dans cette version, seam-gen fonctionne mieux pour tous avec JBoss AS. Vous pouvez utiliser le projet généré avec d'autre serveurs d'application J2EE ou Java EE5. en réalisant plusieurs modifications à la main à la configuration du projet.

Vous *pouvez* utiliser seam-gen sans Eclipse, mais dans ce tutoriel, nous voulons vous montrer comment l'utiliser en conjonction avec Eclipse pour le débogage ou l'intégration des test. Si vous ne voulez pas installer Eclipse vous pouvez quand même suivre ce tutoriel—toutes les étapes peuvent être réalisées depuis la ligne de commande.

Seam-gen est simplement un gros script Ant bien déguelasse utilisant des outils Hibernate, liant ensemble des patrons. Ce qui rends facile la personnalisation cela vos besoins.

2.1. Avant de démarrer

Soyez sur d'avoir JDK5 ou JDK6 (voir *Section 42.1, « Les dépendances du JDK »* pour les détails), JBoss AS 4.2 ou 5.0 et Ant 1.7.0, avec des versions récentes d'Eclipse, de Jboss IDE plugin pour Eclipse et de TestNG plugin pour Eclipse correctement installés avant le démarrage. Ajoutez votre installation JBoss a la vue Server JBoss dans Eclipse. Démarrer JBoss en mode debug. Enfin, démarrez un interpréteur de commande dans le dossier où vous avez dézipper la distribution de Seam.

Jboss dispose d'un support sophistiquer pour le redéploiement à chaud des WARs et des EARs. Malheureusement, à cause de bugs dans la JVM, le redéploiement répété de EAR—ce qui est habituel dans la phase de développement—peut éventuellement entrainer la JVM a être à cours d'espace perm gen. Pour cette raison, nous recommandons d'exécuter JBoss dans une JVM avec un large espace perm gen pendant la période de développement. Si vous exécuter Jboss depuis JBoss IDE, vous pouvez configurer cela dans la configuration de lancement du serveur, dans "VM arguments". Nous vous conseillons les valeurs suivantes :

```
-Xms512m -Xmx1024m -XX:PermSize=256m -XX:MaxPermSize=512m
```

Si vous ne disposez pas d'assez de mémoire, ce qui suit est notre recommandation minimale :

```
-Xms256m -Xmx512m -XX:PermSize=128m -XX:MaxPermSize=256m
```

vous exécuter JBoss depuis la ligne de commande, vous pouvez configurer les options de la JVM dans `bin/run.conf`.

Si vous ne voulez pas vous inquiéter de ce truc pour l'instant, nous n'êtes pas obligé, revenez y plus tard quand vous rencontrer votre premier `OutOfMemoryException`..

2.2. Configurer un nouveau projet Eclipse

La première chose que vous avez besoin de faire est de configurer seam-gen pour votre environnement : le dossier d'installation de JBoss AS, l'espace de travail d'Eclipse et connexion à la base de données. C'est simple, entrez juste :

```
cd jboss-seam-2.2.x
seam setup
```

Et vous allez être interrogé pour un complément d'information :

```
~/workspace/jboss-seam$ ./seam setup
Buildfile: build.xml

init:

setup:
     [echo] Welcome to seam-gen :-)
     [input] Enter your project workspace (the directory that contains your Seam projects) [C:/
Projects] [C:/Projects]
/Users/pmuir/workspace
    [input] Enter your JBoss home directory [C:/Program Files/jboss-4.2.3.GA] [C:/Program Files/
jboss-4.2.3.GA]
/Applications/jboss-4.2.3.GA
    [input] Enter the project name [myproject] [myproject]
helloworld
     [echo] Accepted project name as: helloworld
    [input] Select a RichFaces skin (not applicable if using ICEFaces) [blueSky] ([blueSky], classic,
 ruby, wine, deepMarine, emeraldTown, sakura, DEFAULT)

     [input] Is this project deployed as an EAR (with EJB components) or a WAR (with no EJB
 support) [ear]  ([ear], war, )
```

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

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

    [input] What kind of database are you using? [hsql]  ([hsql], mysql, oracle, postgres, mssql,
 db2, sybase, enterprisedb, h2)
mysql
     [input] Enter the Hibernate dialect for your database [org.hibernate.dialect.MySQLDialect]
 [org.hibernate.dialect.MySQLDialect]

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

   [input] Enter the JDBC URL for your database [jdbc:mysql:///test] [jdbc:mysql:///test]
jdbc:mysql:///helloworld
   [input] Enter database username [sa] [sa]
pmuir
   [input] Enter database password [] []

   [input] skipping input as property hibernate.default_schema.new has already been set.
   [input] Enter the database catalog name (it is OK to leave this blank) [] []

   [input] Are you working with tables that already exist in the database? [n]  (y, [n], )
y
   [input] Do you want to drop and recreate the database tables and data in import.sql each time
 you deploy? [n]  (y, [n], )
n
   [input] Enter your ICEfaces home directory (leave blank to omit ICEfaces) [] []

[propertyfile] Creating new property file: /Users/pmuir/workspace/jboss-seam/seam-gen/
build.properties
     [echo] Installing JDBC driver jar to JBoss server
     [echo] Type 'seam create-project' to create the new project

BUILD SUCCESSFUL
Total time: 1 minute 32 seconds
```

```
~/workspace/jboss-seam $
```

Cet outil fourni des valeurs par défaut judicieuses que vous pouvez accepter juste en appuyant sur la couche enter à la demande.

Le choix le plus important que vous devez faire est entre un déploiement EAR et un déploiement WAR pour votre projet. Les projets EAR supportent EJB 3.0 et nécessite Java EE5. Les projets WAR ne supportent pas EJB 3.0 mais peuvent être déployés dans un environnement J2EE. Le conditionnement en WAR est aussi plus simple à comprendre. Si vous avez installé un server d'application opérationnel EJB3 comme JBoss, choisissez `ear`. Sinon, choississez `war`. Nous allons partir du principe que vous avez choisi un déploiement EAR pour le reste du tutoriel, mais vous pouvez suivre exactement les même étapes pour un déploiement WAR.

Si vous travaillez avec un modèle de données déjà existant, soyez sur d'indiquer à seam-gen que des tables existent déjà dans la base de données.

Les régleages sont stockés dans `seam-gen/build.properties`, mais vous pouvez aussi les modifier simplement en relançant `seam setup` une seconde fois.

Maintenant vous pouvez créer un nouveau projet dans le dossier de votre espace de travail d'Eclipse, en entrant :

```
seam new-project
```

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

...

new-project:
     [echo] A new Seam project named 'helloworld' was created in the C:\Projects directory
     [echo] Type 'seam explode' and go to http://localhost:8080/helloworld
    [echo] Eclipse Users: Add the project into Eclipse using File > New > Project and select General
 > Project (not Java Project)
     [echo] NetBeans Users: Open the project in NetBeans

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

Cela recopie les jars de Seam, les jars nécéssaires et le jar du pilote JDBC dans un nouveau projet Eclipse et génère tout les fichiers de configuration et les ressources requises, le fichier modèle facelets et la feuille de style, ainsi que les metadata Eclipse et le script de construction

Ant. Le projet Eclipse va être automatiquement déployé dans une structure de dossiers transférer dans JBoss AS dès que vous ajouterez le projet en utilisant `New -> Project... -> General -> Project -> Next`, entrez le `Project name` (`helloworld` dans notre cas), et alors cliquez sur `Finish`. Ne sélectionnez pas `Java Project` dans l'assistant de Nouveau Projet.

Si votre JDK par défaut dans Eclipse n'est pas un SDK Java SE 5 ou Java SE 6, vous allez devoir sélectionner un JDK compatible Java SE 5 en faisant `Project -> Properties -> Java Compiler`.

utre alternative, vous pouvez déployer le projet hors d'Eclipse en entrant `seam explode`.

Allez sur `http://localhost:8080/helloworld` pour voir la page d'accueil. C'est une page facelets, `view/home.xhtml`, l, en utilisant le modèle `view/layout/template.xhtml`. Vous pouvez éditer cette page, ou le modèle dans Eclipse et voir le résultat *immediately*, en appuyant sur le bouton actualiser de votre navigateur.

Ne soyez pas effrayé par les documents de configuration XML qui ont été généré dans le dossier du projet. Ils sont pour la plus part des trucs standards Java EE, un truc créé une fois et plus jamais consulté ensuite et ceci est le cas dans 90% des projets Seam. (ils sont si facile à écrire que même seam-gen peut le faire.)

Le projet généré inclu trois bases de données et des configurations de persistance. Les fichiers `persistence-test.xml`, persistence-test.xml et `import-test.sql` sont utilisés quand vous exécutez les tests unitaires testNG avec HSQLDB. Le schéma de base de données et les données de test dans `import-test.sql` sont toujours exportés vers la base de données avant l'exécution des tests. Les fichiers `myproject-dev-ds.xml`, `persistence-dev.xml` et `import-dev.sql` ont utilisés pendant le déploiement de l'application vers la base de données de développement. Le schéma peut être exporté automatiquement pendant le déploiement, cela dépend si vous avez indiqué à seam-gen que vous travaillez avec une base de données déjà existante. Les fichiers `myproject-prod-ds.xml`, `persistence-prod.xml` et `import-prod.sql` sont utilisés pendant le déploiement de l'application vers la base de données de production. Le schema n'est pas exporté automatiquement pendant le déploiement.

2.3. Création d'une nouvelle action

Si vous avez l'habitude de serveur d'application web de style action traditionnel, vous vous demandez comment créer une simple page web avec un méthode d'action sans état en Java. Si vous entrez :

```
seam new-action
```

Seam va vous demander des informations, et génèrer une nouvelle page facelets ainsi qu'un composant Seam pour votre projet.

```
C:\Projects\jboss-seam>seam new-action
```

```
Buildfile: 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\helloworld\src\hot\org\jboss\helloworld
    [copy] Copying 1 file to C:\Projects\helloworld\src\hot\org\jboss\helloworld
    [copy] Copying 1 file to C:\Projects\helloworld\src\hot\org\jboss\helloworld\test
    [copy] Copying 1 file to C:\Projects\helloworld\src\hot\org\jboss\helloworld\test
    [copy] Copying 1 file to C:\Projects\helloworld\view
    [echo] Type 'seam restart' and go to http://localhost:8080/helloworld/ping.seam

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

Vue que nous avons ajouté un nouveau composant Seam, nous devons redémarrer le déploiement de la structure du dossier. Vous pouvez faire cela en entrant `seam restart`, en en execution la cible `restart` qui se trouve dans le fichier `build.xml` avec Eclipse. Une autre façon de forcer un redémarrage est d'éditer le fichier `resources/META-INF/application.xml` dans Eclipse. *Notez que vous navez pas à redémarrer JBoss à chaque vois que vous modifier votre application*.

Maintenant allez à `http://localhost:8080/helloworld/ping.seam` et cliquez sur le bouton. Vous pouvez voir le code ci-dessous en action en regardant dans le dossier `src`. Placez un point d'arrêt dans la méthode `ping()` et cliquez sur le bouton de nouveau.

Finalement, localisez le fichier `PingTest.xml` dans le package de test et lancez les tests d'intégration en utilisant le plugin testNG pour Eclipse. Autre alternative, lancez les tests en utilisant `seam test` ou via la cible du fichier généré `test`.

2.4. Création d'un formulaire avec une action

L'étape suivante est de créer un formulaire Entrez :

```
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\hot\com\hello
     [copy] Copying 1 file to C:\Projects\hello\src\hot\com\hello
     [copy] Copying 1 file to C:\Projects\hello\src\hot\com\hello\test
     [copy] Copying 1 file to C:\Projects\hello\view
     [copy] Copying 1 file to C:\Projects\hello\src\hot\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>
```

Relancer l'application encore et allez sur `http://localhost:8080/helloworld/hello.seam`. Ensuite, jetez un coup d'œil sur le code généré. Lancez le test. Essayez d'ajouter de nouveau champs dans le formulaire et dans le composant Seam (rappelez vous de redémarrer le deploiement chaque fois que vous modifiez le code Java).

2.5. Generation d'une application depuis une base de données existante

Manuellement créez plusieurs tables dans votre base de données. (Si vous devez basculer vers une autre base de données, relancez juste encore une fois `seam setup`.) Maintenant entrez :

```
seam generate-entities
```

Relancez le déploiement et allez sur `http://localhost:8080/helloworld`. Vous pouvez naviguer dans la base de données, éditer les objets existants et créez de nouveaux objets. Si vous allez jeter un coup d'œil au code généré, vous être probablement émerveillé par sa simplicité ! Seam a été prévu pour que le code d'accès aux données soit simple à écrire à la main, même pour les gens qui ne veulent pas tricher en utilisant seam-gen.

2.6. Génération d'une application depuis des entitées JPA/EJB3 existantes

Placez votre existant, les classes entitées valides dans `src/main`. Ensuite entrez

```
seam generate-ui
```

Redémarrez le déploiement, et allez sur `http://localhost:8080/helloworld`.

2.7. Déploiement d'une application avec un EAR

Au final, nous voulons être capable de déployer l'application en utilisant l'empaquetage standard de Java EE. En premier, vous avons besoin de retirer le dossier fabriqué en exécutant `seam unexplode`. Pour déployer le EAR, nous pouvons entrer `seam deploy` à l'invite de commande, ou exécuter la cible `deploy` du script généré de construction du projet. Vous pouvez le désinstaller en utilisant `seam undeploy` ou la cible `undeploy`.

Par défaut, l'application sera déployée avec le *dev profile*. Le EAR inclura les fichiers `persistence-dev.xml` et `import-dev.sql`, et le fichier `myproject-dev-ds.xml` sera déployé. Vous pouvez modifier le profil et utilisez le *prod profile*, en entrant

```
seam -Dprofile=prod deploy
```

Vous pouvez même définir de nouveaux profils de déploiement pour votre application. Ajoutez juste les fichier dénommés de manière appropriés à votre projet —par exemple, `persistence-staging.xml`, `import-staging.sql` et `myproject-staging-ds.xml`—et sélectionnez le nom du profil en utilisant `-Dprofile=staging`.

2.8. Seam et le déploiement incrémental à chaud

Quand vous déployez votre application Seam dans un dossier complet, vous allez avoir un peu d'indications pour le déploiement incrémental à chaud au moment du développement. Vous avez besoin d'activer le mode debug à la fois dans Seam et Facelets, en ajoutant cette ligne à `components.xml`:

```
<core:init debug="true"
>
```

Maintenant, les fichiers suivants devraient être redéployés sans nécéssiter un rédémarrage complet de l'application web:

- toutes les pages facelets
- et tout fichier `pages.xml`

Mais si nous voulons changer le code Java, nous continuons à avoir besoin de réaliser un redémarrage complet de l'application.(Dans JBoss, cela peut être accompli en changeant le descripteur de déploiement de plus haut niveau `application.xml` pour un déploiement d'un EAR ou `web.xml` pour un déploiement d'un WAR.)

Mais si vous voulez réellement un cycle rapide d'édition/compilation/test, Seam supporte un redéploiement incrémental des composants JavaBeans. Pour utiliser cette fonctionnalité, vous devez déployer les composants JavaBeans dans le dossier `WEB-INF/dev`, donc il vont être rechargé par un chargeur de classes spécialisé de Seam au lieu du chargeur de classes WAR ou EAR.

Vous devez être au courrant sur les limitations suivantes:

- les composants doivent être des composants JavaBeans, ils ne peuvent pas être des beans EJB3 (nous travaillons pour régler cette limitation)
- les entités ne peuvent pas être déployées à chaud
- les composants déployés via `components.xml` ne devraient pas être déployées à chaud

- les composants déployables à chaud ne seront pas visibles dans les classes déployées hors de `WEB-INF/dev`
- e mode de debug de Seam doit être activé et `jboss-seam-debug.jar` doit être dans `WEB-INF/lib`
- Vous devez avoir le filtre de Seam installé dans web.xml
- Vous pouvez avoir des erreurs si le systèm est placé en chargement et débogage est actif.

Si vous créé un projet WAR en utilisant seam-gen, le déploiement incrémental à chaud est disponible automatiquement pour les classes placées dans le dossier source `src/hot`. Mais, sean-gen ne permet le support du déploiement incrémental à chaud pour les projets EAR.

2.9. Utilisation de Seam avec JBoss 4.0

Seam 2 a été développé pour JavaServer Faces 1.2. Avec l'utilisation de JBoss AS, nous vous recommandons d'utiliser JBoss 4.2 ou JBoss 5.0 les deux sont liés avec l'implémentation de référence JSF 1.2. Malgré tout, il est toujours possible d'utiliser Seam 2 sur une plateforme JBoss 4.0. il y a deux étapes de bases pour faire cela : installer une version de JBoss 4.0 avec EJB3 activé et remplacer MyFaces par l'implémentation de référence JSF 1.2. Une fois que ces étapes sont faites, l'application Seam 2 peut être déployé sur JBoss 4.0.

2.9.1. Installation de JBoss 4.0

JBoss 4.0 ne propose pas une configuration par défaut compatible avec Seam. Pour exécuter Seam, vous devez installer JBoss 4.0.5 en utilisant l'installateur JEMS 1.2 avec le profil ejv3 sélectionné. Seam ne va pas s'exécuter avec une installation qui n'inclu pas le support EJB3. L'installateur JEMS peut être téléchargé depuis *http://www.jboss.org/jbossas/downloads*.

2.9.2. Installation de JSF 1.2 RI

La configuration de JBoss 4.0 peut être trouvée dans `server/default/deploy/jbossweb-tomcat55.sar`. Vous allez avoir besoin d'effacer `myfaces-api.jar` et `myfaces-impl.jar` du dossier `jsf-libs`. Ensuite, vous allez devoir copier `jsf-api.jar`, `jsf-impl.jar`, `el-api.jar`, et `el-ri.jar` dans ce même dossier. Les JARs de JSF peuvent être trouvés dans le dossier `lib` de Seam. Les JARs EL peuvent être obtenue depuis la version 1.2 de Seam.

Vous allez aussi devoir éditer le fichier `conf/web.xml`, en remplaçant`myfaces-impl.jar` par `jsf-impl.jar`.

Démarrer avec Seam en utilisant JBoss Tools

JBoss Toolsest une collection de plugins Eclipse. JBoss Tools est un projet de d'assistant de création de Seam, Content Assist pour Unified Expression Language (EL) à la fois en facelets et en code Java, un editeur graphique pour le jPDL, un editeur graphique pour les fichiers de configuration de Seam, le support pour l'exécution des tests d'intégration de Seam depuis Eclipse, et bien plus.

Pour faire simple, si vous ête un utilisateur d'Eclipse, alors vous allez vouloirs JBoss Tools!

JBoss Tools, avec seam-gen, fonctionne mieux avec le JBoss AS, mais si c'est pas possible avec quelques trucs d'avoir votre application qui s'exécute sur les autres seveurs d'applications. Le changement est surtout pour ce qui est décrit plus loin pour seam-gen dans ce manueld e référence.

3.1. Avant de commencer

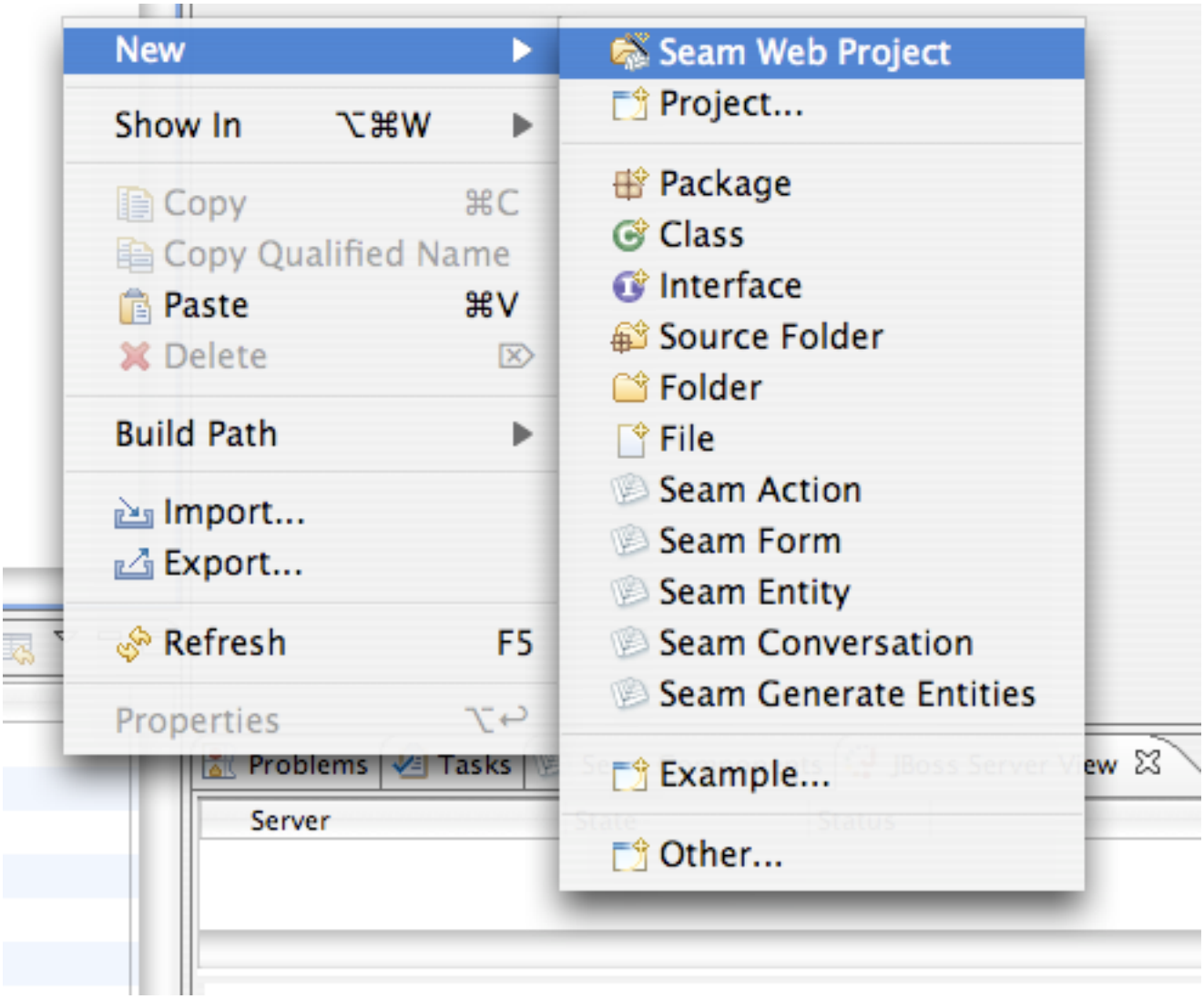
Vérifiez que vous avez JDK 5, JBoss AS 4.2 ou 5.0, Eclipse 3.3, le JBoss Tools plugins (à minima Seam Tools, le Visual Page Editor, jBPM Tools et JBoss AS Tools) et le plugin TestNG pour Eclipse correctement installé avant de commencer.

Merci de consulter la page officielle *JBoss Tools installation* [http://www.jboss.org/tools/download/installation] pour la plus rapide façon d'avoir l'installation de JBoss Tools dans Eclipse. Vous pouvez aussi consulter la page *Installing JBoss Tools* [http://www.jboss.org/community/wiki/InstallingJBossTools] sur le wiki de la communauté JBoss pour les détails les plus extrèmes et une suite d'approches alternatives.

3.2. Configuration d'un nouveau projet Seam

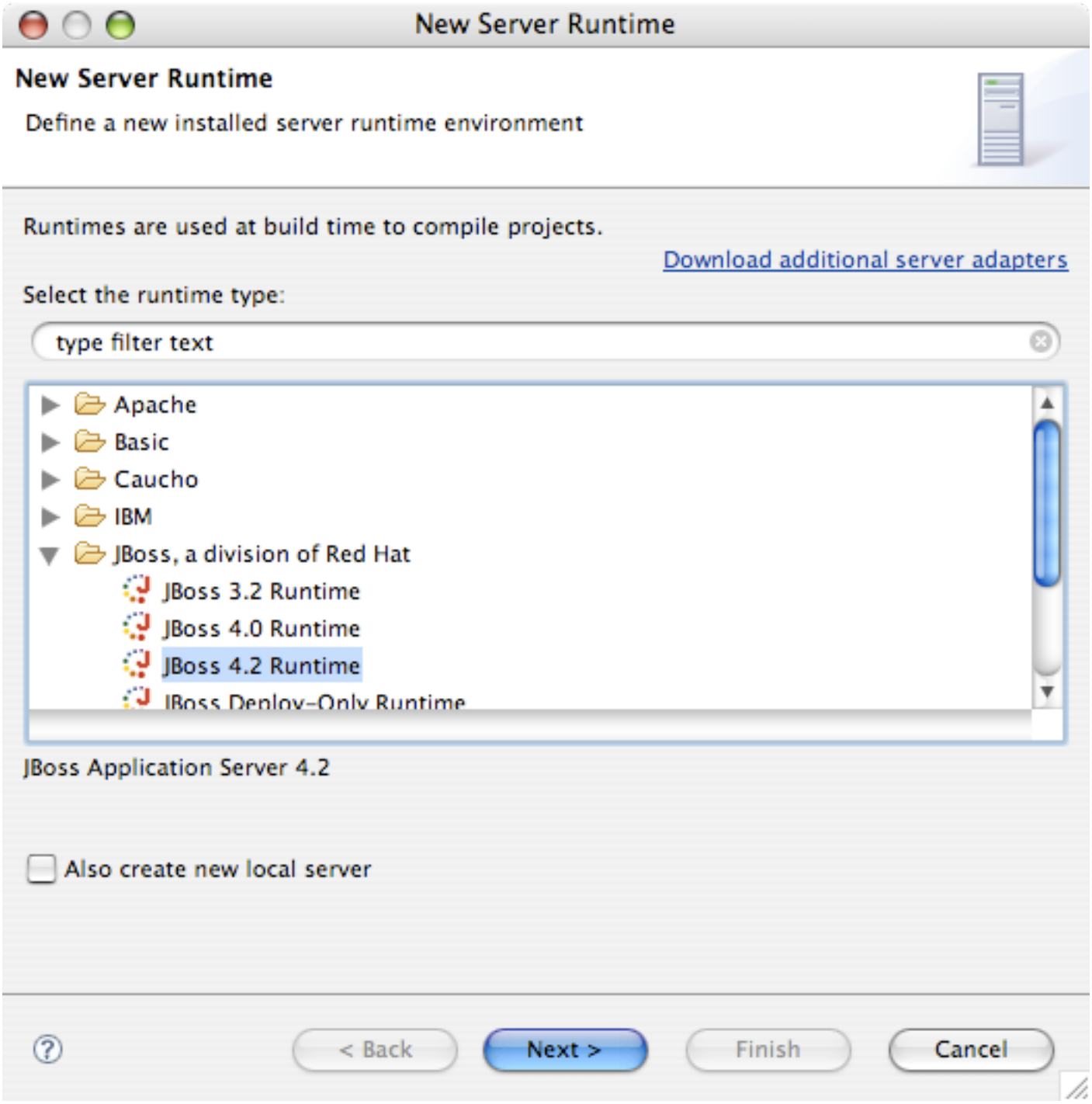
Démarrez Eclipse et sélectionnez la perspective *Seam*.

Allez dans *File -> New -> Seam Web Project*.

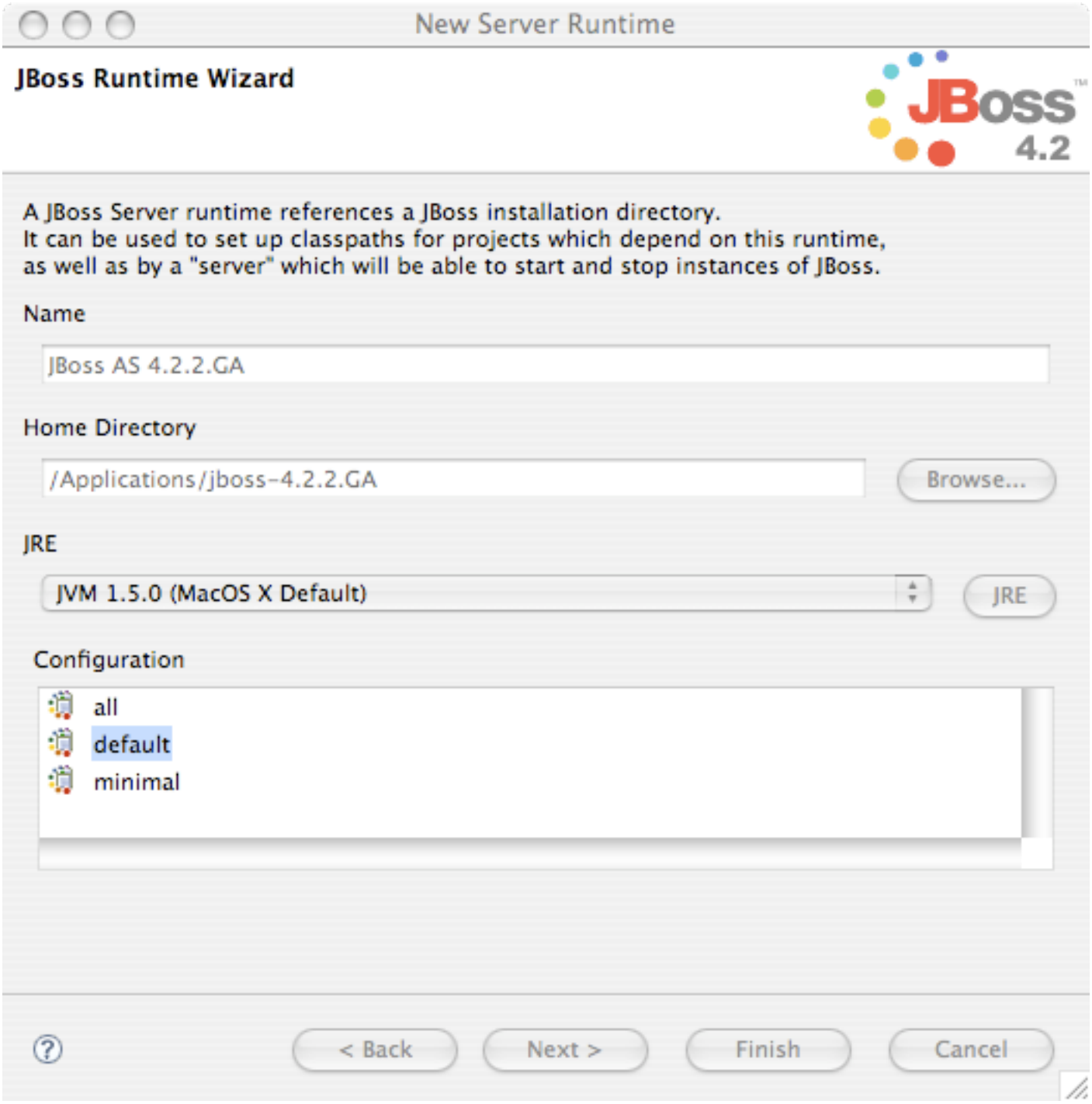


En premier lieu, entrez un nom pour votre nouveau projet. Pour ce tutoriel, nous allons utiliser`helloworld` .

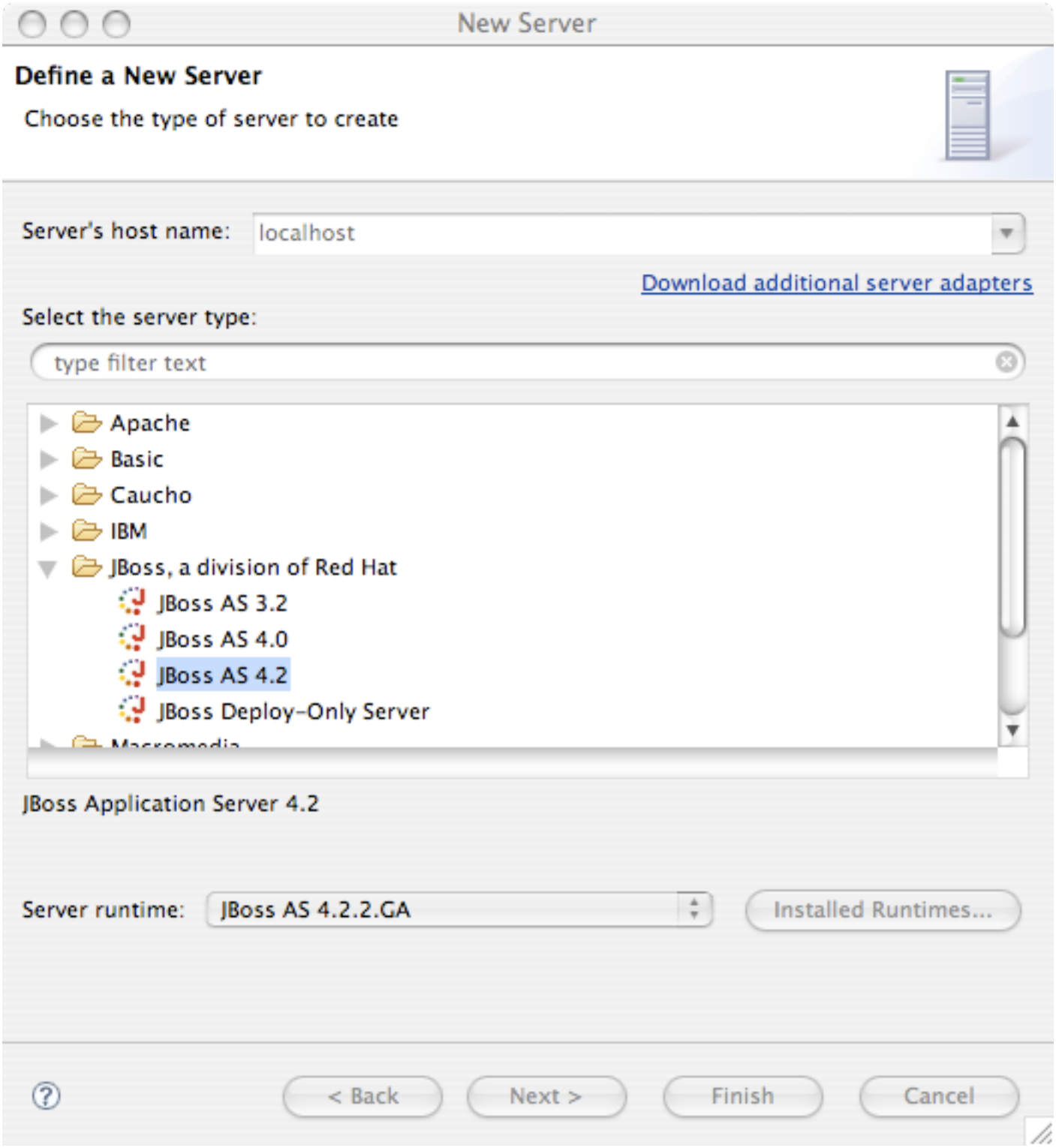
Maintenant, vous devez indiquer à JBoss Tools où est JBoss AS. Dans cet exemple, nous utilisons JBoss AS, pensez que vous pouvez certainement utiliser JBoss AS 5.0 tout aussi bien. La sélection de JBoss AS est un processus en deux étapes. En premier vous avez besoin de le définir à l'exécution. Ensuite, vous allez choisir JBoss AS, dasn ce cas:



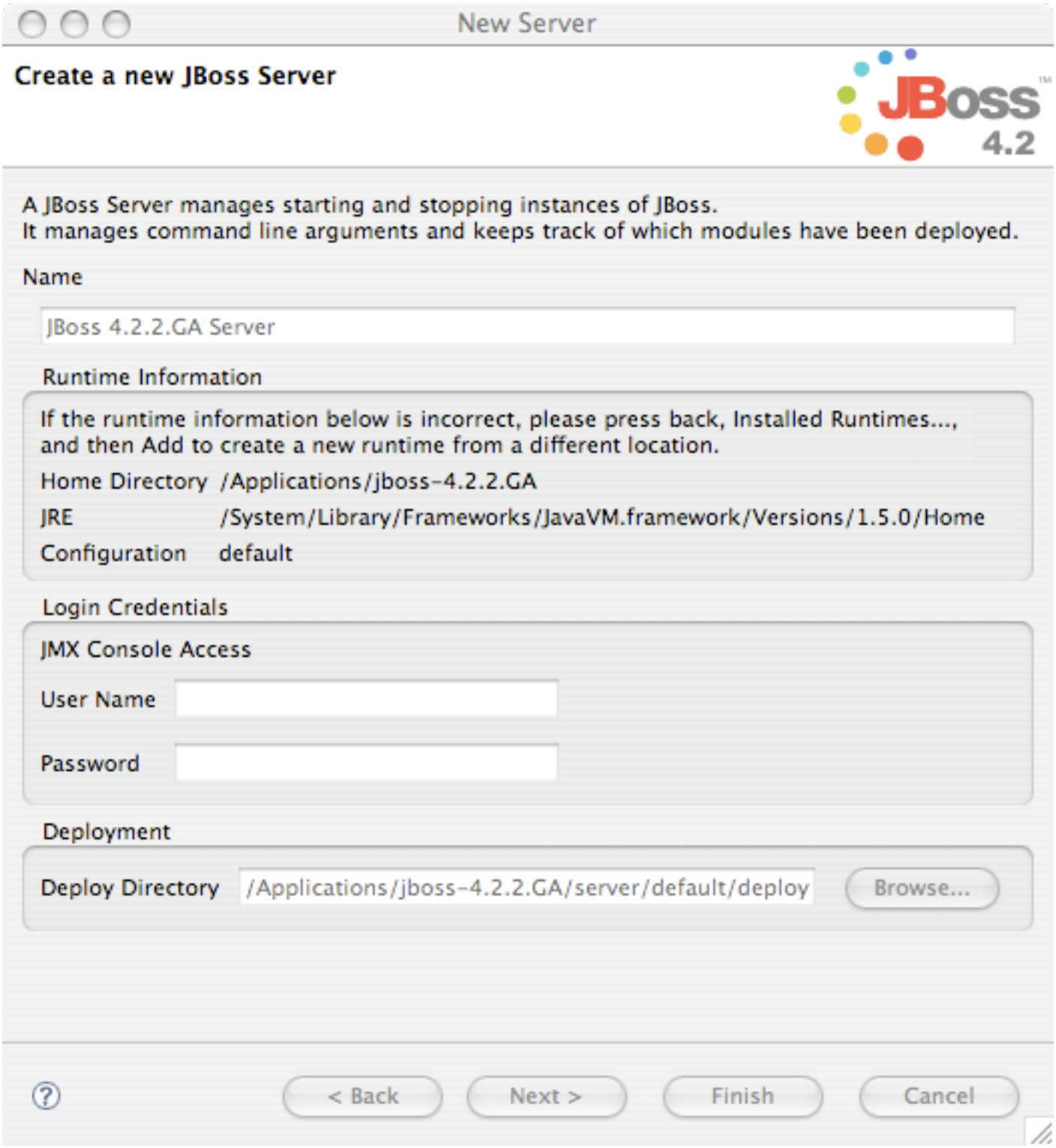
Entrez le nom à l'exécution, et localisez le sur votre disque dur:



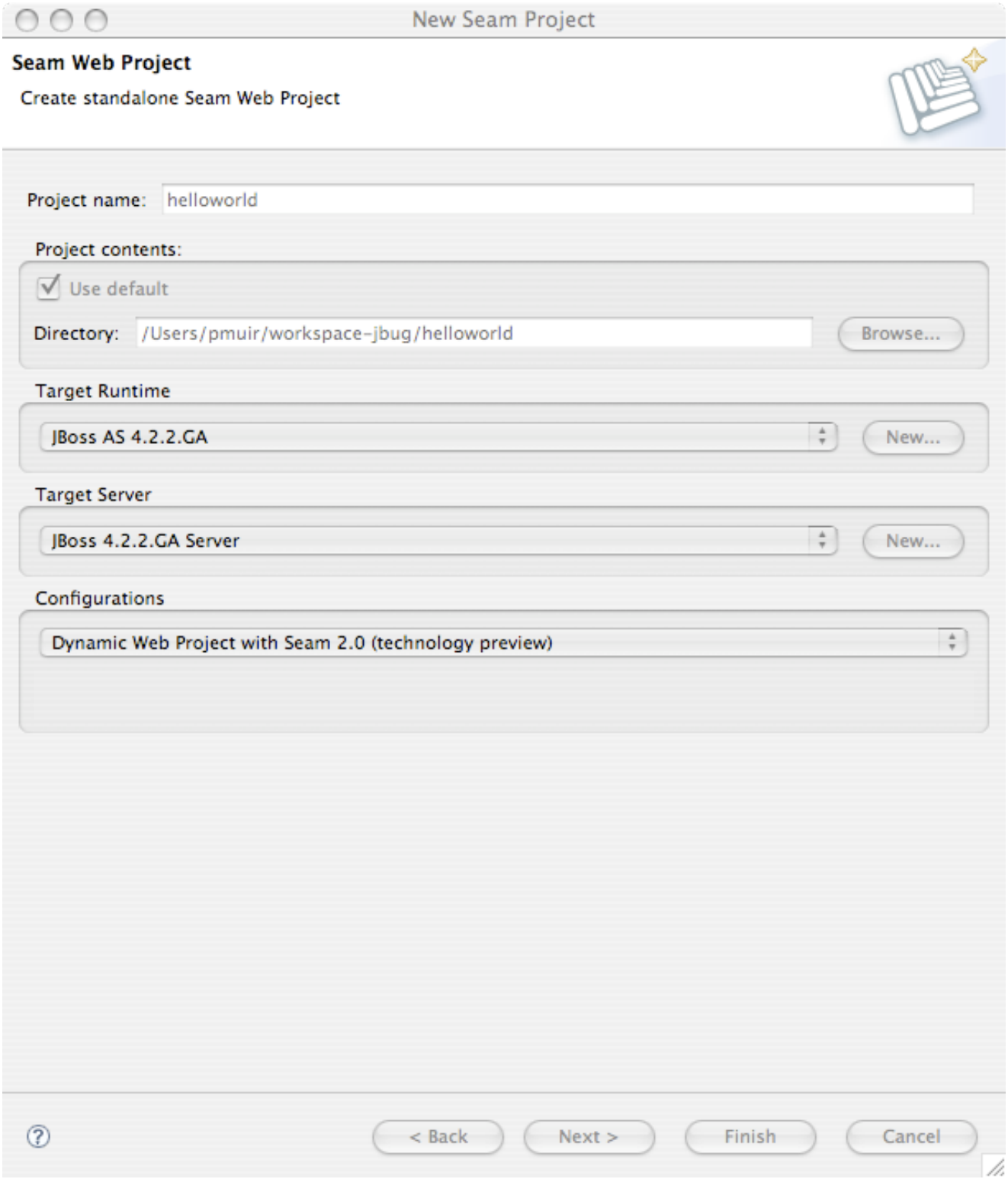
Ensuite, vous devez définir un serveur pour que JBoss Tools puisse y déployer le projet. Soyez sur de sélectionner encore une fois JBoss AS et aussi à l'exéuction vous devez juste définir:



Sur l'écran suivant donnez au serveur un nom, et appuyez sur *Finish*:

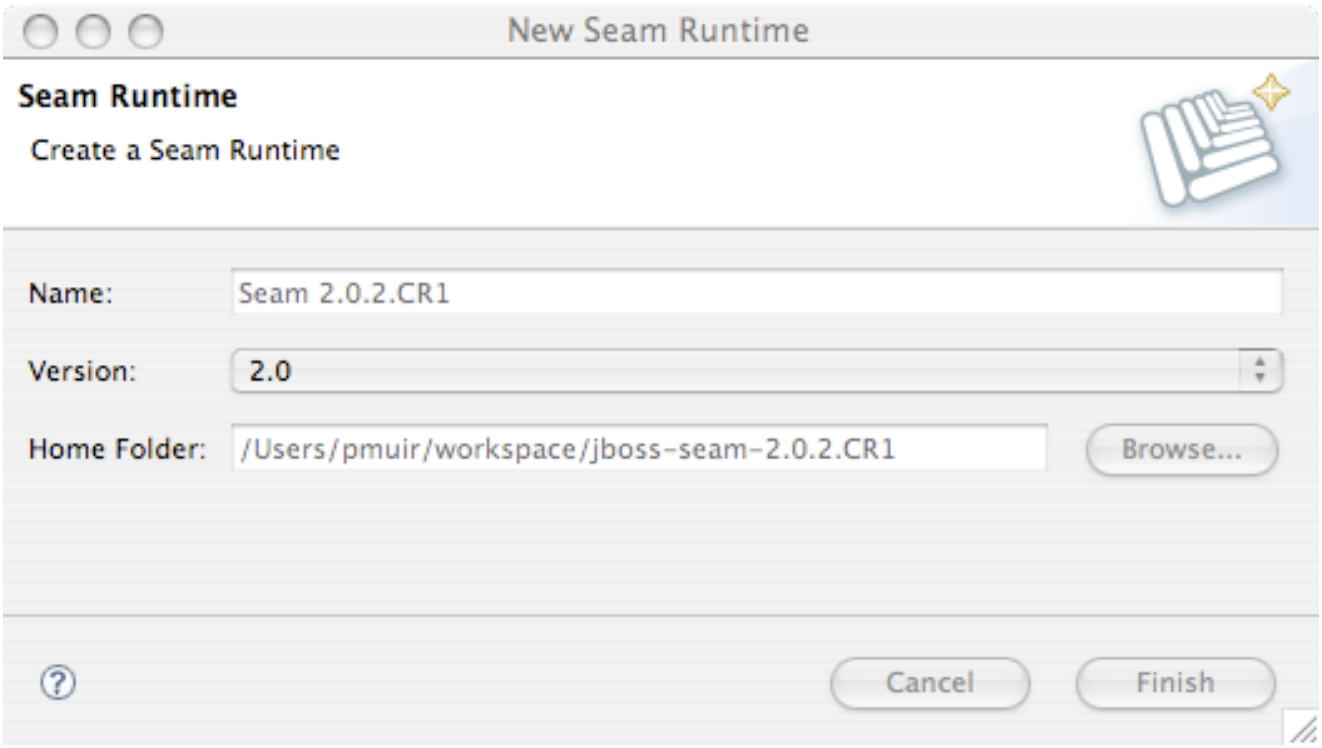


Soyez sur qu'à l'exécution et que le server que vous avez juste créez sont sélectionné, sélectionnez *Dynamic Web Project with Seam 2.0 (technology preview)* et appuyez sur *Next*:



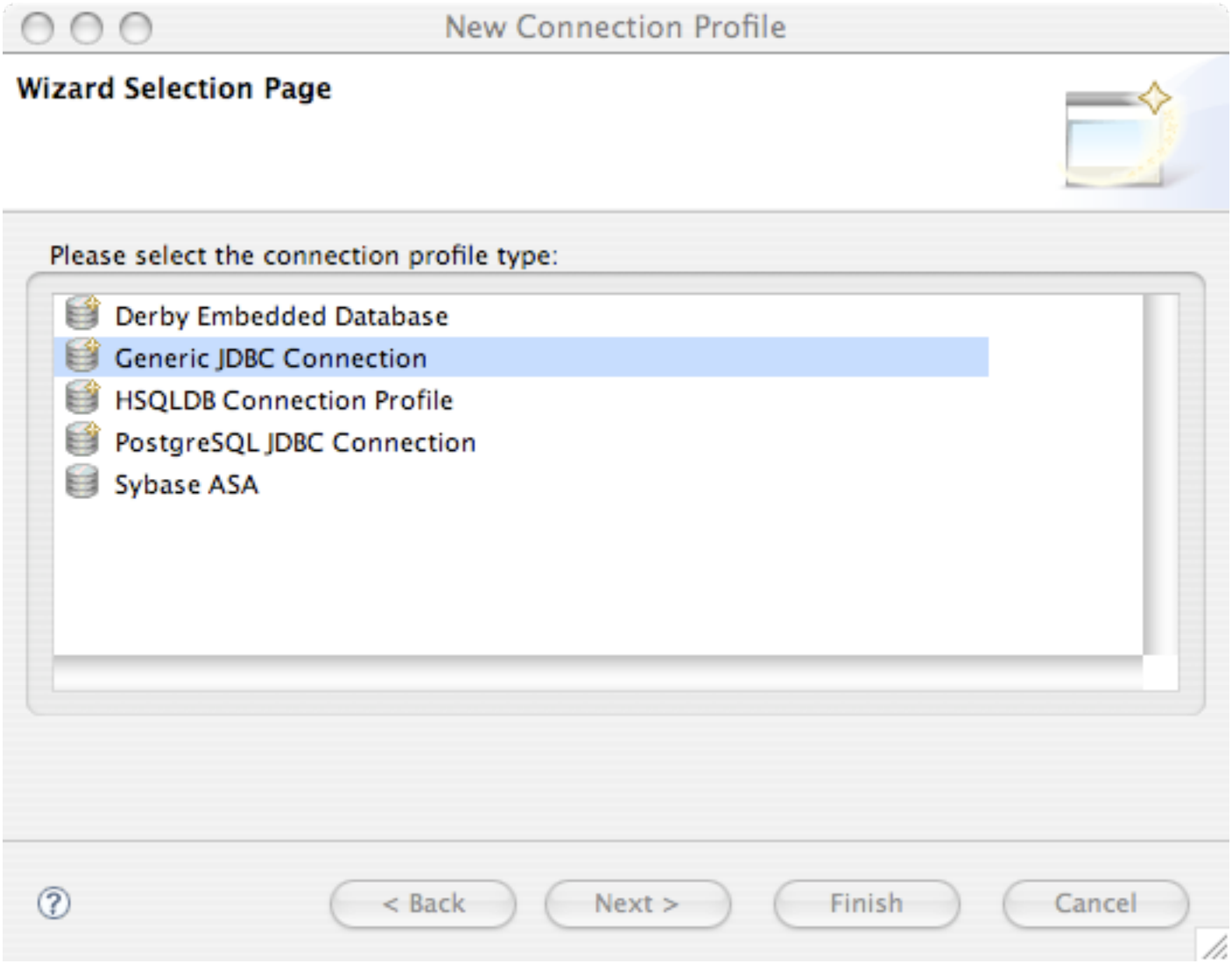
Les 3 écrans suivants vous permettent de personnalisé votre nouveau projet, mais pour nous les réglages par défaut seront très bien. Donc appuyez juste sur *Next* tant que vous n'avez pas atteind l'écran final.

La première étape ici est de dire à JBoss Tools où se trouve le Seam téléchargé que vous voulez utiliser. *Add* un nouveau *Seam Runtime* - soyez sur d'indiquer le nom , et sélectionez *2.0* comme version:

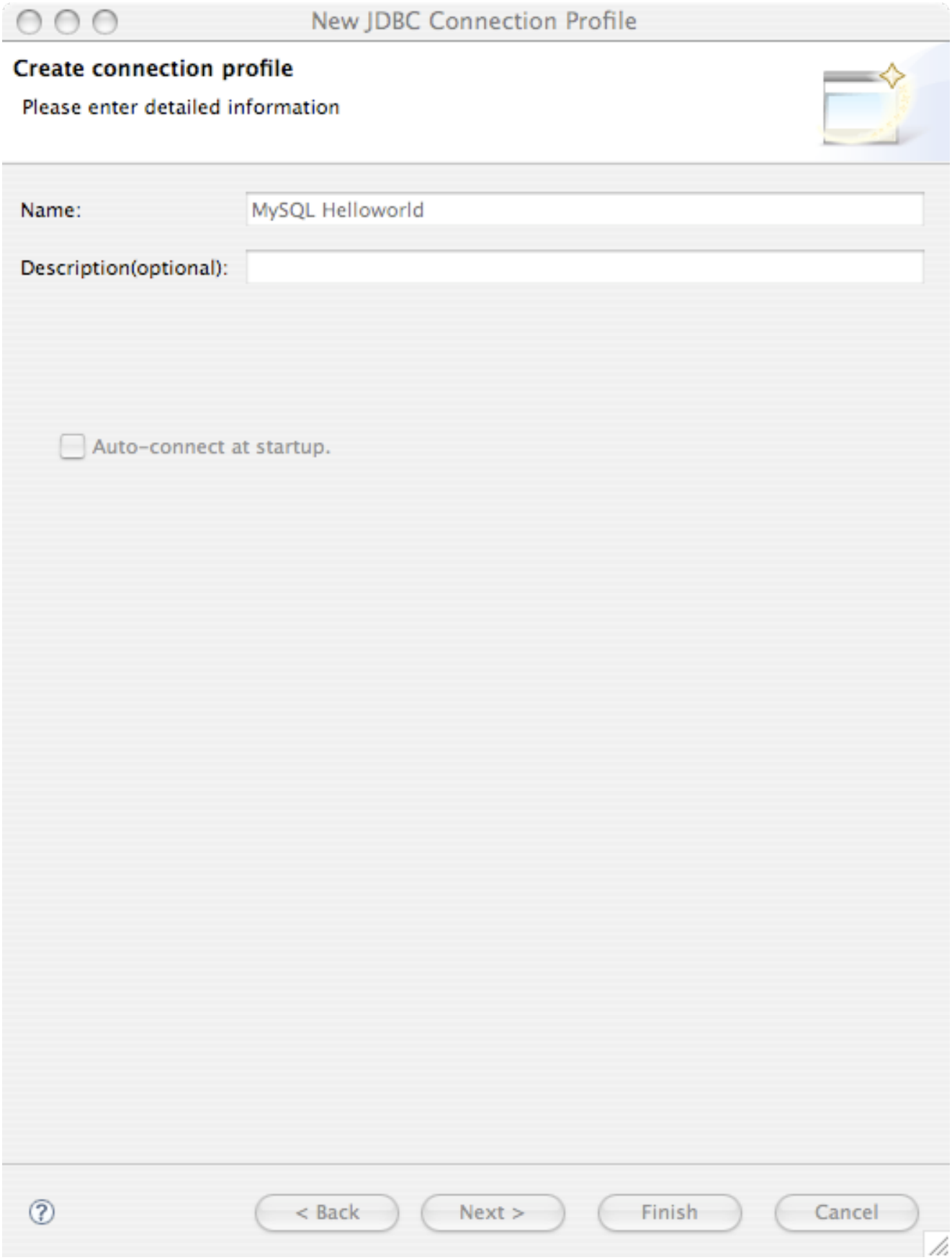


La choix le plus important que vous devez faire est entre un déploiement EAR ou un déploiement WAR de votre projet. Les projets EAR supportent les EJB 3.0 et nécéssitent Java EE5. Les projets WAR ne supportent pas les EJB3.0 mais peuvent être déployés dans un environement J2EE. L'empaquetage d'un WAR est aussi plus simple à comprendre. Si vous installez un serveur d'application prêt pour EJB3, choissisez *EAR*. Sinon choississez *WAR*. Nous allons supposez que vous avez choisi un déploiement WAR pour le reste du tutoriel, mais vous pouvez suivre exactement les mêmes étapes pour un déploiement de EAR.

Ensuite, sélectionnez votre type de base de données. Nous allons supposé que vous installé MySQL avec un schéma de base de données existant Vous allez devir indiquer à JBoss Tools où est la base de données, selectionnez *MySQL* comme base de données, et créez un nouveau profil de connection. Sélectionnez *Generic JDBC Connection*:

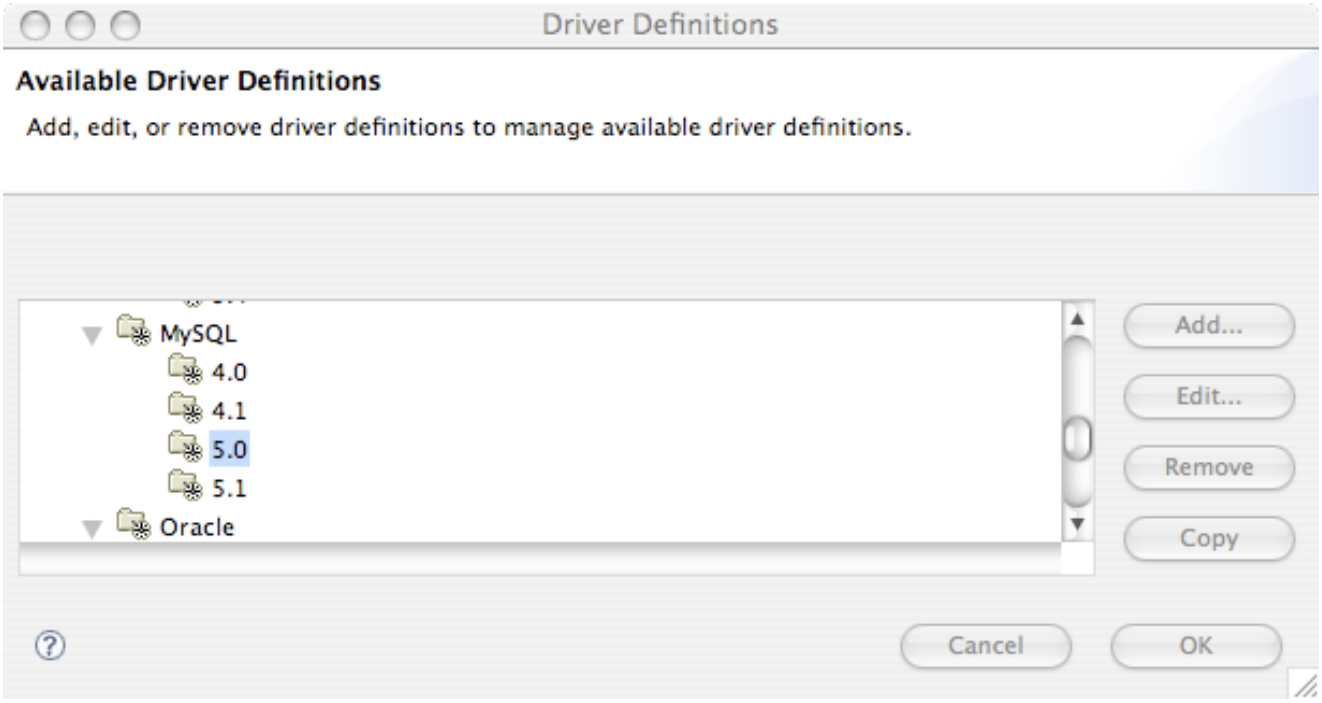


Indiquez lui un nom:

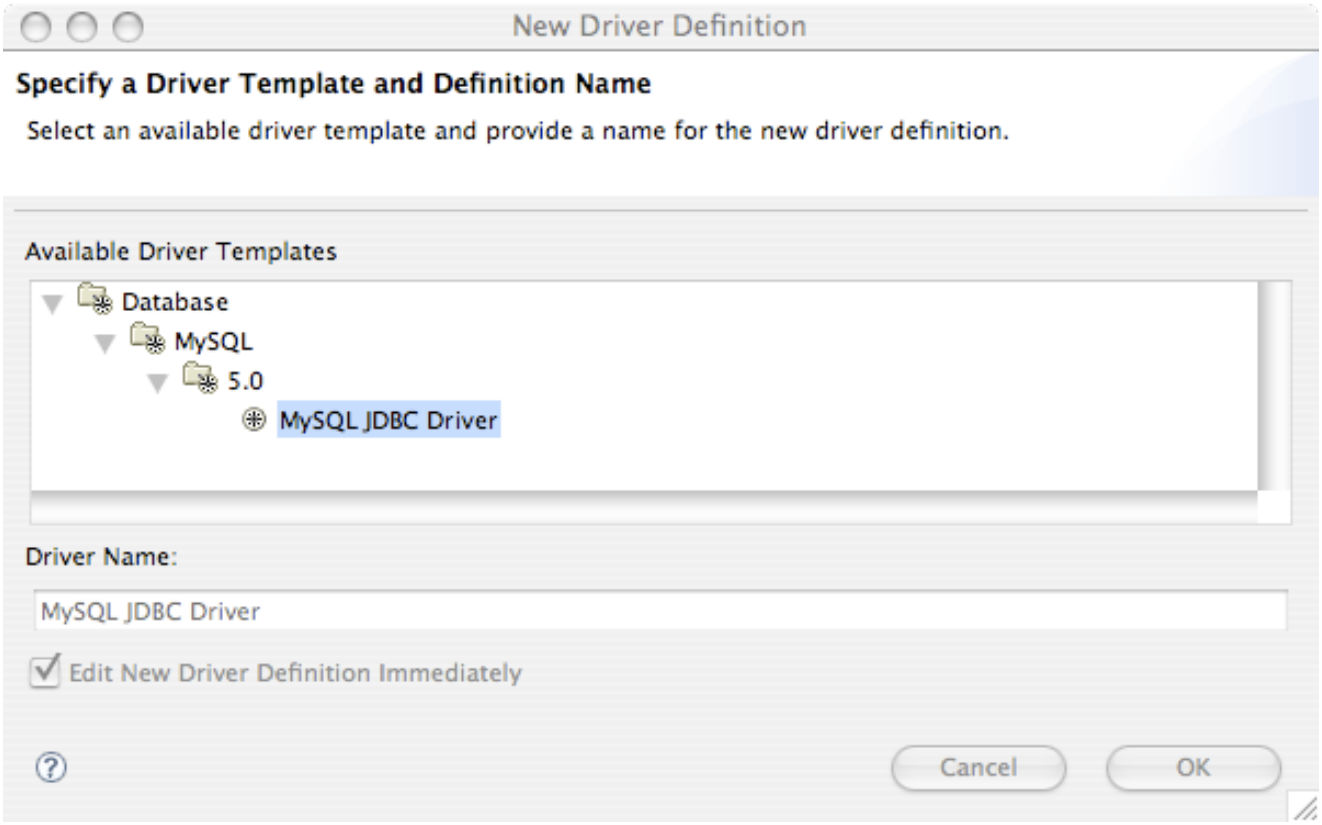

New JDBC Connection Profile
Create connection profile
Please enter detailed information
Name:
MySQL Helloworld
Description(optional):
Auto-connect at startup.
< Back
Next >
Finish
Cancel

JBoss Tolls n'est pas fourni avec des pilotes pour les bases de données, vous devez dire à JBoss Tools où se trouve le pilote MySQL JDBC. Indiquez lui où se trouve le pilote en cliquant sur ....

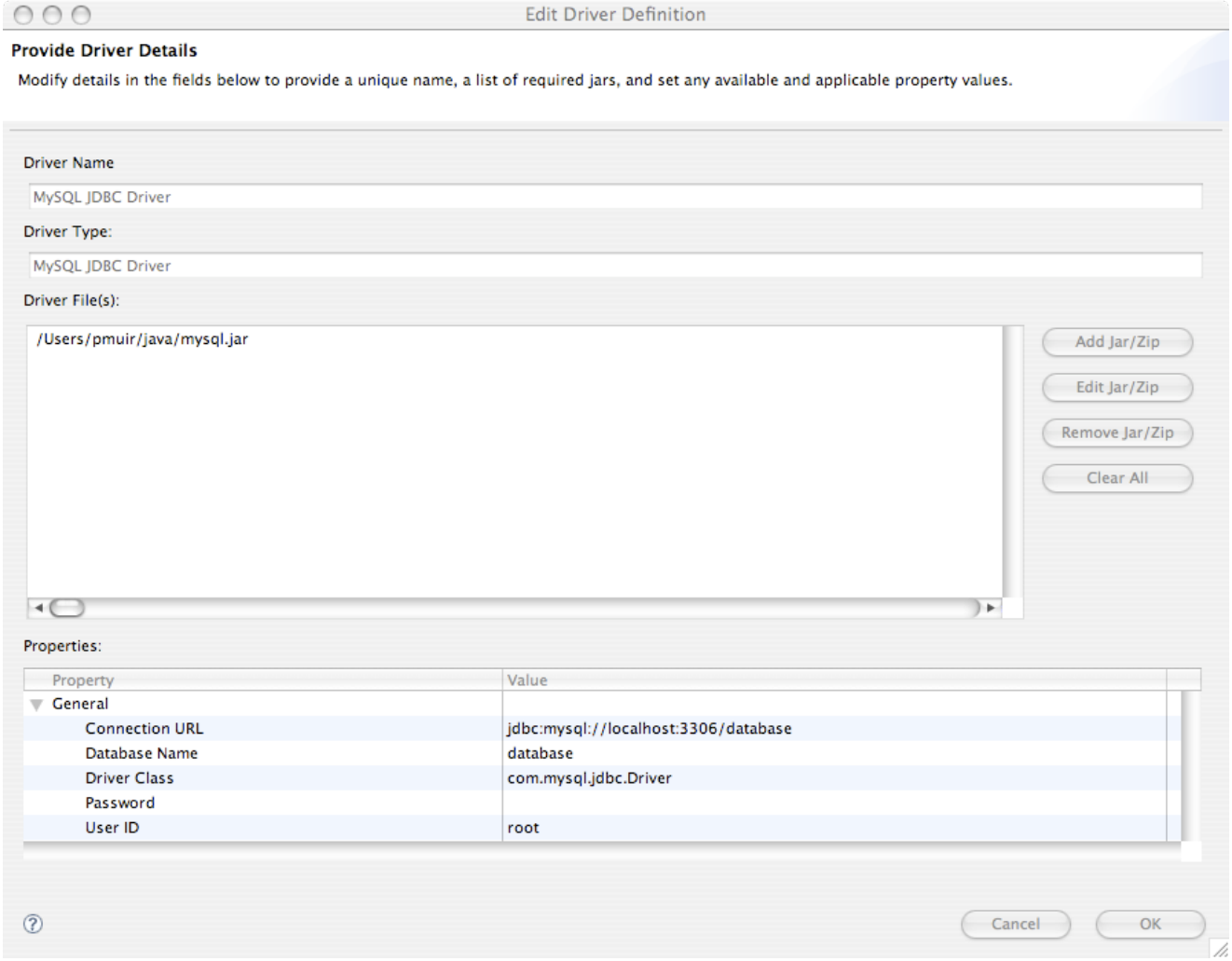
Localisez le MySQL 5, et appuyez sur*Add...*:



Choisissez le modèle de *MySQL JDBC Driver*:

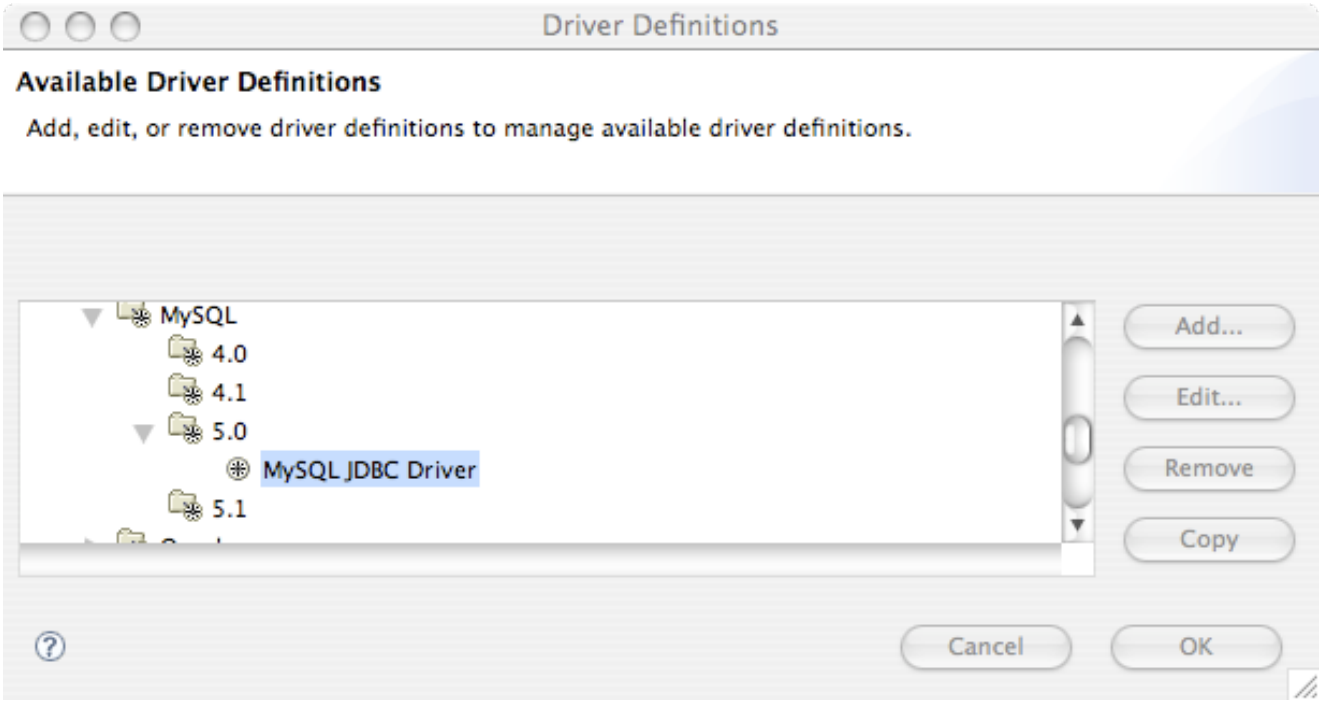


Localisez le jar sur votre ordinateur en choisissant *Edit Jar/Zip*:



Indiquez votre nom d'utilisateur et votre mode passe à utiliser pour se connecter, et si c'est correcte, appuyez sur *Ok*.

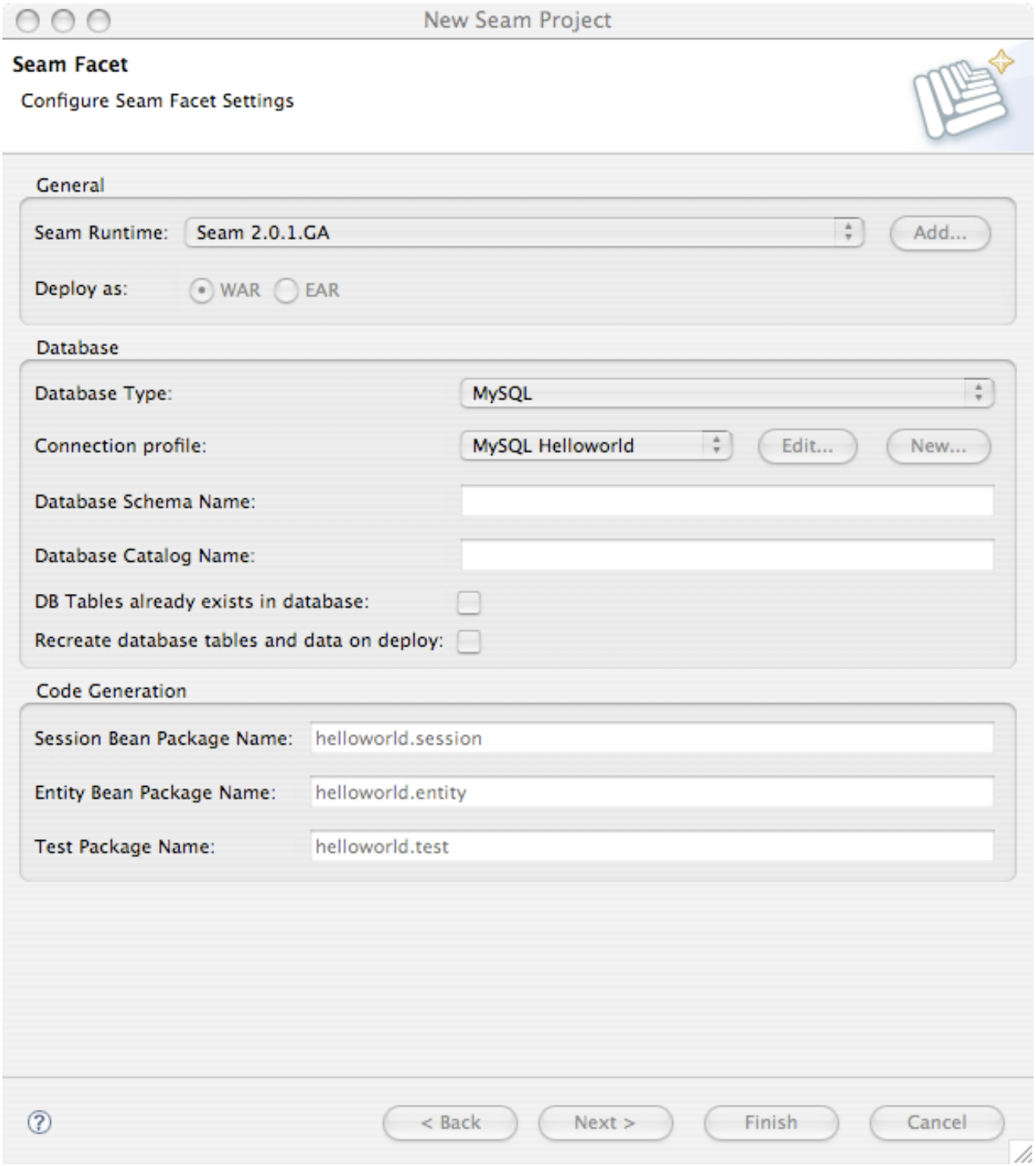
AU final, choisissez le pilote nouvellement créer:



Si vous travaillez avec un modèle de données existant, soyez sur de dire à JBoss Tools que des tables existent déjà dans la base de données.

Indiquez le nom d'utilisateur et le mot de passe utilisé pour se connecter, testez la connection en utilisant le bouton *Test Connection* , et si cela fonctionne, appuyez sur *Finish*:

Au final, vérifiez les noms des paquet de vos beans générés et si vous en êtes content, cliquez sur *Finish*:



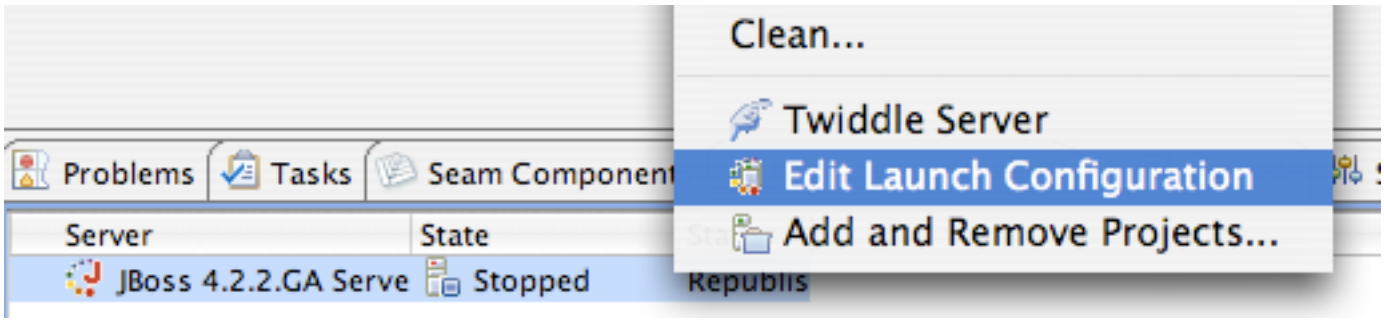
JBoss dispose d'un support sophistiqué pour le re-déploiement à chaud des EARs et des WARs. Malheureusement, à cause de bugs dans la JVM, le redéploiement d'un EAR—ce qui est common pendant le développement—peut éventuellement déclencher un épuisement de l'espace perm gen. Pour cette raison, nous recommendons d'exécuter JBoss dans une JVM avec un espace perm gen important pendant la phase de développement. Nous sudgerons les valeurs suivantes:

```
-Xms512m -Xmx1024m -XX:PermSize=256m -XX:MaxPermSize=512
```

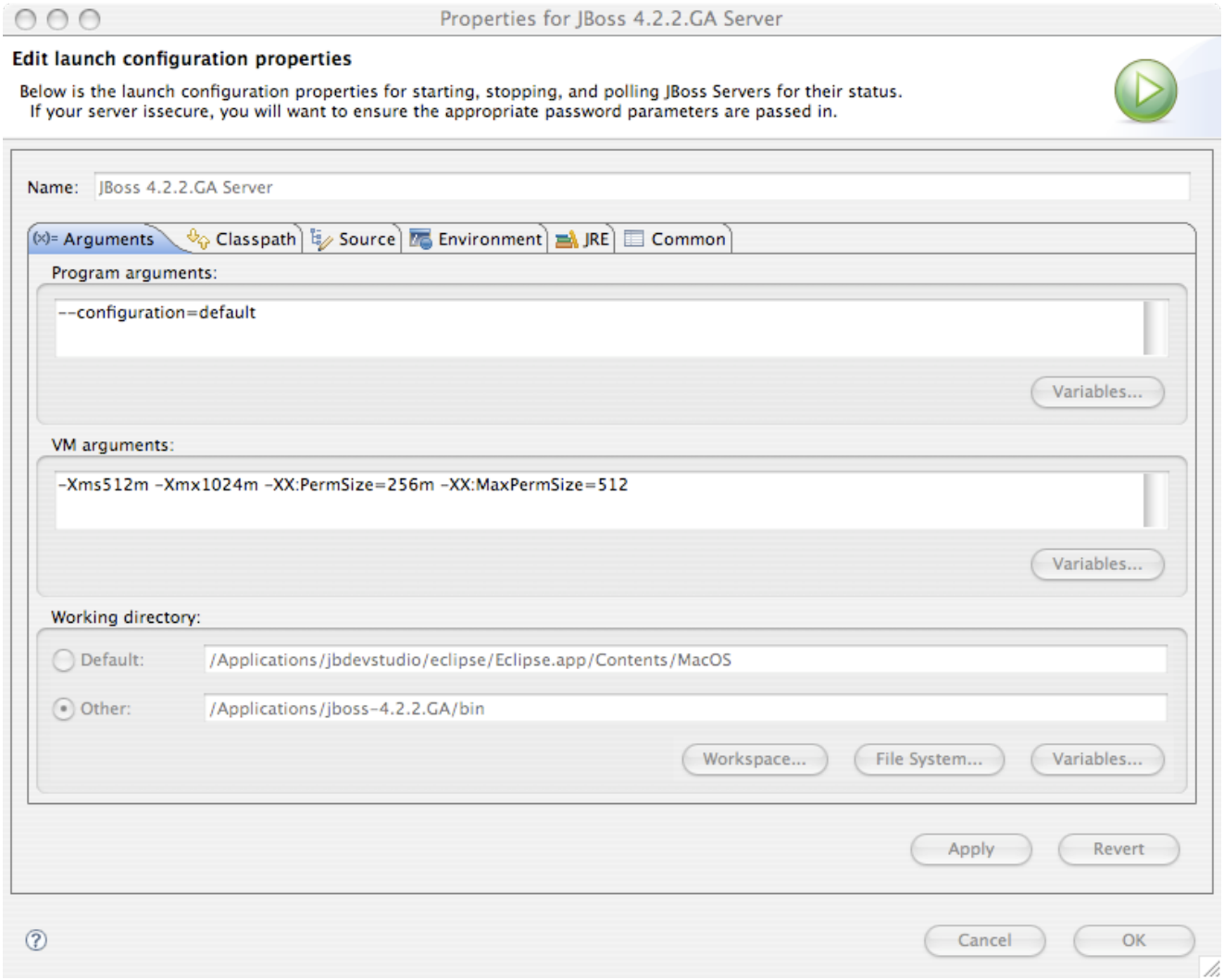
Si vous navez pas autant de mémoire disponible, les valeurs suivantes sont notre recommendation minimale:

```
-Xms256m -Xmx512m -XX:PermSize=128m -XX:MaxPermSize=256
```

Localisez le serveur dans le *JBoss Server View*, clic droit sur le server et selectionnez *Edit Launch Configuration*:

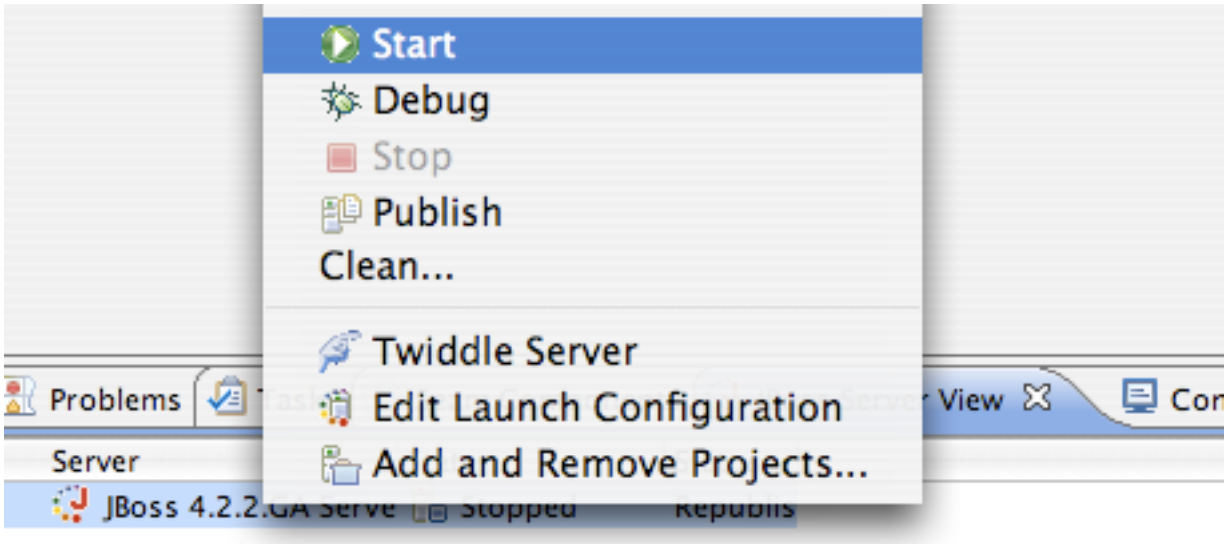


Ensuite, modifier les arguments de la VM:



SI vous ne voulez pas vous inquietes avec tout ce bidule maintenant, vous n'avez pas à le faire—revennez plus tard, quand vous aurez votre premier `OutOfMemoryException`.

Pour démarrer JBoss, et deployez votre projet, clic droit sipplement sur le server que vous avez créés et clic sur *Start*, (ou sur *Debug* pour démarrer en mode debogage):

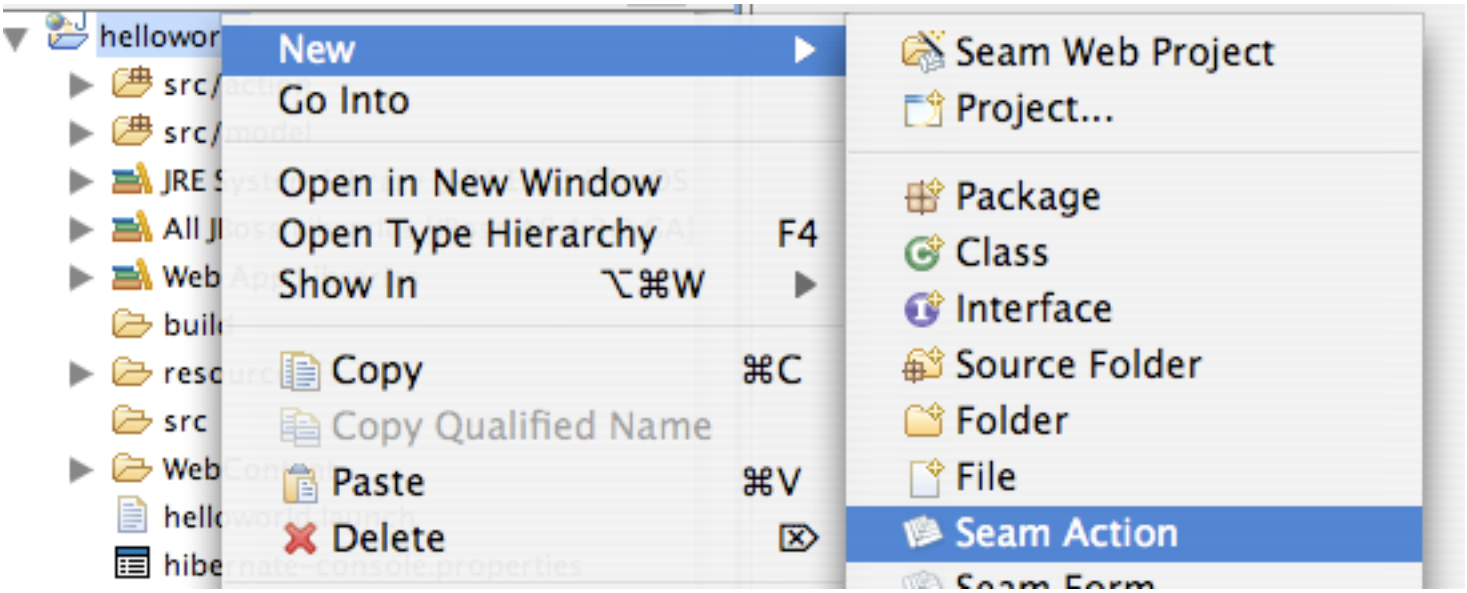


Ne soyez pas appeuré par les documents de configuration en XML qui sont générés dans le dossier du projet. Ils sont pour la plus par des trucs de Java EE standards, le genre de truc que vous avez besoin de créer une fois et ensuite de ne jamais plus le regarder, et ils sont dans 90% des cas les même entre tous les projets Seam.

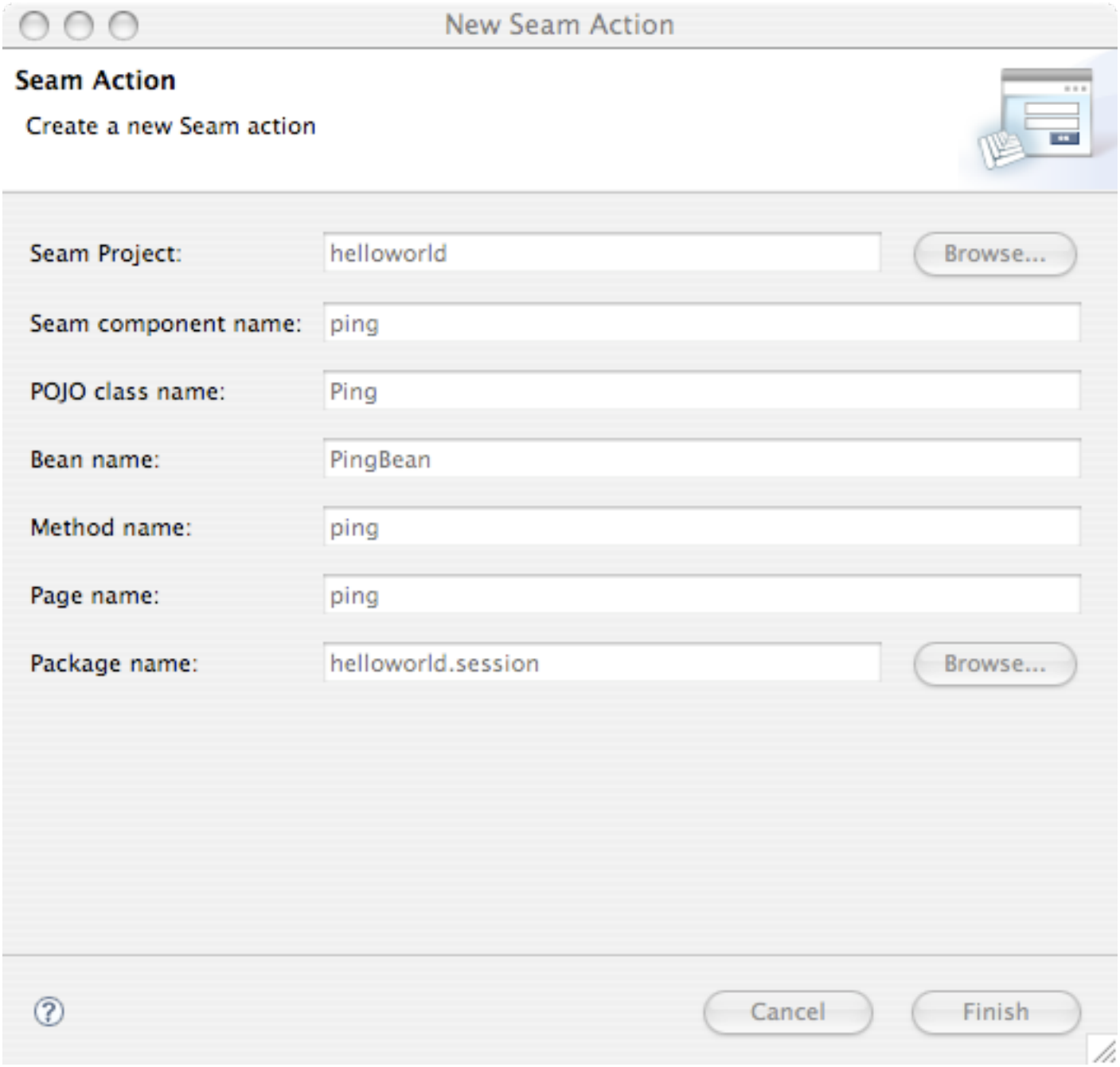
3.3. Création d'une nouvelle action

SI vous utilisez un serveur d'application web de style action traditionnel, vous vous inquiétez surement de comment créer une simple page web avec des méthode d'action sans état en Java.

En premier, sélectonnez *New -> Seam Action*:



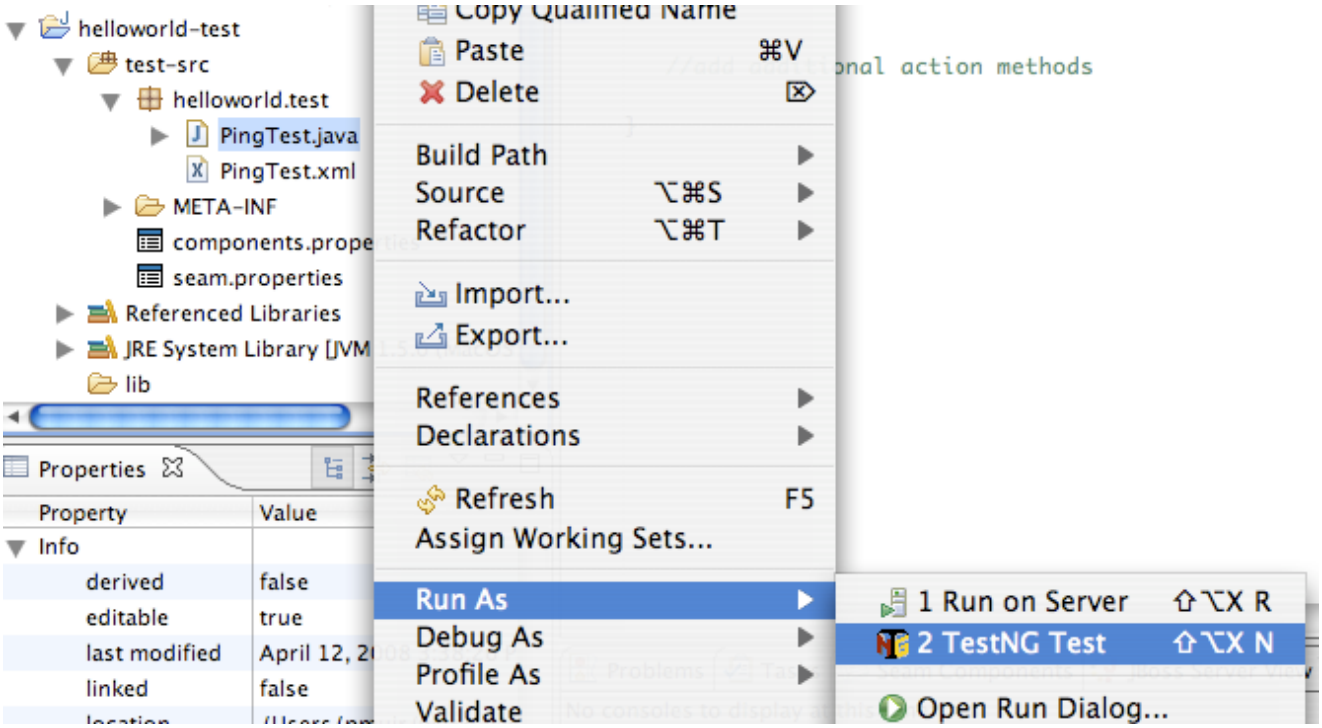
Maintenant, entrez le nom du composant de Seam. JBoss Tools sélectionne des valeurs par défauts judicieuses pour les autres champs:



Au final, appuyez sur *Finish*.

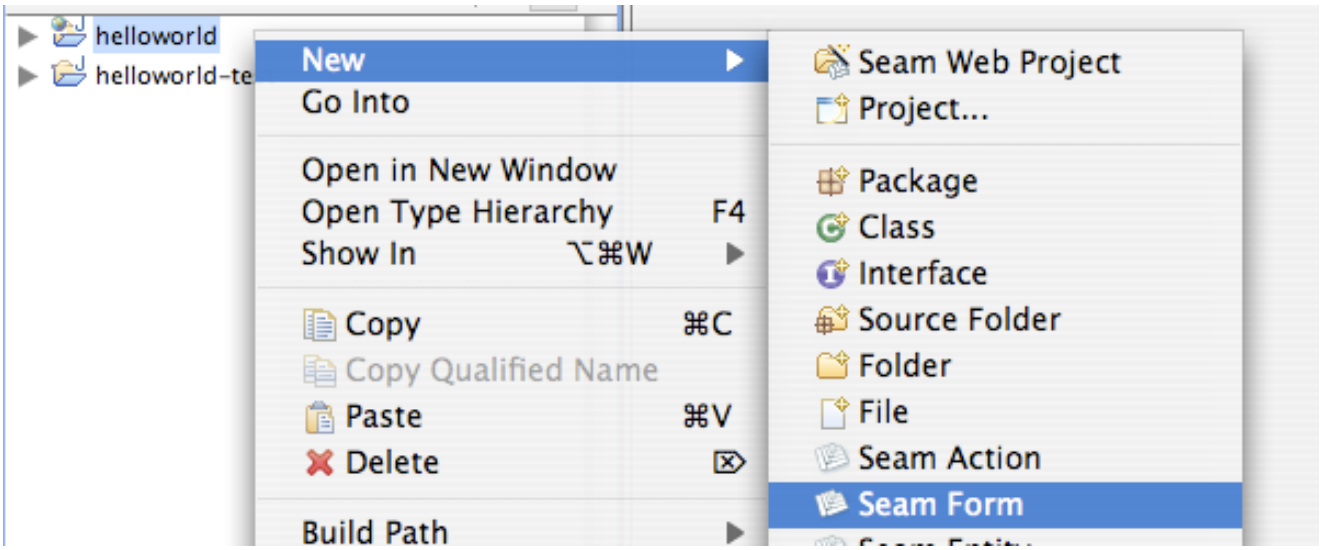
Maintenant allez sur `http://localhost:8080/helloworld/ping.seam` et cliquez sur le bouton. Vous pouvez voir le code derrière cette action en regardant dans le dossier `src` du projet. Metez un point d'arrêt dans la méthode `ping()` et cliquez de nouveau sur le bouton.

Au final, ouvrez le projet `helloworld-test`, localisez la classe `PingTest`, clic droit sur elle et choisissez *Run As -> TestNG Test*:

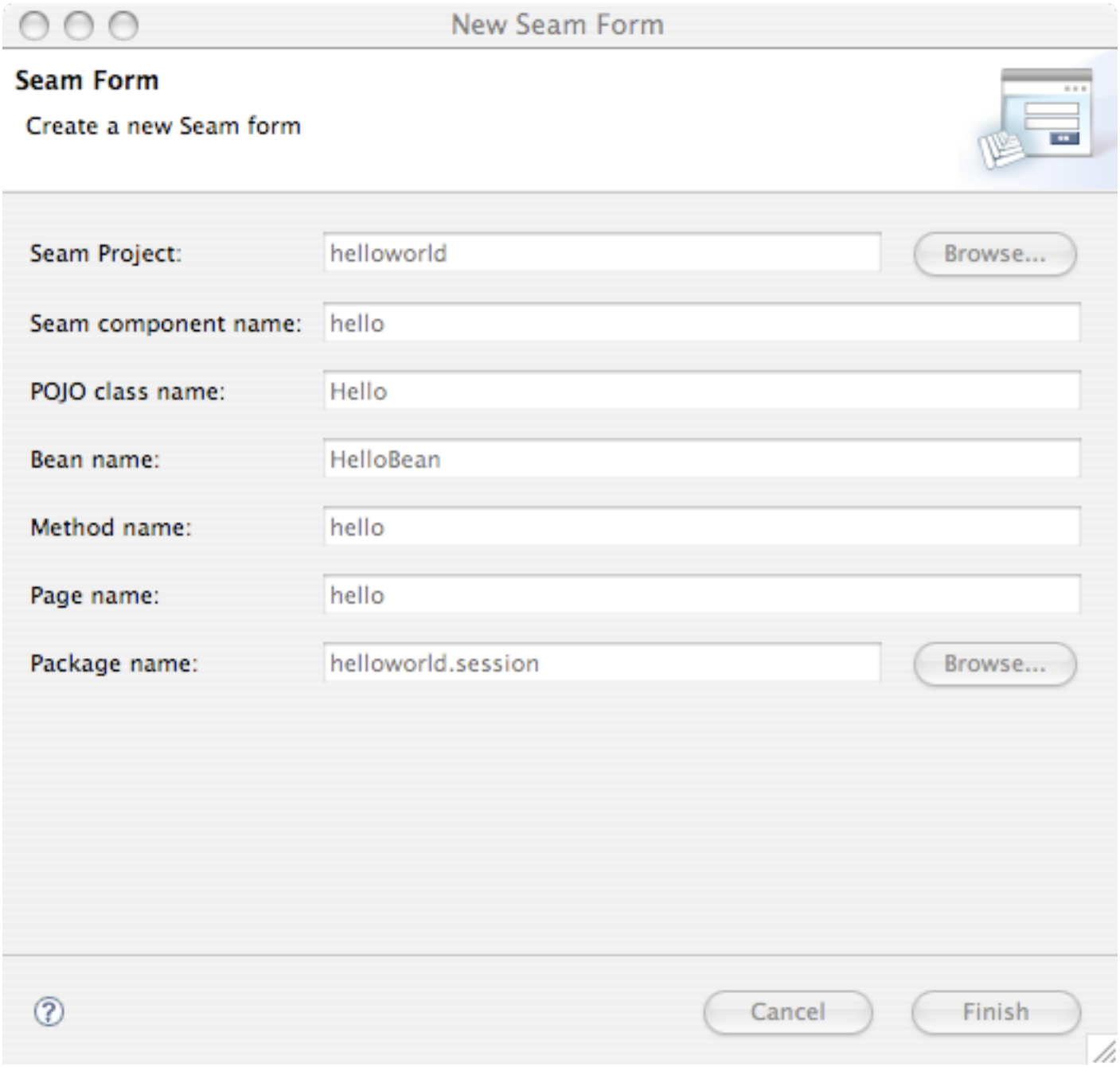


3.4. La création d'un formulaire avec une action

La première étape pour créer un formulaire. Sélectionnez *New* -> *Seam Form*:



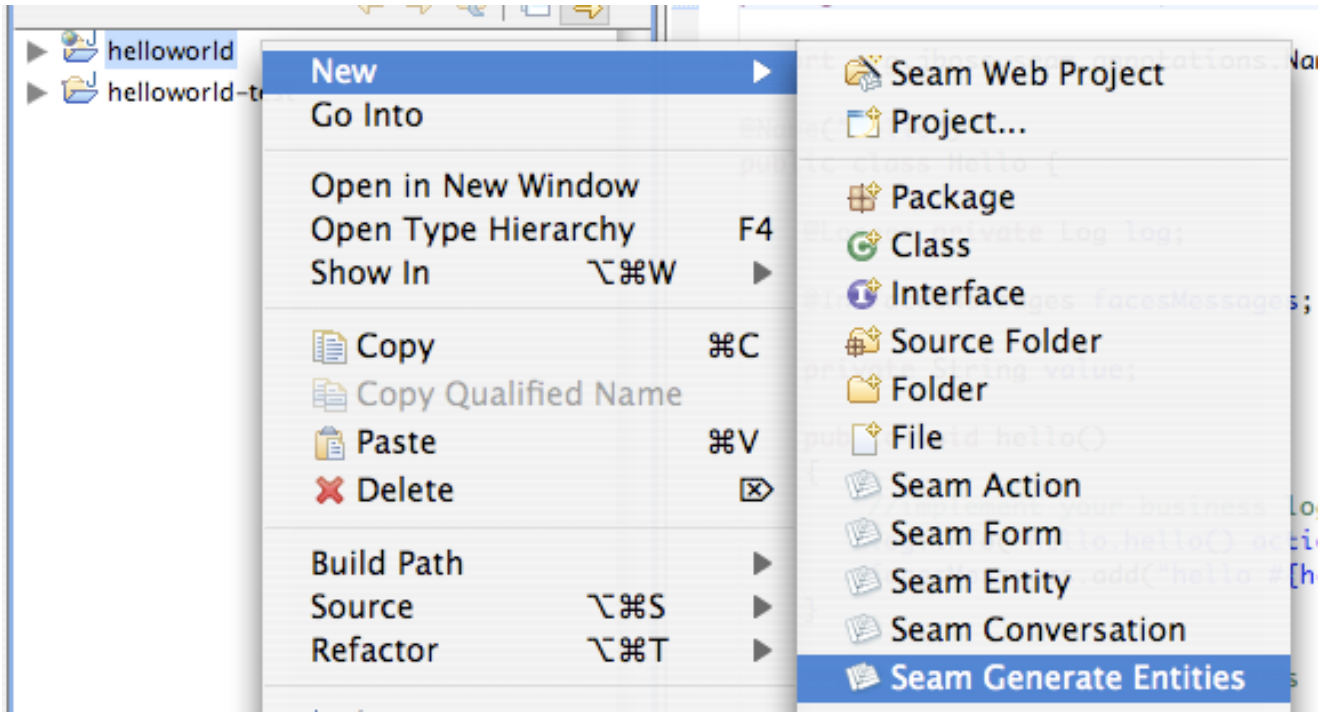
Maintenant, entrez le nom du composant de Seam. JBoss Tools sélectionne des valeurs par défauts judicieuses pour les autres champs:



Allez sur `http://localhost:8080/helloworld/hello.seam`. Ensuite, regardez comment est le code généré. Exécutez le test. Essayez d'ajouter de nouveaux champs au formulaire et au composant de Seam (notez, vous n'avez pas à redémarrer le serveur d'application à chaque fois que vous modifier le code dans `src/action` car Seam recharge à chaud le composant pour vous *Section 3.6, « Seam et le déploiement incrémental à chaud avec JBoss Tools »*).

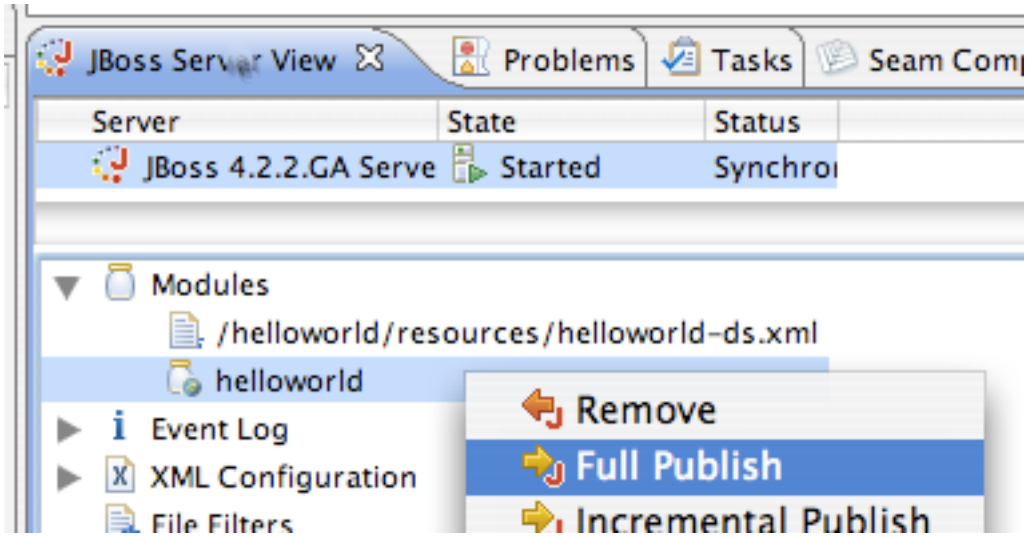
3.5. La génération d'une application depuis une base de données existante

Manuellement, créez quelques tables dans votre base de données. (Si vous avez besoin de basculer vers une base de données différente, créez une nouveau projet et sélectionnez la base de données correcte. Ensuite, sélectionnez *New -> Seam Generate Entities*:



JBoss Tools vous offre aussi l'option de réaliser une ingénieurie inverse des entitées, composants et des vues depuis votre schéma de base de données ou depuis les composants et les vues des entitées JPA existantes. Nous allonrs réaliser *une ingénieurie inverse depuis une abase de données*.

Redémarrer le déploiement:



Ensuite, allez vers `http://localhost:8080/helloworld`. Vous pouvez naviguer dans la base de données, editer des objets existants, et créez de nouveaux objets. Si vous regardez dans le code généré, vous allez probablement être impréssionné par sa simplicité! Seam a été conçu pour que le code d'accès aux données soit simple à écrire à la main, même par des gens qui ne veulent pas tricher en utilisant l'ingénieurie inverse.

3.6. Seam et le déploiement incrémental à chaud avec JBoss Tools

JBoss Tools permet le déploiement incrémental à chaud de :

- toute page facelets
- tout fichier `pages.xml`

au déballage de la boite.

Mais si nous voulons modifier n'importe quel code Java, nous avons toujours besoin de faire un redémarrage complet de l'application en faisant un *Full Publish*.

Mais si vous voulez un cycle rapide d'édition/compilation/test, Seam permet de redéploiement incrémental des composants JavaBean. Pour vous permetre d'utilisez cette fonctionnalité, vous devez déploiyer les composants JavaBean dans le dossier`WEB-INF/dev` , ainsi ils sont recharger par le classloader spécial de Seam, au lieu du classloader WAR ou EAR.

Vous devez être au courrant des limitations suivantes:

- les composants doivent être des composants JavaBean, il ne peuvent pas être des beans EJB3 (nous travaillons pour corriger cette limitation)
- les entités ne peuvent jamais être déploiyées à chaud.
- les composants déployés via `components.xml` ne peuvent pas être rechargés à chaud.
- les composants rechargés à chaud ne seront pas visible pour toute les classes déployés à l'extérieur de `WEB-INF/dev`
- Le mode de débogage de Seam doit être actif et `jboss-seam-debug.jar` doit être dans `WEB-INF/lib`
- Vous devez avoir le filtre Seam installé dans web.xml
- Vous pouvez voir des erreurs si le système est situé pendant le chargement ou le débogage est activé.

Si vous créez un projet WAR en utilisant JBoss Tools, le déploiement incrémental à chaud est disponible immédiatement pour les classes se trouvant dans le dossier des sources `src/action`. Cependant, JBoss Tools ne fourni pas de déploiement incrémental à chaud pour les projets EAR.

Le modèle de composant contextuel

Les deux concepts centraux dans Seam sont la notion de *contexte* et la notion de *composant*. Les composants sont des objets avec état, habituellement des EJBs, et une instance d'un composant est associé avec un contexte, et défini par un nom dans ce contexte. La *Bijection* fourni un mécanisme pour nommer les composants par un alias interne (les variables d'instances) dans les noms du contexte, autorisant des arbres de composants a être dynamiquement assemblés et réassemblés par Seam.

Commençons par la description des contextes existant dans Seam.

4.1. Les contextes de Seam

Les contextes de Seam sont créés et détruit par le serveur de squelette d'application (framework) via des appels explicite à l'API Java. Les contextes sont habituellement implicites. Dans certains cas, malgré tout, les contextes sont spécifiés via des annotations.

Les contextes de base de Seam

- Le contexte sans état
- Le contexte évènementiel (autrement dir, de requête)
- Le contexte de Page
- Le contexte conversatinnel
- Le contexte de Session
- Le contexte de processus métier
- Le contexte d'Application

Nous pouvons reconnaître certain de ces contexte de part les spécifications des servlets autour des servlets. Malgré cela, deux d'entre elles devraient être nouvelle pour vous : *contexte conversationnel*, et *contexte de processus métier*. Une des raisons pour que le gestionnaire d'état qui est dans les applications web, soit si fragile et un nid à erreurs c'est que trois des contextes livrés (requête, session et application) ne sont pas particulièrement utile du point de vue de la logique métier. Une session d'authentification d'un utilisateur, par exemple, est construite presque arbitrairement dans une application de flot de travail. Ainsi, les composants Seam sont bornés à la conversation ou aux contextes des processus métier, ainsi ils sont les contextes qui ont le plus de sens en termes d'application.

Ayons un regard sur chacun de ces contextes l'un après l'autre.

4.1.1. Le contexte sans état

Les composants qui sont réellement sans état (les beans sessions sans état, principalement) vivent toujours dans un contexte sans état (ce qui est basiquement une absence de contexte).

Les composants sans état ne sont pas vraiment intéressant et sont dénigrés pour ne pas être orienté objet. Malgré cela, ils sont importants, très souvent utile et aussi une part important de toute application Seam.

4.1.2. Le contexte évenementiel

Le contexte d'évènement est le contexte avec état le plus "réduit", et est la généralisation de la notion du contexte d'une requête web pour couvrir les autres types d'évènements. Malgré cela, le contexte d'évènement associé avec un cycle de vue dans une requête JSF est l'exemple le plus important d'un contexte d'évènement, et le seul qui va fonctionner le plus souvent. Les composants associés avec un contexte d'évènement sont détruit à la fin de la requête, mais leurs état est disponible et bien définie pour au moins le cycle de vie de la requête.

Quand vous invoquez un composant Seam via RMI ou Seam Remoting, le contexte d'évènement est créé et détruit juste pour l'invocation.

4.1.3. Le contexte de Page

Le contexte de page vous permet d'associer un état avec une instance particulière d'un page à rendre. Vous pouvez initialiser un état dans votre écouteur d'évènement ou au moment ou la page est en train d'être rendue et ensuite avoir un accès sur elle depuis tout évènement qui a comme origine cette page. Ceci est spécialement utile pour la fonctionnalité comme une liste cliquable quand la liste est controlée par une donnée changeante côté serveur. L'état est actuellement sérialisé vers le client, donc cette construction est extrêmement robuste dans le respect d'opération multi fenêtre et du bouton précédent.

4.1.4. Le contexte conversationnel

Le contexte de conversation est un vraiment un concept central dans Seam. Une *conversation* est une unité de travail du point de vue de l'utilisateur. Elle peut s'étendre sur plusieurs interactions de l'utilisateur, plusieurs requêtes, et plusieurs transactions de base de données. Mais pour l'utilisateur, une conversation résout un seul problème. Par exemple, "réserver un hôtel", "valider un contact", "créer un bon de commande" sont toutes des conversations. Vous devriez apprécier de penser en termes d'implémentation de la conversation comme un seul "cas d'utilisation" ou une "exemple d'utilisation" mais la relation n'est pas nécessairement aussi directe.

Une conversation retient un état associé avec "ce que l'utilisateur est en train de faire maintenant dans cette fenêtre". Un simple utilisateur peut avoir de multiples conversations en cours à tout moment dans le temps, habituellement dans de multiples fenêtres. Le contexte de conversation nous permet de nous assurer que cet état provenant de différentes conversations ne peut se caramboler et causer des bugs.

Il est possible que cela vous prennes du temps de maitriser la façon de penser l'application en termes de conversations. Mais une fois que vous avez l'habitude de le faire, nous pensons que vous allez adorer cette notion, et que vous ne serez plus capable de ne jamais plus penser en termes de conversations!

Quelques conversations perdurent dans une simple requête. Les conversations qui s'étendent sur plusieurs requêtes doivent être bien démarquées en utilisant les annotations fournies par Seam.

Quelques conversations sont aussi des *tâches*. une tache est une conversation ce qui a un sens en termes de processus métier à exécution longue et a le potentiel de déclencher une transition d'état pour un processus métier quand elle réussit à se terminer. Seam fournis une série d'annotations spéciales pourdifférencie cette tâche.

Les conversations doivent être *reliées*, avec une conversation prenant sa place "à l'intérieur" d'une plus grande conversation. Ceci est une fonctionnalité avancée.

Habituellement, l'état de la conversation est actuellement maintenu par Seam dans la session servlet entre les requêtes. Seam implémente *un delais de péremption* configurable, détruisant automatiquement les conversations inactives et ainsi s'assurant que l'état maintenu par une session de connexion d'un seul utilisateur ne grandi hors limitation si l'utilisateur abandonne les conversations.

Seam sérialise les requêtes de processus concurent qui ont lieu dans le même contexte de conversation à exécution longue, dans le même processus.

Autre alternative, Seam peut être configuré pour converser l'état conversationnel dans le navigateur du client.

4.1.5. Le contexte de Session

Un contexte de session conserve un état associé avec une session de connexion utilisateur. Pour les quelques cas où il est utilise de partager l'état entre plusieurs conversation, nous libèrons habituellement l'utilisation du contexte de session pour stocker les composants autre que les informations globales à propos de la connection de l'utilisateur..

Dans l'environnement portail JSR168, le contexte de session représente la session portail.

4.1.6. Le contexte de processus métier

Le contexte de processus métier stocke l'état associé au processus métier à exécution longue. Cet état est géré et est rendu persistant par le moteur BMP (JBoss JBPM). Le processus métier s'étend sur de multiples interactions avec de multiples utilisateurs, donc cet état est partagé entre plusieurs utilisateurs mais de manière parfaitement définie. La tache courante détermine l'instance du processus métier courante et le cycle de vie du processus métier est définie en externe en utilisant *un language de définition de processus*, donc il n'y a pas d'annotatation spéciale pour la séparation du processus métier.

4.1.7. Le contexte d'Application

Le contexte d'application est le contexte de servlet familier de la spécification servlet. Le contexte d'application est principalement utilisé pour stocker l'information statique comme des données de

configuration, des données de référence ou des méta-modèles. Par exemple, Seam stocke sa propre configuration et son méta-modèle dans le contexte d'application.

4.1.8. Les variables de contexte

Un contexte défini un espace de nom, un ensemble de *variables de contexte*. Ceci fonctionne à peu prêt comme des attributs de session ou de requête dans la spécification servlet. Vous devez faire correspondre n'importe quelle valeur que vous voulez à une variable de contexte, mais habituellement nous faisons correspondre les instances de composant Seam à des variables de contexte.

Donc, dans un contexte, une instance de composant est identifiée par le nom de la variable de contexte (ceci est habituellement, mais pas toujours, la même chose que le nom du composant). Vous pouvez par programmation accéder à l'instance du composant nommée dans une étendue particulière via la classe `Contexts`, qui fournis un accès aux différentes instances reliées par thread de l'interace `Context`:

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

Vous pouvez aussi définir ou changer la valeur associé avec son nom :

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

Habituellement, malgré tout, nous obtenons les composants depuis le contexte via une injection, et mettons les instances de composants dans le contexte via une extrusion.

4.1.9. Priorité dans la recherche de contexte

Parfois, comme ci-dessus, les instances des composants sont obtenus depuis une étendue connus particulière. D'autre fois, tous les étendues avec états sont parcourues, dans un *ordre de priorité*. Cet ordre est indiqué ci-dessous:

- Contexte d'évènement
- Le contexte de Page
- Le contexte conversatinnel
- Le contexte de Session
- Le contexte de processus métier
- Le contexte d'Application

Vous pouvez réaliser une recherche par priorité en appelant`Contexts.lookupInStatefulContexts()`. Malgré cela, si vous accédez a un composant par son nom depuis une page JSF, une recherche par ordre de priorité est faite.

4.1.10. Modèle de concurrence

Ni les spécifications servlet ni EJB ne définissent la moindre indications pour l'organisation des requêtes originaire du même client. Le container de servlet simplement laisse tous les threards s'exécuter de manière concurrente et laisse la sureté des threads se renforcer dans le code applicatif. Le container EJB autorise les composants sans état à être accédés de manière concurrente et déclenche des exceptions si de multiples threads accèdent à un bean session avec état.

Cette fonctionnalité devrait être ok dans les applications web au style ancien qui sont basées autour de requêtes bien dimensionnées et synchrones. Mais les applications modernes qui font une utilisation lourde de nombreuses requêtes bien dimensionnées et asynchrones (AJAX), de manière concurrente est un état de fait doivent être supporté par le modèle de programmation. Seam fourni une couche de gestion concurrente dans son modèle de contexte.

La session Seam et les contextes d'application sont multi threadées. Seam va nous autoriser des requêtes concurrentes dans un contexte pouvant être exécuté en concurrence. Les contextes page et d'évènements sont par nature un simple thread. Le contexte de processus métier est strictement parlant multi threadé, mais en pratique la concurrence est suffisamment rare pour que ce fait puisse être assez peu controlé la plus part du temps. Finalement, Seam renforce le modèle *d'un seul thread par conversation par processus* pour le contexte de conversation en sérialisant les requêtes concurrentes dans un même contexte de conversation d'exécution longue.

Depuis que le contexte de session est multi-threadé et souvent il contient un état volatile, les composants de l'étendue session sont toujours protégés par Seam d'accès concurrents aussi longtemps que les intercepteurs de Seam ne sont pas désactivés par ce composant. Si les intercepteurs sont désactivés, alors tout accès sécurisé au thread doit être implémenté par le concurent lui-même. Seam sérialise les requêtes dans les beans de session l'étendue de session et dans les JavaBeans par défaut (et détecte et brise toute étreinte mortele qui apparait). Ceci n'est pas la fonctionnalité par défaut pour les composants de l'étendue application, car les composants de l'étendue application ne conservent pas habituellement un état volatile et tout cela à cause de la synchronisation au niveau global qui est *extrèmement* couteuse. Malgré tout, vous pouvez forcer le modèle de threads en sérialisation sur n'imoporte quel bean session ou composant JavaBean en ajoutant l'annotation @Synchronized.

Le modèle concurrent signifie que les clients AJAX peuvent de manière sure utiliser la session volatile et un état conversationnel, sans le besoin d'un travail particulier de la part du développeur.

4.2. Les composants de Seam

Les composants de Seam sont ds POJOs (Plain old java Objects). En particulier, s'ils sont des JavaBeans ou bean entreprise EJB3.0. Tant que Sean ne requière pas que ces composants

soient des EJBs et ainsi qu'il puisse même être utilisés sans un container compatible EJB3, Seam a été conçu avec EJB3.0 comme état d'esprit et il inclus une intégration profonde avec EJB3.0. Seam supporte les *types de composants* suivants.

- Beans de session sans état EJB 3.0
- Beans de session avec état EJB 3.0
- Beans entité EJB 3.0 (autrement dit, des classes d'entité JPA)
- JavaBeans
- Les beans conducteur de message EJB 3.0
- Les beans de Spring (see *Chapitre 27, Spring Framework integration*)

4.2.1. Les beans de session sans état

Les composants de bean session sans état ne sont pas capables de retenir un état au travers de multiples invocations. Ainsi, ils travaillent habituellement en opérant au dessus de l'état d'autres composants dans les différents contextes de Seam. Ils peuvent être utilisés comme écouteur d'action JSF mais ne peuvent fournir une propriété à des composants JSF pour l'affichage.

Les beans session sans états existent toujours dans un contexte sans état.

Les beans de session sans état peuvent être accédés de manière concurente comme une nouvelle instance qui est utilisé pour chaque requête. Assigner l'instance à une requête est de la responsabilité du containeur EJB3 (les instances normallement seront alloués depuis un groupement réutilisable ce qui signifie que vous pourriez trouver des variables d'instance contenant des données d'un utilisation précédente du bean).

Les beans session sans états sont les moins intéressants des types de composant Seam.

Les beans session sans états peuvent être instanciés en utilisant `Component.getInstance()` ou `@In(create=true)`. Ils ne devraient pas être instanciés via l'observateur JNDI ou par l'opérateur `new`.

4.2.2. Les Beans de session avec état

Les composants beans de session avec état sont capables de retenir un état pas seulement au travers de multiples invocations du bean mais aussi au travers de multiples requêtes. L'état de l'application ne doit pas s'attarder dans la base de données pour être éventuellement retenu par des beans de sessions avec état. Ceci est une différence principale entre Seam et d'autres serveurs d'application web. Plutôt que de coller de l'information à propos de la conversation courante directement dans le `HttpSession`, vous devriez conserver ces variables d'instance d'un bean session avec état qui va être relié au contexte de conversation. Ceci autorise Seam à gérer

le cycle de vie de cet état pour vous et vous assure qu'il n'y aura pas de collision entre les états dans différentes conversations concurrentes.

Les beans de session avec état sont souvent utilisés comme écouteur d'action JSF, et comme beans de soutient qui fournissent des propriétés aux composants JSF pour l'affichage ou la soumissions de formulaire.

Par défaut, les beans de session avec état sont reliés au contexte de conversation. Ils ne peuvent jamais reliés aux contextes de page ou sans état.

Les requêtes concurrentes pour les sessions avec état dans l'étendue de session sont toujours sérialisées par Seam si les intercepteurs de Seam ne sont pas désactivés par le bean.

Les beans session avec états peuvent être instanciés en utilisant `Component.getInstance()` ou `@In(create=true)`. Ils ne devraient pas être instanciés via l'observateur JNDI ou par l'opérateur `new`.

4.2.3. Les beans entité

Les beans entité peuvent être reliés à une variable de contexte et fonctionne comme un composant de Seam. Avec des entitées ayant un identité persistante en plus de leur identité contextuelle, les instance d'entité sont habituellement reliées explicitement par le code Java, plutôt qu'être instanciées implicitement par Seam.

Les composants bean entité ne supportent pas la bijection ou la démarcation du contexte. Aucune invocation d'un bean entité ne déclenchera la validation.

Les beans entité ne sont habituellement pas utilisés comme écouteur d'action JSF, mais ont aussi la fonction d'un bean de soutient qui fourni des propriétés aux composants JSF pour l'affichage et la soumission de formulaire. En particulier, il est commun d'utiliser une entité comme un bean de soutient, avec un écouteur d'action de bean de session sans état pour implémenter les fonctionnalités de type création/mise à jour/effacement.

Par défaut, les beans entités sont reliés au contexte de conversation. Ils ne devraient jamais être reliés au contexte sans état.

Notez que dans un environement en cluster il est parfois moins efficace de relier un bean entité directement à une conversation ou à une variable de contexte de Seam d'étendue de session que de maintenir une reference au bean entité dans une bean de session avec état. C'est pour cette raison que toutes les applications Seam ne définissent pas des beans entité a être des composants Seam.

Les composant beans entité de Seam peuvent être instanciés en utilisant `Component.getInstance()` ou `@In(create=true)` ou directement en utilisant l'opérateur `new`.

4.2.4. JavaBeans

Les Javabeans devraient être utilisé tout comme un bean de session avec ou sans état. Magrè tout, ils ne fournissent pas les fonctionnalités de bean de session (démarcation de transaction

déclarative, sécurité déclarative, réplication d'état en cluster efficace, peristance EJB3.0, méthode hors-delais,etc).

Dans un chapitre suivant, nous vous montrerons comment utiliser Seam et Hibernate sans un container EJB. Dans ce cas d'utilisation, les composants sont des JavaBeans au lieu de beans de session. Notez, malgré cela, dans beaucoup de serveurs d'application il est parfois moins efficace de rassembler les conversations ou les composants JavaBeans de Seam d'étendue session que de rassembler les composants bean de session avec état.

Par défaut, les JavaBeans sont reliés au contexte d'évènement.

Les requêtes concurrentes dans les Javabeans d'étendue session sont toujours sérialisées par Seam.

Les composant JavaBean de Seam peuvent être instanciés en utilisant `Component.getInstance()` ou `@In(create=true)`. Ils ne devraient jamais être instancié en utilisant l'opérateur `new`.

4.2.5. Les beans conducteur-de-message

Les beans conducteur-de-message peuvent fonctionner comme un composant Seam. Malgrè cela, les beans conducteur-de-message sont appeler un peu différemment des autres composants de Seam- au lieu de les invoquer via une variable de contexte, ils écoutent les messages envoyés vers une file d'attente JMS ou un topic.

Les beans conducteur-de-message peuvent ne pas être reliés à un contexte de Seam. Ils n'ont pas non plus d'accès à la session ou l'état de conversation de leur "appelant". Magrè cela, ils peuvent supporter la bijection et quelques autres fonctionnalités de Seam.

Les beans conducteur-de-message ne sont jamais instanciés par l'application. Ils sont instanciés par le containeur EJB quand un message est reçu.

4.2.6. L'intercepteur

Pour réussir à faire sont tour de magie (bijection, démarcation de contexte, validation, etc.), Seam doit intercepter les invocations de composant. Pour les JavaBeans, Seam est un contrôleur total de l'instanciation du composant, et aucune configuration spéciale n'est requise. Pour les beans entité, l'interception n'est pas requise à moins d'une bijection et la démarcation de contexte ne sont pas défini. Pour les beans session, nous devons enregistrer un intercepteur EJB pour le composant bean de session. Nous pourrions utiliser une annotation, comme ci-dessous :

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

Mais la meilleure façon de faire est de définir un intercepteur dans `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
>
```

4.2.7. Les noms de component

Tous les composants de Seam au besoin d'un nom. Nous pouvons assigner un nom à un composant en utilisant l'annotation `@Name`:

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
   ...
}
```

Ce nom est un *nom de composant seam* et n'est pas lié a un tout autre nom défini par la spécification EJB. Malgré tout, les noms de composant Seam fonctionnent tous comme les noms des beans de gestion JSF et vous pouvez penser que les deux concepts sont identiques.

`@Name` n'est pas la seule façon de définir un nom de composant, mais nous avons toujours besoin de spécifier ce nom *quelque part*. Si nous ne le faisons pas, alors aucune autre annotation de Seam ne va fonctionner.

A chaque fois que Seam instancie un composant, il relie cette nouvelle instance à une variable dans l'étendue configuré pour ce composant qui correspondant au nom du composant. Cette fonctionnalité est identique à comment JSF fait travailler les beans qu'il gère, exception que Seam vous permet de configurer cette relation en utilisant des annotation au lieu de XMM. Par exemple, le connecté courant dans `User` peut être relié à une variable de contexte de session `currentUser`

tant que le `User` dont il est sujet à quelques fonctionnalités administratives qui peuvent être reliées à la variable de contexte de conversation user. Soyez prudent, pensez-y, parce qu'au traver d'une affectation programmée, il est possible de surcharger une variable de contexte qui a une référence sur un composant de Seam, avec de potentiel problèmes de confusion.

Pour de plus grande applications, et pour les composants de Seam livrés, les noms qualifiés sont souvent utilisés.

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

Nous devrions utiliser le nom de composant qualifié à la fois dans le code Java et dans le langage d'expression de JSF:

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

Comme c'est perturbant, Seam fourni aussi une manière pour renommer un nom qualifié en nom simple. Ajouter une ligne comme celle-ci dans le fichier components.xml:

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

Tous les composants Seam livrés ont un nom qualifié, mais la plus part d'entre eux sont renommés vers un nom simple grâce à la fonctionnalité d'importation de l'espace de nommage de Seam. Le fichier `components.xml` inclus dans le JAR de Seam définie les espaces de nommage suivant.

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

   <import>org.jboss.seam.core</import>
   <import>org.jboss.seam.cache</import>
   <import>org.jboss.seam.transaction</import>
   <import>org.jboss.seam.framework</import>
   <import>org.jboss.seam.web</import>
   <import>org.jboss.seam.faces</import>
   <import>org.jboss.seam.international</import>
   <import>org.jboss.seam.theme</import>
   <import>org.jboss.seam.pageflow</import>
```

```
    <import>org.jboss.seam.bpm</import>
    <import>org.jboss.seam.jms</import>
    <import>org.jboss.seam.mail</import>
    <import>org.jboss.seam.security</import>
    <import>org.jboss.seam.security.management</import>
    <import>org.jboss.seam.security.permission</import>
    <import>org.jboss.seam.captcha</import>
    <import>org.jboss.seam.excel.exporter</import>
    <!-- ... --->
</components>
```

Pour essayer de résoudre un nom non-qualifié, Seam va vérifier chacun de ces espace de nommage, dans l'odre. Vous pouvez inclure des espaces de nommages additionnels dans votre fichier `components.xml` de votre application pour des espaces de nommages spécifiques à votre application.

4.2.8. Définition de l'étendue de composant

Nous pouvons surchargé létendue par défaut (le contexte) d'un composant en utilisation l'annotation `@Scope`.Ceci nous permets de définir quel contexte une instance de composant est relié à, quand il est instancié par Seam.

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

`org.jboss.seam.ScopeType` defini une énumération des étendues possibles.

4.2.9. Les composants avec des rôles multiples

Des classes de composants Seam peuvent être utilisées avec plus d'un rôle dans le système. Par exemple, nous avons souvent une classe `User`qui est habituellement utilisée comme un composant d'étendue de session représentant l'utilisateur courant mais elle est aussi utilisée dans les écrans de l'administration de l'utilisateur comme un composant d'étendue de conversation. L'annotation `@Role` nous permet de définir un rôle avec un nom additionnel, avec une étendue différente — il nous laisse relier la même classe de composant dans des variables de contextes différentes. (Toute *iinstance*i de composant Seam peut être reliée à de multiples variables de contexte, mais cela nous laisse le faire au niveau classe et d'avoir l'avantage d'une auto instanciation.)

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

L'annotation `@Roles` nous permet de spécifier autant de rôles additionnels que nous voulons.

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

4.2.10. Les composants livrés

Tout comme beaucoup de bon serveur d'application, Seam mange sa propre nourriture et il implémente un série d'intercepteurs livrés (voir ci-dessous) et de composants de Seam. Ceci rend plus facile pour les applications d'interagir avec les composants livrés à l'exécution ou même de personnaliser les fonctionnalités de base de Seam en remplaçant les composants livrés avec des implémentations personnalisées. Les composants livrés sont défini dans l'espace de nommage de `org.jboss.seam.core` et le paquet Java du même nom.

Les composants livrés peuvent être injectés, tout comme des composants Seam, mais ils fournissent aussi des méthodes `instance()` pratiques statiques:

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

4.3. La bijection

L'injection de dépendance ou *l'inversion de contrôle* est maintenant un concept familier pour la plus part des développeurs Java. L'injection de dépendance permet à un composant d'obtenir une référence sur un autre composant en passant par le containeurs qui "injecte" l'autre composant par une méthode assesseur ou par une variable d'instance. Dans toutes les implémentations d'injection de dépendances que nous avons vu, l'injection intervient quand le composant est construit, et la référence ne change pas postérieurement pour le cycle de vie de l'instance du

composant. Pour les composants sans état, ceci est raisonnable. Du point de vue d'un client, toutes les instances d'un composant sans état particulier sont interchangeables. Dans un autre côté, Seam accentue l'utilisation de composants avec état. Donc l'injection de dépendance traditionnelle n'est plus une construction vraiment utilisable. Seam introduit la notion de *bijection* comme une généralisation de l'injection. A la différence de l'injection, la bijection est :

- *contextuelle* - la bijection est utilisé pour assembler les composants avec état depuis plusieurs contextes différents (un composant d'un contexte "évasé" peut même contenir une référence sur un composant d'un contexte "étroit")
- *bidirectionelle* - les valeurs sont injectées depuis les variables de contexte dans des attributs du composant qui a été invoqués et aussi *extradé* depuis les attributs du composant en retour vers le contexte, autorisant le composant à être invoqué pour manipuler les valeurs des variables contextuelles simplement en définissant ces propres variables d'instances
- *dynamique* - avec une valeur des variables contextuelles changeante au cours du temps et avec des composants Seam qui sont avec état, la bijection a lieu à chaque fois qu'un composant est invoqué

Par essence, la bijection vous permet de renommer une variables de contexte dans une variables d'instance de composant en spécifiant que la valeur de la variable d'instance est injectée, extradée ou les deux. Bien sûr, nous utilisons des annotation pour activer la bijection.

L'annotation `@In` indique que la valeur devrait être injectée, aussi dans une variable d'instance.

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

ou dans une méthode assesseur :

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

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

   ...
```

```
}
```

Par défaut, Seam va donner une priorité de recherche dans tous les contextes en utilisant le nom de la propriété ou de la variable d'instance qui a été injectée. Vous devriez vouloir spécifier le nom de la variable de contexte explicitement, en utilisant, par exemple, `@In("currentUser")`.

Si vous voulez que Seam puisse créer une instance du composant quand il n'y a pas d'instance de composant existante reliée à une variable de contexte nommée, vous pouvez spécifier `@In(create=true)`. Si la valeur est optionnel (elle peut être null) spécifiez `@In(required=false)`.

Pour quelques composants, il peut être répétitif d'avoir à spécifier `@In(create=true)` partout où ils sont utilisés . Dans ce genre de cas, vous pouvez annoter le composant `@AutoCreate`, et alors il va toujours être créé, n'important quand, même sans l'utilisation explicite de `create=true`.

Vous pouvez même injecter la valeur d'une expression :

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

Les valeurs injectées sont désalouées (autrement dit, définie à `null`) immédiatement après la réalisation de la méthode et sont extraction.

(Il y a beaucoup plus d'information sur le cicle de vie d'un composant et l'injection dans le chapitre suivant).

L'annotation `@Out` indique qu'un attribue devrait être extradé, tout comme une variable d'instance:

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

ou comme une méthode assesseur:

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

```
public class LoginAction implements Login {
   User user;

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

   ...
}
```

Un attribut peut être à la fois injecté et extradé:

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

ou:

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

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

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

   ...
}
```

4.4. Les méthode du cycle de vie

Le bean de session et les composants Seam de bean entité supportent tous les rappels du cycle de vie EJB3.0 usuel (`@PostConstruct`, `@PreDestroy`, etc). Mais Seam étend tous ces rappels aux composant JavaBean. Cependant tant que ces annotation ne sont pas disponible dans un environement J2EE, Seam défini deux composants additionnel pour le rappel dans le cycle de vie, équivalent à `@PostConstruct` et à `@PreDestroy`.

La méthode `@Create` est appelée après que Seam instancie un composant. Les composants ne peuvent définir seulement qu'une seule méthode `@Create`.

La méthode `@Destroy` est appelé quand le contexte qui lie le composant de Seam se termine. Les composants ne peuvent définir qu'une seule méthode `@Destroy`.

De plus, les composants bean avec état *doivent* définir une méthode sans aucun paramètre annotée `@Remove`. Cette méthode est appelé par Seam quand le contexte se termine.

Finalement, une annotation liée est l'annotation `@Startup` qui peut être appliquée à toutes l'application ou un composant d'étendue session. L'annotation `@Startup` indique à Seam d'instancier le composant immédiatement, quand le contexte commence, au lieu d'attendre avant son premier référencement par un client. Il est possible de controler l'ordre d'instanciation du démarrage des composants en spécifiant `@Startup(depends={....})`.

4.5. Installation conditionelle

L'annotation `@Install` vous permet de contrôler l'installation conditionnelle du composant qui sont requis dans certains scénarios de déploiement et pas dans d'autres. C'est utile si :

- Vous voulez singer quelques composants d'infrastructure dans des tests.
- Vous voulez modifier l'implémentation d'un composant dans certains scénarios de déploiement.
- Vous voulez installer quelques composants seulement si leurs dépendances sont disponibles (utile pour les auteurs des serveur d'applications).

`@Install` fonctionne en vous laissant spécifier la *précédence* et les *dépendances*.

La précédence d'un composant est un numéro que Seam utilise pour décider quel composant est à installer quand il a plusieurs classes avec le même nom de composant dans le classpath. Seam va choisir le composant avec la précédence la plus haute. Il y a quelques valeurs de précédences prédéfinie (dans l'ordre ascendant) :

1. `BUILT_IN` — les composants de précédence la plus faible sont les composants livrés avec Seam.
2. `FRAMEWORK` — les composants définis par un serveur d'application tierces peuvent surcharger les composants livrés, mais ils sont surchargés par les composants d'application.
3. `APPLICATION` — la précédence par défaut. C'est la valeurappropriée pour la plus part des composants d'application.

4. `DEPLOYMENT` — pour les composants d'application qui ont un déploiement spécifique.

5. `MOCK` — pour les objets singeant qui sont utilisés en test.

Supposez que nous avons un composant appelé `messageSender` qui parle à la queue de JMS.

```
@Name("messageSender")
public class MessageSender {
   public void sendMessage() {
      //do something with JMS
   }
}
```

Si dans vos tests unitaires, nous n'avons pas de queue de JMS disponible, mais que nous aimerions simuler cette méthode. Nous allons créer un composant le *singeant* et qui existe dans le classpath quand les tests unitaires sont exécutés mais il n'est jamais déployé avec l'application:

```
@Name("messageSender")
@Install(precedence=MOCK)
public class MockMessageSender extends MessageSender {
   public void sendMessage() {
      //do nothing!
   }
}
```

La `precedence` aide Seam à décider quel version à utiliser quand il trouve deux composants dans la classpath.

C'est bien si vous étiez capable de contrôler exactement quelles classes sont dans le classpath. Mais si j'ai écrit un serveur d'application réutilisable avec beaucoup de dépendances, je ne veux pas à avoir à briser ce serveur d'applications au travers de nombreux jars. Je veux d'être capable de décider quel composants sont à installer selon quel autres composants sont déjà installés et selon quelles classes sont disponibles dans le classpath. L'annotation `@Install` contrôle aussi cette fonctionnalité. Seam utilise ce mécanisme interne pour activer l'installation conditionnelle de beaucoup des composants livrés. Malgré tout, vous n'aurez probablement pas besoin de l'utiliser dans votre application.

4.6. Mettre des traces

Qui n'en n'a pas assez de voir du code aussi touffu que celui-ci ?

```
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);
}
```

Il est difficile d'imaginer comment le code pour un simple message de log peut devenir beaucoup verbeux. Il y a beaucoup plus de lignes de code dédié à l'affichage des traces plus que de la réelle logique métier! Je reste totalement abasourdie que la communauté Java n'a pas encore trouvé mieux depuis 10 ans.

Seam fourni un API d'affichage des traces qui simplifie significativement le code :

```
@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.name(), quantity);
   return new Order(user, product, quantity);
}
```

Il n'est pas utile si vous déclarez la variable `log` statique ou non— elle va fonctionner dans tous les cas, à l'exception des composants bean entité qui demande à la variable de `log` d'être statique.

Notez que nous n'avons pas besoin d'être touffu `si ( log.isDebugEnabled() )` conserver, depuis la concaténation des chaines de caractères apparaissent *dans* la méthode `debug()`.Notez aussi que nous n'avons habituellement pas besoin de spécifier la catégorie de log explicitement, car Seam connaît quel composant au sein du quel il va injecter le `Log`.

Si `User` et `Product` sont des composants Seam disponible dans les contextes courant, il n'y a pas mieux :

```
@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", quantity);
   return new Order(user, product, quantity);
}
```

Seam logue auto-magiquement choisi s'il faut envoyer la sortie vers log4j ou vers le logging JDK. Si log4j est dans le classpath, Seam va l'utiliser. Si ce n'est pas le cas, Seam va utiliser le logging JDK.

4.7. Les interfaces `Mutable` et `@ReadOnly`

Plusieurs serveurs d'applications fonctionnent comme d'incroyable implémentation brissant le clustering `HttpSession` quand les modifications d'état d'objet mutables relié à une session sont seulement répliqués quand l'application appelle `setAttribute()` explicitement. Ceci est la source de bugs qui ne peuvent pas effectivement être testé pendant le temps de développement, jusqu'à qu'ils se manifestent quand les erreurs surviennent. En outre, le message actuel répliqué contient le graphe d'objet intégralement sérialisé relié à l'attribut de la session, ce qui est inefficace.

Bien sur, les beans de session avec état EJB peuvent réaliser automatiquement la sale vérification et la réplication des états mutés et un containeur EJB sophistiqué peut introduire des optimisations comme la réplication au niveau attribut. Malheureusement, tous les utilisateurs de Seam n'ont la bonne surprise de travailler dans un environnement qui supporte EJB3.0. Donc, pour la session et le JavaBean d'étendue conversationnelle et les composants bean entité, Seam propose une couche additionnelle pour le gestionnaire d'état sécurisé pour le cluster au dessus du clustering de session du container web.

Pour la session ou des composants de JavaBeans d'étendue conversationnel, Seam automatiquement force la réplication qui survient en appelant`setAttribute()` une fois sur chaque requète quand le composant est invoqué par l'application.Bien sur, cette stratégie est inéficace pour les composants principalement en lecture seul. Vous pouvez controler cette fonctionnalité en implémentant l'interface `org.jboss.seam.core.Mutable` et écrire votre propre logique de sale-vérification dans votre composant. Par exemple :

```
@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;
   }
```

```
    ...

}
```

Ou, vous pouvez utiliser l'annotation @ReadOnly pour obtenir un effet similaire :

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

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

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

    ...

}
```

Pour la session ou pour des composants bean entité d'étendue conversation, Seam automatiquement forces la réplication qui intervient en appelant `setAttribute()` une fois par requête, *à moins que l'entité (d'étendue conversationnelle) ne soit actuellement associée avec le contexte de persistance gérée par Seam, dans ce cas aucune réplication n'est requise*. Cette stratégie n'est pas nécéssairement efficace, donc une session ou des beans entité d'édendue conversationnelle devraient l'utiliser avec prudence. Vous devez toujours écrire un bean session avec état ou un composant JavaBean pour "gérer" l'instance du bean entité. Par exemple,

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

    @Unwrap
    public Account getAccount()
```

```
    {
        return account;
    }

    ...

}
```

Notez que la classe `EntityHome` dans le Seam Application Framework fourni un bel exemple de gestion d'une instance bean entité utilisant un composant Seam.

4.8. Fabrique et composants gestionnaire

Nous avons souvent besoin de travailler avec des objets qui ne sont pas des composants Seam. Mais nous continuons à vouloir être capable de les injecter dans notre composants en utilisant `@In` et de les utiliser en tant que valeur et méthode reliées aux expressions, etc. De temps en temps, nous avons même besoin de les attacher dans le cycle de vie du contexte de Seam (`@Destroy`, par exemple). Donc les contextes de Seam peuvent obtenir des objets qui ne sont pas des composants Seam et Seam fourni quelques fonctionnalités sympa qui rendent plus facile le travail avec les objets non-composants reliés aux contextes.

Le *modèle de conception fabrique de composant* autorise un composant Seam à agir comme l'instanciateur pour un objet non-composant. Une *méthode fabrique* va être appelée quand une variable du contexte est référencée mais n'a pas de valeur reliée à elle. Nous définissons des méthodes fabrique en utilisant l'annotation `@Factory`. La méthode fabrique relie une valeur à une variable de contexte et détermine l'étendue de la valeur liée. Il y a deux types de méthode fabrique. La première méthode retourne une valeur, qui est relié au contexte par Seam :

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

Le second style est une méthode de type void qui relie cette valeur à la variable de contexte elle-même :

```
@DataModel List<Customer
> customerList;

@Factory("customerList")
public void initCustomerList() {
```

```
   customerList = ... ;
}
```

Dans les deux cas, la méthode fabrique est appelée quand nous référençons la variable de contexte `customerList` et cette valeur est null, et alors n'a absolument pas à intervenir dans le cycle de vie de la valeur. Et même un patron de conception plus puisssant est *le pattron de conception gestionnaire de composant*. Dans ce cas, nous avons un composant Seam qui est relié à une variable de contexte qui gère la valeur de la variable de contexte restant invisible aux clients.

Un composant gestionnaire est n'importe quel composant avec une méthode `@Unwrap`. Cette méthode retourne la valeur qui va être visible pour les clients, et elle est appeléeà *chaque fois* qu'une variable de contexte est référencée.

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

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

Le patron de conception gestionnaire est particulièrement utile si nous avons un objet où nous avons besoin de contrôler le cycle de vie du composant. Par exemple, si vous avez des objets très lourds qui ont besoin d'opération de nettoyage quand le contexte s'arrête, vous pouvez `@Unwrap` l'objet et réaliser un nettoyage dans la méthode `@Destroy` du gestionnaire.

```
@Name("hens")
@Scope(APPLICATION)
public class HenHouse
{
   Set<Hen
> hens;

   @In(required=false) Hen hen;

   @Unwrap
```

```
    public List<Hen
> getHens()
    {
        if (hens == null)
        {
            // Setup our hens
        }
        return hens;
    }

    @Observer({"chickBorn", "chickenBoughtAtMarket"})
    public addHen()
    {
        hens.add(hen);
    }

    @Observer("chickenSoldAtMarket")
    public removeHen()
    {
        hens.remove(hen);
    }

    @Observer("foxGetsIn")
    public removeAllHens()
    {
        hens.clear();
    }
    ...
}
```

Ici le composant gestionnaire observe tous les évènement qui modifie l'objet sous-jacent. Le composant gère ces actions lui-même, et parce que l'objet est empaqueté donc à chaque accès, une vue cohérante est fournie.

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 in `web.xml`, and configuration via `components.xml`.

5.1. Configuring components via property settings

Seam components may be provided with configuration properties either via servlet context parameters, via system properties, 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, a system property named `org.jboss.seam.properties.com.jboss.myapp.settings.locale` via -D at startup, 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`, `seam.properties`, or via a system property prefixed with `org.jboss.seam.properties`. (There is a built-in Seam component named `org.jboss.seam.core.manager` with a setter method named `setConversationTimeout()`.)

5.2. 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 with 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` annotation 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 jBPM:

```
<components xmlns="http://jboss.com/products/seam/components"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:bpm="http://jboss.com/products/seam/bpm">
   <bpm:jbpm/>
</components>
```

This example does the same thing:

```
<components>
   <component class="org.jboss.seam.bpm.Jbpm"/>
</components>
```

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

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

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

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

</components>
```

As does this one:

```
<components>
```

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

  <component name="accountingDatabase"
        class="org.jboss.seam.persistence.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:persistence="http://jboss.com/products/seam/persistence"

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

</components>
```

```
<components>

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

</components>
```

It is common to use the `auto-create` option for infrastructural objects like persistence contexts, which saves you from having to explicitly specify `create=true` when you use the `@In` annotation.

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

```
 <persistence:managed-persistence-context name="productDatabase"
                                  auto-create="true"
                        persistence-unit-jndi-name="java:/productEntityManagerFactory"/>

</components>
```

```
<components>

   <component name="productDatabase"
      auto-create="true"
           class="org.jboss.seam.persistence.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>
```

It is especially common to see the use of `auto-create="true"` with the `<factory>` declaration:

```
<components>

    <factory name="session" value="#{entityManager.delegate}" scope="STATELESS" auto-
create="true"/>

</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.

5.3. 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`.

5.4. 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.bpm.jbpm.processDefinitions order.jpdl.xml, return.jpdl.xml, inventory.jpdl.xml
```

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

```
<component name="org.jboss.seam.bpm.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>
```

When configuring multi-valued properties, by default, Seam will preserve the order in which you place the attributes in `components.xml` (unless you use a `SortedSet`/`SortedMap` then Seam will use `TreeMap`/`TreeSet`). If the property has a concrete type (for example `LinkedList`) Seam will use that type.

You can also override the type by specifying a fully qualified class name:

```
<component name="issueEditor">
  <property name="issueStatusOptions" type="java.util.LinkedHashMap">
    <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>
```

Seam also resolves an EL expression string prior to assigning the initial value to the bean property of the component. So you can inject some contextual data into your components.

```
<component name="greeter" class="com.example.action.Greeter">
    <property name="message">Nice to see you, #{identity.username}!</property>
</component>
```

However, there is one important exception. If the type of the property to which the initial value is being assigned is either a Seam `ValueExpression` or `MethodExpression`, then the evaluation of the EL is deferred. Instead, the appropriate expression wrapper is created and assigned to the property. The message templates on the Home component from the Seam Application Framework serve as an example.

```
<framework:entity-home name="myEntityHome"
    class="com.example.action.MyEntityHome" entity-class="com.example.model.MyEntity"
    created-message="'#{myEntityHome.instance.name}' has been successfully added."/>
```

Inside the component, you can access the expression string by calling `getExpressionString()` on the `ValueExpression` or `MethodExpression`. If the property is a `ValueExpression`, you can resolve the value using `getValue()` and if the property is a `MethodExpression`, you can invoke the method using `invoke(Object args...)`. Obviously, to assign a value to a `MethodExpression` property, the entire initial value must be a single EL expression.

5.5. 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:

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://jboss.com/products/seam/components"
            xsi:schemaLocation="http://jboss.com/products/seam/components http://jboss.com/
products/seam/components-2.2.xsd">

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

```
</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-2.2.xsd
                    http://jboss.com/products/seam/components  http://jboss.com/products/seam/
components-2.2.xsd">

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

</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`
- security — `http://jboss.com/products/seam/security`

- mail — `http://jboss.com/products/seam/mail`
- web — `http://jboss.com/products/seam/web`
- pdf — `http://jboss.com/products/seam/pdf`
- spring — `http://jboss.com/products/seam/spring`

Evènements, intercepteurs et gestion des exceptions

En complément du modèle composant contextuel, il y a deux autres concepts de base qui facilitent le couplage extrèmement faible qui est une fonctionnalité distincte des applications Seam. Le premier est un modèle d'évènement solide où les évènements peuvent être liés à des écouteurs d'évènements via des expressions reliant des méthodes ressemblant à JSF. Le second est une utilisation efficace des annotations et des intercepteurs pour appliquer des coupes/incisions dans les composants qui implémentent la logique métier.

6.1. Les évènements de Seam

Le modèle de composant de Seam a été développé pour l'utiliser avec les *applications conducteur-d-évènement*, en particulier disponibles pour développer des composants à grains fins et faiblement couplés dans un modèle évènementiel finement dimensionné. Les évènements dans Seam viennent de plusieurs types, la plus part ont déjà été vu:

- les évènements JSF
- les évènements de transition jBPM
- les actions de page de Seam
- les évènements conducteur-de-composant de Seam
- les évènements contextuel de Seam

Tous ces différents types d'évènements sont reliés à des composants Seam via des expressions relian une méthode JSF EL. Pour un évènement JSF, c'est défini dans un modèle JSF:

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

Pour un évènement de transition jBPM, il est spécifié dans la définition de processus jBPM ou dans une définition d'enchainement de page:

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

Vous pouvez découvrir beaucoup d'information à propos des évènements JSF et des évènements jBPM plus loin. Concentrons nous sur les deux autres types d'évènements additionnel défini par Seam.

6.2. Les actions de page

Une action de page de Seam est un évènement qui intervient juste avant que nous rendions la page. Nous déclarons les actions de pages dans `WEB-INF/pages.xml`. Nous pouvons définir une action de page à la fois comme un identifiant de vue JSF particulier:

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

Ou nous pouvons utiliser un jocker `*` comme suffixe à `view-id` pour spécifier une action qui s'applique à tous les identifiants de vue qui corresponde à ce patron:

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

Gardez à l'esprit que l'élément `<page>` est définie dans un descipteur de page à granularité fine, l'attribut `view-id` peux être laissé de côté avant d'avoir besoin de l'appliquer.

Si des pages d'action avec de multiples jockers correspondent à l'identifiant d'une vue, Seam va appeler touutes les actions dans l'ordre du moins spécifique au plus spécifique.

La méthode d'action de page retourne une sortie JSF. Si la sortie est non-nulle, Seam va l'utiliser pour définir les règles de navigation pour naviguer vers une vue.

Par la suite, l'identifiant de vue est mentionné dans l'élément `<page>` qui n'a pas besoin de correspondre à une vrai JSP ou une page Facelets! Donc, nous pouvons reproduire la fonctionnalité d'une serveur de squellette d'application orienté action comme Struts ou WebWork en utilisant des actions de page. Ceci est bien utile si vous voulez faire des choses complexes en réponse à des reqquêtes non-faces (par exemple, des requêtes HTTP GET).

Des actions de pages multiples ou conditionnelles peuvent être spécifié en utilisant le tag `<action>`:

```
<pages>
   <page view-id="/hello.jsp">
```

```
      <action execute="#{helloWorld.sayHello}" if="#{not validation.failed}"/>
      <action execute="#{hitCount.increment}"/>
   </page>
</pages
>
```

Les actions de page sont exécuté à la fois dans un requête initiale (non-faces) et dans une requête postérieure (faces). Si vous utilisez les actions de page pour charger des données, cette opération peut rentrer en conflit avec les action(s) standard(s) de JSF en cours d'éxécution dans la phase postérieure. Une façon de désactiver l'action de la page est de configurer une condition qui se résoud à vrai seulement avec une requête initiale.

```
<pages>
   <page view-id="/dashboard.xhtml">
      <action execute="#{dashboard.loadData}"
         if="#{not facesContext.renderKit.responseStateManager.isPostback(facesContext)}"/>
   </page>
</pages
>
```

Cette condition consulte le `ResponseStateManager#isPostback(FacesContext)` pour déterminer si la requête est une postérieure. Le ResponseStateManager est accédé en utilisant `FacesContext.getCurrentInstance().getRenderKit().getResponseStateManager()`.

Pour vous préserver de ce côté verbeux des API de JSF, Seam vous offre une condition livrée qui vous permet d'accomplir la même chose avec un gain en quantité à taper. Vous pouvez désactiver une action de page sur la phase postérieure simplement en définiant le `on-postback` à `false`:

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

Pour des raisons de compatiblité descendante, la valeur par défaut de l'attribut `on-postback` est vrai, pensez donc que vous allez finir par utiliser le réglage inverse de plus en plus souvent.

6.3. Les paramètres de page

Une requête faces JSF(une soumission de formulaire) est encapsulée à la fois dans une "action" (par relation de méthode) et des "paramètres" (par relation de valeur entrées). Une action de page peut aussi avoir besoin de paramètres!

Avec les requêtes GET qui sont en marque-page, les paramètres de page sont passés comme des paramètres de requêtes lisibles par un être humain. (A la différence des entrées de formulaire, qui sont tout autrement!)

Vous pouvez utiliser les paramètres de page avec ou sans une méthode d'action.

6.3.1. La liaison des paramètres de requêtes avec le modèle

Seam nous permet de fournir une valeur liée qui est en relation avec le paramètre de la requête d'un attribut de l'objet modèle.

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

La déclaration de `<param>` est bidirectionnel, tout comme la valeur liée à une entrée de JSF:

- Quand une requête non-faces (GET) pour l'identifiant de la vue intervient, Seam affecte la valeur du paramètre de la requête dénomée après avoir réaliser les conversions de types approriées.
- Tous les `<s:link>` ou `<s:button>` vont inclure le paramètre de la requête de manière transparentre. La valeur du paramètre est détérminée en évaluant la valeur liée pendant la phase de rendu (quand le `<s:link>` est rendu).
- Chaque règle de navigation avec un `<redirect/>` vers l'identifiant de la vue inclus de manière transparente le paramètre de la requête. La valeur du paramètre est déterminée en évaluant la valeur liée à la fin de la phase d'invocation de l'application.
- La valeur est de manière transparent propagée avec chaque soumission de formulaire JSF pour la page avec l'identifiant de vue donné. (Cela signifie que les paramètres de la vue fonctionne comme des variables de contexte d'étendue `PAGE` pour les requêtes de faces).

L'idée essentielle derrière tout cela est que depuis *n'importe* quelle page dont nous venions en allant vers `/hello.jsp` (ou depuis `/hello.jsp` vers `/hello.jsp`), la valeur de l'attribut du modèle est référencée dans la valeur liée qui est "mémorisée", sans avoir le besoin d'une conversation (ou d'un autre état côté serveur).

6.4. La propagation des paramètres de requêtes

Si seulement l'attribut `name` est spécifié alors le paramètre de requête est propagé en utilisant le contexte de `PAGE`(s'il n'est pas lié avec une propriété du model).

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

La propagation des paramètres de page est particulièrement utile si vous voulez construire des pages CRUD à multiples couches de type maitre-détails. Vous pouvez l'utiliser pour pour vous "souvenir" quelle vue vous etiez précédemment (par exemple en appuyant sur le bouton Sauver) et quelle entité vous etiez en train d'éditer.

- Tout `<s:link>` ou `<s:button>` propage de manière transparente le paramètre de requête si le paramètre est listé comme un paramètre de page de la vue.
- La valeur est propagée de manière transparente avec une soumission de formulaire JSF de la page avec l'identifiant de vue donné. (Ce qui signifie que les paramètres de vue fonctionne comme les variables de contexte d'étendue `PAGE` pour les requêtes faces).

Tout ce bazar à l'air assez comple, et vous allez probablement vous inquiéter d'une telle construction exotique vaut réellement l'effort. Pour l'instant, l'idée est très naturelle une fois que vous "l'avez". C'est définitivement beaucoup mieux de prendre son temps pour comprendre ce truc. Les paramètres de pages sont la façon la plus élégante de propager un état au travers de requêtes non-faces. C'est spécialement sympa pour les problèmes comme des écrans de recherches avec des pages de résultats stocké en favoris, quand vous voudrier être capable d'écrire le code de votre application pour gérer à la fois les requêtes GET et POST avec le même code. Les paramètres de page éliminent la vérification répétitive des paramètres de requêtes dans la définition de la vue et rend la redirection plus facile à coder.

6.5. La ré-écriture des URL avec les paramètres de page

La ré-écriture intervient se basant sur des patrons de ré-écriture trouvés pour les vues dans `pages.xml`. La ré-écriture d'URL de Seamfait à la fois de la ré-écriture entrante et sortante se bansant sur les mêmes patrons. Voici un patron simple:

```
<page view-id="/home.xhtml">
   <rewrite pattern="/home" />
</page>
```

Dans ce cas, une requête entrante pour `/home` sera envoyé vers `/home.xhtml`. Plus intéréssant, tout lien généré qui devrait normallement pointer vers `/home.seam` sera alors ré-écrit comme

`/home`. Les patrons de ré-écriture font correspondrent seulement la portion de l'URL avant les paramètres de requêtes. Donc, `/home.seam?conversationId=13` et `/home.seam?color=red` seront tout deux détecté par la règle de ré-écriture.

Les règles de ré-écritures peuvent prendre des paramètres de requêtes en considération, comme indiqué dans les règles suivantes.

```
<page view-id="/home.xhtml">
   <rewrite pattern="/home/{color}" />
   <rewrite pattern="/home" />
</page>
```

Dans ce cas, une requête entrante pour `/home/red` sera servie comme si elle était une requête pour `/home.seam?color=red`. De manière similaire, si la couleur est un paramètre de page d'une URL sortante qui devrait normallement se voir comme `/home.seam?color=blue` sera plutôt écrite comme `/home/blue`. Les règles sont exécutées dans l'odre, donc il est importante de les lister les règle les plus spécifiques avant les règles les plus généralistes.

Les paramètres par défaut de requêtes de Seam peuvent aussi être mis en correspondance en utilisant la ré-écriture d'URL, permettant d'avoir l'option de cacher les données techniques de Seam. Dans l'exemple suivant, `/search.seam?conversationId=13` sera ré-écrit comme `/search-13`.

```
<page view-id="/search.xhtml">
   <rewrite pattern="/search-{conversationId}" />
   <rewrite pattern="/search" />
</page>
```

La ré-écriture d'URL de Seam fourni une ré-écriture bi-directionnelle et simple basée pour chaque vue. Pour des règles de ré-écritures plus complexes qui couvrent des composants non-sean, les applications de Seam peuvent continuer à utiliser `org.tuckey URLRewriteFilter` ou appliquer des règles de ré-écritures dans le serveur web.

La ré-écriture d'URL nécéssite que le filtre de ré-écriture de Seam soit actif. La configuration du filtre de ré-écriture sera vue dans *Section 30.1.4.3, « URL rewriting »*.

6.6. La conversion et la validation

Vous pouvez indiquer un convertisseur JSF pour les propriétés du modèles complexes:

```
<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
>
```

Ou autrement:

```
<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
>
```

Les validateurs de JSF, et`required="true"` peuvent être utilisés:

```
<pages>
   <page view-id="/blog.xhtml">
      <param name="date"
           value="#{blog.date}"
           validatorId="com.my.blog.PastDate"
           required="true"/>
   </page>
</pages
>
```

Ou autrement:

```
<pages>
   <page view-id="/blog.xhtml">
      <param name="date"
           value="#{blog.date}"
           validator="#{pastDateValidator}"
           required="true"/>
```

```
  </page>
</pages
>
```

Même mieux, les annotations de validation de modèles basés sur Hibernate sont automatiquement reconnues et validées. Seam peut aussi fournir un convertisseur de date par défaut pour convertir une valeur de type chaine de caractère vers une date et dans l'autre sens.

Quand une conversion de type ou une validation échoue, un `FacesMessage` global est ajouté au `FacesContext`.

6.7. La navigation

Vous pouvez utiliser les règles de navigation JSF standard définies dans le `faces-config.xml` pour une application Seam. Malgré tout cela, les règles de navigation JSF ont de nombreuses limitations génantes:

- Il n'est pas possible de spécifier les paramètres de requêtes pouvant être utilisés dans la redirection.
- Il n'est pas possible de démarrer ou de finir des conversations depuis une règle.
- Les règles fonctionnent en évaluant le retour de valeur des méthode d'action; il n'est pas possible d'évaluer une expression EL arbitraire.

Un problème plus important est que la logique "d'orchestration" doit être éclaté entre `pages.xml` et `faces-config.xml`. C'est mieux d'unifier cette logique dans `pages.xml`.

Cette règle de navigation JSF:

```
<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>

</page
>
```

Mais c'est franchement mieux si nous pouvons éviter de poluer notre composant `DocumentEditor` avec une valeur retournée de type string (la sortie JSF). Ainsi Seam nous permet d'écrire:

```
<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>

</page
>
```

Ou même:

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

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

```
</page
>
```

La première forme évalue une valeur liée pour déterminer la valeur de la sortie qui doit être utilisée par les règles subséquentes. La seconde approche ignore la sortie et evalue une valeur liée pour chaque règle possible.

Bien sûr, quand une mise à jour réussie, nous voulons probablement finir la conversation courante. Nous pouvons faire cela comme ceci:

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

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

</page
>
```

En terminant la conversation toutes les requêtes soujacentes ne vont pas savoir quel document nous nous sommes intéréssé. Nous pouvons passer l'identifiant du document comme un paramètre de requête ce qui rend la vue disponible pour être mise en favoris:

```
<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>

</page
>
```

La sortie Null est un cas spécial en JSF. La sortie null est interprétée comme un "réaffiche la page". La règle de navigation suivante correspond à une sortie non-null, mais *pas* à la sortie null:

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

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

</page
>
```

Si vous voulez exécutez l'enchainement de page quand une sortie null intervient, utilisez plutôt la forme suivante:

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

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

</page
>
```

Avertissement

In case you are using JSF RI 2, you have to define navigation rule for each of the possible non-null outcome values from a page action, or else implicit navigation is going to render. It is annoying, hopefully will be fixed in the next maintenance version release of JSF 2.

L'identifiant de vue peut être donné comme une expression EL de JSF:

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

   <navigation>
      <rule if-outcome="success">
         <redirect view-id="/#{userAgent}/displayDocument.xhtml"/>
      </rule>
```

```
    </navigation>

</page
>
```

6.8. Les fichiers bien dimmensionnés pour la navigation, les actions de page et les paramêtres

Si vous avez beaucoup d'actions de page différentes et de paramêtres de pages, ou même juste beaucoup de règles de navigation, vous voudriez certainement diviser leurs déclarations dans plusieurs fichiers. Vous pouvez définir les actions et les paramêtres dans une page avec l'identifiant de vue `/calc/calculator.jsp` dans une ressource nommée `calc/calculator.page.xml`. L'élément root dans ce cas est un élément `<page>`, et l'identifiant la vue est tacite:

```
<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
>
```

6.9. Les évènements conducteur de composant

Les composants de Seam peuvent interagir en appelant simplement les méthodes des autres. Les composants avec état peuvent implémenter le patron de conception observeur/observable. Ainsi pour activer les composants à intéragir d'une façon faiblement couplée c'est possible quand le composant appele les méthodes des autres composants directement, Seam fournie *des évènement conducteur de composants*.

Vous pouvez spécifier un écouteur d'évènement (observateurs) dans `components.xml`.

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

Ici le *type d'évènement* est juste une chaine de caractères arbitraire.

Quand un évènement se produit, les actions enregistrées pour cet évènement vont être appelés dans l'ordre dans lequel ils apparaissent dans `components.xml`. Comment fait un composant pour déclencher un évènement ? Seam fourni un composant livré pour cela.

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

Or you can use an annotation.

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

Notez que ce producteur d'évènement n'a aucune dépendence vis-à-vis des consommateurs d'évènements. L'écouteur d'évènement devrait maintenant être implémenté sans absolument aucune dépendence avec le producteur:

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

La méthode liante défnie dans `components.xml` ci-dessus prend garde à la liaison de l'évènement vers le consommateur. Si vous n'aimez pas perdre du temps avec le fichier `components.xml`, vous pouvez à la place utiliser une annotation:

```
@Name("helloListener")
public class HelloListener {
   @Observer("hello")
```

```
    public void sayHelloBack() {
        FacesMessages.instance().add("Hello to you too!");
    }
}
```

Vous devez vous interroger pourquoi je n'ai rien mentionné à propos des objets évènements dans cette discution. Dans Seam, il n'y a pas besoin d'un objet évènement pour propager l'état entre un producteur d'évènement et un écouteur. L'état est contenu dans les contextes de Seam, et il est partagé entre les composants. Malgrès tout si vous voulez passer un objet évènement, vous pouvez:

```
@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);
    }
}
```

6.10. Les évènements contextuels

Seam définie un nombre d'évèvement livrés que l'application peut utiliser pour réaliser des trucs spéciaux de l'intégration du serveur d'application. Les évènements sont:

- `org.jboss.seam.validationFailed` — appelé quand la validation JSF échoue
- `org.jboss.seam.noConversation` — appelé quand il n'y a pas de conversation à éxécution longue et qu'une conversation à éxécution longue est requise
- `org.jboss.seam.preSetVariable.<name>` — appelé quand la variable de contexte <name> est définie

- `org.jboss.seam.postSetVariable.<name>` — appelé quand la variable de contexte <name> est définie
- `org.jboss.seam.preRemoveVariable.<name>` — appelé quand la variable de contexte <name> est non-définie
- `org.jboss.seam.postRemoveVariable.<name>` — appelé quand la variable de contexte <name> est non-définie
- `org.jboss.seam.preDestroyContext.<SCOPE>` — appelé avant que le contexte <SCOPE> ne soit détruit
- `org.jboss.seam.postDestroyContext.<SCOPE>` — appelé après que le contexte <SCOPE> est détruit
- `org.jboss.seam.beginConversation` — appelé à chaque fois qu'une conversation à éxécution longue commence
- `org.jboss.seam.endConversation` — appelé à chaque fois qu'une conversation à éxécution longue finie
- `org.jboss.seam.conversationTimeout` — appelé quand une conversation à éxécution longue se périme. L'identifiant de conversation est en paramètre.
- `org.jboss.seam.beginPageflow` — appelé quand un echainement de page commence
- `org.jboss.seam.beginPageflow.<name>` — appelé quand l'enchainement de page <name> commence
- `org.jboss.seam.endPageflow` — appelé quand un enchainement de page se termine
- `org.jboss.seam.endPageflow.<name>` — appelé quand l'enchainement de pages <name> se termine
- `org.jboss.seam.createProcess.<name>` — appelé quand le processus <name> est créé
- `org.jboss.seam.endProcess.<name>` — appelé quand le processus <name> se termine
- `org.jboss.seam.initProcess.<name>` — appelé quand le processus <name> est associé avec la conversation
- `org.jboss.seam.initTask.<name>` — appelé quand la tâche <name> est associée avec la conversation
- `org.jboss.seam.startTask.<name>` — appelé quand la tâche <name> est démarrée
- `org.jboss.seam.endTask.<name>` — appelé quand la tâche <name> est terminée
- `org.jboss.seam.postCreate.<name>` — appelée quand le composant <name> est créé

- `org.jboss.seam.preDestroy.<name>` — appelé quand le composant <name> est détruit
- `org.jboss.seam.beforePhase` — appelé avant le démarrage d'une phase JSF
- `org.jboss.seam.afterPhase` — appelé après la fin d'une phase JSF
- `org.jboss.seam.postInitialization` — appelé quand Seam est initialisé et démarre tous les composants
- `org.jboss.seam.postReInitialization` — appelé quand Seam a été ré-initialisé et démarre tous les composants après un redéploiement
- `org.jboss.seam.exceptionHandled.<type>` — appelé quand une exception non traitée est gérée par Seam
- `org.jboss.seam.exceptionHandled` — appelé quand une exception non traitée est gérée par Seam
- `org.jboss.seam.exceptionNotHandled` — appelé quand il n'y a pas de gestionnaire pour une exception non traitée
- `org.jboss.seam.afterTransactionSuccess` — appelée quand une transaction est un succès dans Seam Application Framework
- `org.jboss.seam.afterTransactionSuccess.<name>` — appelé quand une transaction est un succès dans Seam Application Framework gérant une entité appelée `<name>`
- `org.jboss.seam.security.loggedOut` — appelé quand un utilisateur se déconnecte
- `org.jboss.seam.security.loginFailed` — appelé quand l'authentification d'un utilisateur échoue
- `org.jboss.seam.security.loginSuccessful` — appelé quand un utilisateur réussit à s'authentifier
- `org.jboss.seam.security.notAuthorized` — appelé quand une vérification de l'authentification échoue
- `org.jboss.seam.security.notLoggedIn` — appelé quand il n'y a pas d'utilisateur authentifié et que l'authentification est requise
- `org.jboss.seam.security.postAuthenticate.` — appelé quand un utilisateur est authentifié
- `org.jboss.seam.security.preAuthenticate` — appelé avant d'essayer d'authentifier un utilisateur

Les composants de Seam peuvent observer chacun de ces évènements de la même manière qu'ils peuvent observer tous les évènements de conducteurs de composants.

6.11. Les intercepteurs de Seam

EJB 3.0 introduit un modèle d'intercepteur standard pour les composants bean de session. Pour ajouter un intercepteur à un bean, vous avez besoin d'écrire une classe avec une méthode annotée `@AroundInvoke` et annoter le bean avec une annotation `@Interceptors` qui spécifie le nom de classe d'interception. Par exemple, l'intercepteur suivant vérifie que l'utilisateur est connecté avant d'invoquer une méthode de l'écouteur d'action:

```
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";
    }
  }

}
```

Pour appliquer un intercepteur à un bean de session qui va agir comme un écouteur d'action, vous devez annoter le bean de session `@Interceptors(LoggedInInterceptor.class)`. C'est un peu annoté salement. Seam construit par dessus l'intercepteur du serveur d'application en EJB3 en vous autorisant à utiliser `@Interceptors` comme une méta-annotation pour les intercepteurs de niveau classe (ceux annoté `@Target(TYPE)`). Dans notre exemple, nous voudrions créer une annotation `@LoggedIn` , comme ci-dessous:

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

Nous pouvons maintenant simplement annoter notre bean d'écoute d'action avec `@LoggedIn` pour appliquer l'intercepteur.

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

   ...

   public String changePassword() { ... }

}
```

Si l'ordonnancement de l'intercepteur est important (habituellement c'est le cas), vous pouvez ajouter les annotations `@Interceptor` à vos classes d'intercepteur pour indiquer un ordre partiel des intercepteurs.

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

Vous pouvez même avoir un intercepteur "côté client", qui s'exécute avec toute la fonctionnalité livrée d'EJB3:

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

Les intercepteurs EJB sont avec état, avec un cycle de vie qui est le même que le composant qu'ils interceptent. Pour les intercepteurs qui n'ont pas besoin de maintenir un état, Seam vous permet d'avoir une optimisation de la performance en indiquant `@Interceptor(stateless=true)`.

La plus grande part de la fonctionnalité de Seam est implémentée comme un groupe d'intercepteurs livré par Seam, incluant les intercepteurs nommés dans l'exemple précédent. Vous

n'avez pas à spécifier explicitement ces intercepteurs en annotants vos composants; ils existent pour tous les composants Seam interceptables.

Vous pouvez même utiliser les intercepteurs de Seam avec les composants JavaBean components, pas seulement avec les beans EJB3 !

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).EJB défini l'interception pas seuelement pour les méthodes métier (en utilsiant `@AroundInvoke`)), mais aussi pour les méthodes du cycle de vie `@PostConstruct`,`@PreDestroy`, `@PrePassivate` et `@PostActive`. Seam supporte toutes les méthodes du cycle de vie sur à la fois les composants et les intercepteurs pas seulement pour les beans EJB3, mais aussi pour les composants JavaBean (exception avec `@PreDestroy` qui n'est pas significatif pour les composantsJavaBean).

6.12. La gestion des exceptions

JSF est étonnamant limité quand on vient sur la gestion des exceptions. Comme contournement partiel de ce problème, Seam vous permet de définir comment une classe particulière d'exception doit être traitée par l'annotation de la classe d'exception, ou en déclarant la classe d'exception dans un fichier XML. Cette facilité implique d'être combinée avec l'annotation standardisée EJB 3.0 `@ApplicationException` qui spécifie quand l'exception devrait entrainer une annulation de la transaction.

6.12.1. Les exceptions et les transactions

EJB spécifie des règles bien définies qui nous permette de contrôler quand une exception marque immédiatement la transaction courante pour l'annulation quand elle est déclenchée par une méthode métier du bean: *les exceptions systèmes* entrainent toujours une annulation de la transaction, *les exceptions d'application* n'entrainent par une annulation par défaut, mais elles le font si `@ApplicationException(rollback=true)` est spécifiée. (Une exception d'application est toujours une exception vérifiée, ou chaque exception non vérifiée est annotée par `@ApplicationException`. Une exception du système est toujours une exception non vérifiée sans une annotation `@ApplicationException`.)

Notez qu'il y a une différence entre marquer une transaction pour annulation, et réellement faire l'annulation. Les règles d'exceptions indiquent seulement que la transaction devrait être marquée pour annulation, mais elle peut toujours être active après que l'exception soit déclenchée.

Seam applique les règles d'annulation d'exception EJB 3.0 aussi que pour les composants JavaBean de Seam.

Mais ces règles s'appliquent seulement dans la couche composant de Seam. Que se passe t'il avec une exception qui n'est pas attrapé et qui se propage à l'extérieur de la couche composant de Seam, et à l'extérieur de la couche JSF ? Et bien, il est toujours mauvais de lever une transation

brainbalante ouverte, donc Seam rejoue en marche arrière chaque transaction active quand une exception apparait et ce n'est pas gérer dans la couche composant de Seam.

6.12.2. Activer la gestion d'exception de Seam

Pour activer la gestion d'exception de Seam, nous devons être sûr que nous avons un filtre de servlet maitre déclaré dans `web.xml`:

```
<filter>
   <filter-name
>Seam Filter</filter-name>
   <filter-class
>org.jboss.seam.servlet.SeamFilter</filter-class>
</filter>

<filter-mapping>
   <filter-name
>Seam Filter</filter-name>
   <url-pattern
>*.seam</url-pattern>
</filter-mapping
>
```

Vous devriez aussi avoir besoin de désactiver le mode de développement Facelets dans `web.xml` et le mode de déboguage de Seam dans `components.xml` si vous voulez que vos gestionnaire d'exception se déclenchent.

6.12.3. Utilisation des annotations pour la gestion d'exception

L'exception suivant résulte d'une erreur HTTP 404 qui à tout moment peut se propager à l'exterieur de la couche composant de Seam. Elle n'annule pas la transaction courante immédiatement à son déclenchement, mais la transaction va être annulé si l'exception n'est pas prise en compte par un autre composant de Seam.

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

Cette exception résulte dans une redirection du navigateur au moment où elle est propagée à l'extérieur de la couche composant de Seam. Elle finit aussi la conversation courante. Elle déclenche une annulation immédiate de la transaction courante.

```
@Redirect(viewId="/failure.xhtml", end=true)
```

```
@ApplicationException(rollback=true)
public class UnrecoverableApplicationException extends RuntimeException { ... }
```

Note

Il ets important de noter que Seam ne peut gérer les exceptions qui se déclenchent pendant la phase de JSF RENDER_RESPONSE et il n'est pas possible de réaliser une redirection tant que la response n'a pas commencé à être écrite.

Vous pouvez aussi utiliser EL pour indiquer le `viewId` à être rediriger vers.

Cette exception déclenche une redirection, avec un message pour l'utilisateur, quand il se propage hors de la couche composant de Seam. Cela aussi invalide immédiatement la transaction courante.

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

6.12.4. Utilisation d'XML pour la gestion d'exception

Comme nous ne pouvons pas ajouter les annotations pour toutes les classes d'exceptions auquelles nous nous intéressons, Seam nous permet aussi d'indiquer cette fonctionnalité dans `pages.xml`.

```
<pages>

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

  <exception class="javax.persistence.PersistenceException">
    <end-conversation/>
    <redirect view-id="/error.xhtml">
      <message
>Database access failed</message>
    </redirect>
  </exception>

  <exception>
    <end-conversation/>
    <redirect view-id="/error.xhtml">
      <message
```

```
>Unexpected failure</message>
      </redirect>
    </exception>

</pages
>
```

La dernière déclaration `<exception>` ne spécifie pas une classe, mais c'est un fourre-tout pour chaque exception pour laquelle une traitement n'est pas autrement spécifié via les annotation ou `pages.xml`.

Vous pouvez aussi utiliser EL pour indiquer le `view-id` à être rediriger vers.

Vous pouvez aussi accéder à l'instance de l'exception gérée au travers d'EL, Seam la place dans le contexte de conversation, par exemple pour accéder au message de l'exception:

```
...
throw new AuthorizationException("You are not allowed to do this!");

<pages>

    <exception class="org.jboss.seam.security.AuthorizationException">
        <end-conversation/>
        <redirect view-id="/error.xhtml">
            <message severity="WARN"
>#{org.jboss.seam.handledException.message}</message>
        </redirect>
    </exception>

</pages
>
```

`org.jboss.seam.handledException` conserve l'expcetion lié qui a été actuellement géré par un gestionnaire d'exception. L'exception la plus externe (emballé) est aussi disponible, comme une `org.jboss.seam.caughtException`.

6.12.4.1. Retirer la journalisation des exceptions

Pour le hestionnaire d'exception défini dans `pages.xml`, il ets possible de déclarer que le niveau de journalisation pour lequel l'exception sera enregistrée ou même pour supprimer complètement l'enregistrement de l'exception. Les attributs `log` et `log-level` peuvent être utilisé pour controler le niveau de journalisation des exceptions. En définissant `log="false"` comme dans l'exemple suivant, alors aucun message de journalisation ne sera généré quand l'exception spécifié se déclenchera:

```
   <exception class="org.jboss.seam.security.NotLoggedInException" log="false">
      <redirect view-id="/register.xhtml">
         <message severity="warn"
>You must be a member to use this feature</message>
      </redirect>
   </exception
>
```

Si l'attribut `log` n'est pas spécifié, alors par défaut c'est à `true` (i.e. l'exception sera enregistrée). Autre chose, vous pouvez indiquer le `log-level` pour controler quel niveau à parti duquel la journalisation des exceptions seront enregistrée:

```
   <exception class="org.jboss.seam.security.NotLoggedInException" log-level="info">
      <redirect view-id="/register.xhtml">
         <message severity="warn"
>You must be a member to use this feature</message>
      </redirect>
   </exception
>
```

Les valeurs acceptable de `log-level` sont: `fatal, error, warn, info, debug` ou `trace`. Si le `log-level` n'est pas spécifiée, ou si une valeur invalide est configurée, alors cela sera par défaut à `error`.

6.12.5. Quelques exceptions communes

Si vous utilisez JPA:

```
<exception class="javax.persistence.EntityNotFoundException">
  <redirect view-id="/error.xhtml">
    <message
>Not found</message>
  </redirect>
</exception>

<exception class="javax.persistence.OptimisticLockException">
  <end-conversation/>
  <redirect view-id="/error.xhtml">
    <message
>Another user changed the same data, please try again</message>
  </redirect>
```

```
</exception
>
```

Si vous utilisez le Seam Application Framework:

```
<exception class="org.jboss.seam.framework.EntityNotFoundException">
  <redirect view-id="/error.xhtml">
    <message
>Not found</message>
  </redirect>
</exception
>
```

Si vous utilisez Seam Security:

```
<exception class="org.jboss.seam.security.AuthorizationException">
  <redirect>
    <message
>You don't have permission to do this</message>
  </redirect>
</exception>

<exception class="org.jboss.seam.security.NotLoggedInException">
  <redirect view-id="/login.xhtml">
    <message
>Please log in first</message>
  </redirect>
</exception
>
```

et, pour JSF:

```
<exception class="javax.faces.application.ViewExpiredException">
  <redirect view-id="/error.xhtml">
    <message
>Your session has timed out, please try again</message>
  </redirect>
</exception
>
```

Une`ViewExpiredException` se déclenche, si l'utilisateur retourne vers une page une fois que sa session a expirée. Les réglages `conversation-required` et `no-conversation-view-id` dans le descripteur de la page de Seam, vue dans *Section 7.4, « Demander une conversation à exécution longue »*, vous donne un control finement dosé sur l'expiration de la session si vous accéder à une page utilisé dans une conversation.

Les conversations et le gestionnaire de l'espace de travail

Il est temps de comprendre le modèle conversationnel de manière plus détaillée.

Historiquement, la notion d'une "conversation" de Seam vient de la fusion de trois idées différentes:

- L'idée d'un *espace de travail*, que j'ai rencontré dans un projet pour le gouvernement Victorien en 2002. Dans ce projet, j'étais obligé d'implémenter un gestionnaire d'espace de travail par dessus Struts, une expérience que j'espère ne jamais plus répétée.
- L'idée d'une *transaction d'application* avec une sémantique optimiste, et que la réalisation d'un serveur d'application basé autour d'une architecture sans état ne peut fournir une gestion efficace des contextes de persistences étendues. (L'équipe Hibernate est réellement écoeurée d'être accusé pour toutes les *LazyInitializationExceptions*, ce qui n'est pas vraiment la faute d'Hibernate, mais plutôt la faute de la persistence extrèmement limitée du modèle de contexte supporté par les architectures sans état comme le serveur d'application Spring ou le traditionnel (anti)pattron *facade de session sans état* dans J2EE.)
- L'idée d'un enchainement de *tâches*.

En unifiant ces idées et en fournissant un profond support dans le squellete d'application, nous avons une construction percutante qui nous permets de construire de plus riche et de plus éfficace applications avec moins de code qu'auparavant.

7.1. Le modèle conversationnel de Seam

Les exemples que nous avons vu jusque là utilisent un modèle conversationnel très simple qui suit ces règles:

- l y a toujours un contexte conversationnel actif pendant l'application des valeurs de la requête, l'exécution des validations, la mise à jours des valeurs du modèles, l'invocation de l'application et le rendu des phases de responses du cycle de vie de la requête JSF.
- A la fin de la phase de restoration de la vue du cycle de vie de la requête JSF, Seam essaye de restorer tout contexte conversationnel à éxécution longue. Si aucun n'existe, Seam crée un nouveau contexte conversationnel temporaire.
- Quand une méthode `@Begin` est rencontrée, le contexte conversationnel temporaire est promu en conversation à exécution longue.
- Quand la méthode `@End` est rencontré , chaque contexte conversationnel à exécution longue est dégommé en conversation temporaire.

- 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.
- Chaque requête face (un postback JSF) va propager le contexte conversationnel. Par défaut, les requêtes non-faces (requêtes GET, par exemple), ne propage pas le contexte conversationnel, mais regardez ci-dessous pour plus d'information à propos de cela.
- Si le cycle de vie de la requête JSF est réduit par une redirection, Seam entreprose de manière transparente et restitue le contexte conversationnel courrant — à moins que la conversation n'est été déjà terminée via `@End(beforeRedirect=true)`.

Seam propage de manière transparente le contexte conversationnel au travers des postbacks JSF et des redirections. Si vous ne voulez pas faire quelquechose de spécial, une *requête non-faces* (une requête GET par exemple) ne va pas propager le contexte conversationnel et va être exécuté dans une nouvelle conversdation temporaire. C'est habituellement- mais pas toujours- le fonctionnement désiré.

Si vous voulez propager une conversation Seam au travers d'une requête non-faces, vous devez explicitement coder *l'identifiant de conversation* comme un paramètre de requête:

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

Ou beaucoup plus, JSF-ien:

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

Si vous utilisez la librairie de tag de Seam, ceci est équivalent:

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

Si vous voulez désactiver la propagation du contexte conversationnel pour un postback, un truc similaire est utilisé:

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

Si vous utilisez la librairie de tag de Seam, ceci est équivalent:

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

Notez que la désactivation de la propgation du contexte conversationnel n'est absolument pas la même chose que finir la conversation.

Le paramètre de requête `conversationPropagation` ou le tag `<s:conversationPropagation>` peut même être utilisé pour commencer ou finir la conversation liée.

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

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

Ce modèle conversationnel rend facile de construire des applications qui traite correctement en respectant les opération multi-fenètres. Pour beaucoup d'application, c'est tout ce qu'il y a besoin. Quelques applications complexes ont besoin d'une ou des deux exigeances additionnelles suivantes:

- Une conversation s'étend sur plusieurs petites unités d'intéraction de l'utilisateur, qui s'exécute en série ou même en concurence. Les plus petites *conversations liées* ont leur propre groupe d'état conversationnel isolé, et aussi ont accès à l'état des conversation externe.
- L'utilisateur est aussi capable de basculer entre plusieurs conversations dans la même fenètre du navigateur. Cette fonctionnalité est appelé le *gestionnaire d'espace de travail*.

7.2. Les conversations liées

Une conversation liée est créée en invoquant la métode marqué `@Begin(nested=true)` à l'intérieur d'une conversation existante. Une conversation liée a son propre contexte conversationnel et peut aussi lire les valeurs depuis le contexte conversationnel externe . Le contexte conversationnel externe est en lecture seul à l'intérieur d'une conversation liée mais parce que les objets sont obtenue par référence, les modifications des objets eux-même seront réfléchies dans le contexte externe.

- En liant au travers d'une conversation initialise un contexte qui sera empilé sur le contexte de la conversation orginelle ou externe. La conversation externe est considéré comme la parente.
- Toute valeurs extradée ou directement défini dans le contexte conversationnelle lié n'affectera pas l'accéssibilité des objets dans le contexte conversationnel parent.
- L'injection ou le parcours dans le contexte depuis le contexte conversaionnel cherchera en premier la valeur dans le contexte conversationnel courant et si aucune valeur n'est trouvée, descendra dans la pile de conversation si la conversation est liée. Comme vous le voyez pour l'instant, cette fonctionnalité peut être surchargée.

Quand un `@End` est par la suite rencontré, la conversation liée sera détruite et la conversation externe sera terminée, en la "retirant du dessus" de la pile des conversations. Les conversations peuvent être liée à n'importe quelle profondeur arbitraire.

Certaines activités de l'utilisateur (gestionnaire de l'espace de travail ou bouton précédent) peuvent faire en sorte que la conversation externe soit reprise avant que la conversation interne ne soit finie. Dans ce cas, il est possible d'avoir de multiples conversations concurrantes liées appartenant à la même conversation externe. Si la conversation externe se fini avant que la conversation liée ne s'arrête, Seam détruit tous les contextes conversationnels liés en relation avec le contexte externe.

La conversation au bas de la pil de conversation est la conversation racine. La destruction de cette conversation va toujours détruire les conversations enfantées. Vous pouvez réaliser ceci en spécifiant `@End(root=true)`.

Une conversation peut être imaginée comme un *état continue*. Les conversations liée permette à l'application de capture un état constistent et continue à plusieurs moments de l'interaction utilisateur, ainsi assurant de manière réellement correcte le focntionnement vis à vis du bouton précédent et de la gestion de l'espace de travail.

Comme indiqué précédemment, si un composant existe dans la conversation parente de la conversation liée courante, la conversation liée utilisera la même instance. Occasionnellement, il est utile d'avoir une instance différente pour chaque conversation liée, ainsi l'instance du composant qui existe dans la conversation parente est invisible à ses propres conversations filles. Vous pouvez avoir cette fonctionnalité en annotatant le composant `@PerNestedConversation`.

7.3. Démarrage des conversations avec les requêtes GET

JSF ne définie aucun type d'écouteur d'action qui est déclencher quand une page est accédée via une requête non-faces (par exemple, une requête HTTP GET). Cela peut arriver si l'utilisateur fait une favori de la page, ou si nous navigons vers la page via un `<h:outputLink>`.

Parfois nous voulons commencer une conversation immédiatement quand la page est accédée. Depuis qu'il n'y a pas de méthode d'action JSF, nous ne pouvons résoudre le problème de la manière habituelle, en annotation l'action avec`@Begin`.

Un autre problème est levé si la page a besoin des états soient reliés dans une variable de contexte. Nous avons déjà vue deux façon de résoudre ce problème. Si l'état est retenue par un composant Seam, nous pouvons relié l'éata dans une méthode `@Create` . Sinon, nous pouvons définir une méthode `@Factory` pour la variable de contexte.

Si aucune de ces options ne fonctionne pour vous, Seam vous permet de définir une action de page dans le fichier`pages.xml` file.

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

```
>
```

Cette méthode d'action est appelé au début de la phase de rendue de la réponse, à chaque fois que la page est au moment d'être rendue. Si une actiond e page retourne une sortie non-nulle, Seam va réaliser chaque règle de navigation JSF et Seam appropriée, résultant possiblement dans une page complètement différente à être rendue.

Si *tout* ce que vous voulez faire avant de rendre la page est de commencer une conversation, vous pouvez utiliser une méthode d'action livrée qui fait juste cela:

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

Notez que vous pouvez aussi appeler cette action livrée depuis un control JSF, et , de manière simmilaire, vous pouvez utiliser `#{conversation.end}` pour finir les conversations.

Si vous voulez plus de contrôle, pour rejoindre les conversations existantes ou commencer une conversation liée, pour commencer une enchainement de page ou une conversation insécable, vous devriez utiliser l'élément `<begin-conversation>`.

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

Il y aussi un élément `<end-conversation>`.

```
<pages>
   <page view-id="/home.jsp">
     <end-conversation/>
   <page>
   ...
</pages
>
```

Pour résoudre le premier problème, nous avons maintenant cinq options:

- Annoter la méthode `@Create` avec `@Begin`
- Annoter la méthode `@Factory` avec `@Begin`
- Annoter l'action de la page Seam avec `@Begin`
- Utiliser `<begin-conversation>` dans `pages.xml`.
- Utiliser `#{conversation.begin}` dans la méthode d'action de page de Seam

7.4. Demander une conversation à exécution longue

Certaines pages ne sont pertinente que dans le contexte d'une conversation à exécution longue. Une façon de "proteger" ce type de page est de demander une conversation à exécution longue comme un prérequis pour rendre la page. Heureusement, Seam a un mécanisme livré pour mettre en place ce prérequis.

Dans le descripteur de la page de Seam, vous pouvez indiquer que la conversation courante doit être à exécution longue (ou liée) pour que la page soit rendue en utilisant l'attribut `conversation-required` comme ci-dessous:

```
<page view-id="/book.xhtml" conversation-required="true"/>
```

Note

Le seul défaut est qu'il n'y à pas de façpon livré pour indiquer *quelle* conversation à exécution longue est requise. Vous pouvez construire ce mécanisme d'autorisation simple en à al fois vérifiant qu'une valeur spécifique est présente dans la conversation avec une action de page.

Quand Seam détermine que cette page est requise à l'extérieure d'une conversation à exécution longue, les actions suivantes interviennent:

- Un évènement contextuel dénommé `org.jboss.seam.noConversation` est levé
- Un message de status avertissement est enregistré en utilisant la clef fournie `org.jboss.seam.NoConversation`
- L'utilisateur est rediriger vers une page alternative, si définie

La page alternative est définie dans l'attribut `no-conversation-view-id` sur l'élément `<pages>` dans le descripteur de page de Seam comme ci-dessous:

```
<pages no-conversation-view-id="/main.xhtml"/>
```

A ce moment là, vous pouvez seulement définir un seul type de page pour toute l'application.

7.5. L'utilisation de `<s:link>` et de `<s:button>`

Les liens de commande JSF réalise toujours une soumission de formulaire via JavaScript, qui casse la fonctionnalité des navigateur web de "ouvrir une nouvelle fenêtre" ou "ouvrir un nouvel onglet". En pur JSF, vous avez besoin d'utiliser un `<h:outputLink>` si vous avez besoin de la fonctionnalité. Mais il y a deux limitations majeures à `<h:outputLink>`.

- JSF fourni une façon d'attacher un écouteur d'action à un `<h:outputLink>`.
- JSF ne propage pas la ligne sélectionné dans un `DataModel` s'il n'y a pas réellement de soumission de formulaire.

Seam fourni la notion d'*action de page* pour aider à résoudre le premier problème, mais cela ne fait rien pour le second problème. Nous *pourrions*contourner cela en utilisant l'apporche RESTful en passant le paramètre de requête et en requérant l'objet sélectionné du côté serveur. Dans quelques cas — comme l'application blog en exemple de Seam — c'est en fait la meilleure approche. Le style RESTful supporte le marquepage alors qu'il ne requiere par un état du côté serveur. Dans les autres cas, quand nous ne nous inquiétons pas à propos des marques-pages, l'utilisation de `@DataModel` et `@DataModelSelection` est juste plus pratique et plus transparant!

Pour implémenter cette fonctionnalité manquante et pour permettre la propagation de la conversation aussi simple pour gérer, Seam fourni la tag JSF `<s:link>`.

Le lien peut juste spécifier l'iodentifiant de la vue JSF:

```
<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"/>
```

Si vous spécifier les *deux* : un identifiant de vue JSF et une méthode d'action, la 'vue' sera utilisée *à moins* que la méthode d'action ne retourne un résultat non-null:

```
<s:link view="/loggedOut.xhtml"  action="#{login.logout}" value="Logout"/>
```

Le lien propage automatiquement la ligne sélectionnée d'un `DataModel` en l'utilisant avec `<h:dataTable>`:

```
<s:link view="/hotel.xhtml" action="#{hotelSearch.selectHotel}" value="#{hotel.name}"/>
```

Vous pouvez quitter l'étendue d'une conversation existante:

```
<s:link view="/main.xhtml" propagation="none"/>
```

Vous pouvez commencer, finir ou grouper les conversations:

```
<s:link action="#{issueEditor.viewComment}" propagation="nested"/>
```

Si le lien commence une conversation, vous pouvez même spécifier l'enchainement de page à utiliser:

```
<s:link action="#{documentEditor.getDocument}" propagation="begin"
      pageflow="EditDocument"/>
```

L'attribut `taskInstance` à utiliser dans une liste de tâches jBPM:

```
<s:link action="#{documentApproval.approveOrReject}" taskInstance="#{task}"/>
```

(Voir l'application de démonstration du magasin de DVD pour des exemples de cela.)

Enfin, si vous avez besoin d'un "lien" qui soit visualisé comme un bouton, utilisez `<s:button>`:

```
<s:button action="#{login.logout}" value="Logout"/>
```

7.6. Message de succès

Il est assez commun d'afficher un message pour l'utilisateur indiquant le succés ou l'echec d'une action. Il est pratique d'utiliser un `FacesMessage` de JSF pour cela. Malheureusement, une action réussie a besoin souvant d'une redirection du navigateur, et JSF ne peut pas propager un messages faces au travers de redirections. Cela rends les choses un peu plus difficile pour afficher un message de succés en pur JSF.

Le composant de Seam d'étendue conversation livré dénomé `facesMessages` résoud ce problème. (Vous devez avoir un filtre de redirection de Seam installé.)

```
@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");
   }
}
```

Chaque message ajouté à `facesMessages` est utilisé dans la phase de réponse immédiatement suivante pour la conversation courante. Cela fonctionne même quand il y une conversaiton à exécution longue depuis que Seam conserve les contexte de conversation temporaire au travers de redirections.

Vous pouvez même inclure les expressions EL de JSF dans le label du messages faces:

```
facesMessages.add("Document #{document.title} was updated");
```

Vous pouvez afficher le message de manière usuelle, par exemple:

```
<h:messages globalOnly="true"/>
```

7.7. Utilisation d'un identifiant de conversation "explicite"

Ordinairement avec les conversation qui travaille avec des objets persistants, il peut être utile d'utiliser une clef métier explicite au lieu de la version standard, identifiant de conversation de "substitution":

Redirection simple vers une conversation existante

Il peut être utile de rediriger vers une conversation existante si l'utilisateur demande la même opération deux fois. Prenons cet exemple: « Vous êtes sur ebay, au milieu de la phase de paiement pour un cadeau de noël pour vos parents que vous avez gagné. Inutile de dire que vous

voulez le leur envoyé immédiatement - vous entrez les défailts sur le paiement mais sans pouvoir vous souvenir de leur adresse. Vous réutilisez accidentellement la même fenètre du navigateur pour trouver leur adresse. Maintenant, vous avez besoin de retourner vers le paiement pour ce truc. »

Avec une conversation explicite, il est vraiment très facile d'avoir un utilisateur qui rejoind une conversation existante, et la reprendra là où il la laissé - tout simplement lui permettre de rejoindre la conversation payForItem avec l'itemId comme identifiant de conversation.

Utilisation des URLs sympatiques

Pour moi cela consiste dans une hierachie navigable (Je peux naviguer en éditant l'URL) et une URL pertinente (comme font les Wiki - donc sans identifier les choses avec des identifiants kabbalistiques). Pour des applications l'utilisation des URLs sympatiques ne sont pas , bien sûr, du tout utile.

Avec une conversation explicite, quand vous allez construire votre système de réservation d'hotel (ou, bien sûr, quelque soit votre application) vous pouvez générer une URL comme `http://seam-hotels/book.seam?hotel=BestWesternAntwerpen` (bien sûr, quelque soit le paramètre `hotel` correspondant au modèle de votre domaine doit être unique) et avec une ré-écriture des URLs facilement transformable en http://seam-hotels/book/BestWesternAntwerpen.

Encore mieux!

7.8. La création d'une conversation explicite

Les conversations explicites sont définies dans *description*

```
<conversation name="PlaceBid"
              parameter-name="auctionId"
              parameter-value="#{auction.auctionId}"/>
```

La première chose à noter de la définition ci-dessus est qua la conversation a un nom, dans ce cas `PlaceBid`. Ce nom identifiant de manière unique cette conversation particulière, et est utilisé par la définition de `page` pour identifier une conversation sus-nommé pour y participer.

L'attribut suivant, `parameter-name` définit le paramète de la requête qui va contenir l'identifiant explicite de conversation, en lieu et place du paramètre d'identifiant de conversation par défaut. Dans cet exemple, le `parameter-name` est `auctionId`. Ce qui signifie qu'au lieu d'avoir un paramètre de conversation comme `cid=123` qui apparait dans l'URL de votre page, il y aura `auctionId=765432` à la place.

Le dernier attribut dans la configuration ci-dessus, `parameter-value`, défini une expression EL utilisée pour évaluer la valeur de la clef métier explicite à utiliser comme identifiant de conversation. Dans cet exemple, l'identifiant de conversation sera une valeur de clef primaire de l'instance `auction` courante dans l'étendue.

Ensuite, nous définissons quelle page va être dans la convesation dénommée. Ceci est fait en spécifiant que l'attribut `conversation` pour une définition de `page`:

```
 <page view-id="/bid.xhtml" conversation="PlaceBid" login-required="true">
    <navigation from-action="#{bidAction.confirmBid}"
>
       <rule if-outcome="success">
          <redirect view-id="/auction.xhtml">
             <param name="id" value="#{bidAction.bid.auction.auctionId}"/>
          </redirect>
       </rule
>
    </navigation>
 </page
>
```

7.9. Redirection vers une conversation explicite

Au moment de démarrer, ou de rediriger vers, une conversation explicite, il y a plusieurs options pour spécifier le nom de la conversation explicite. Commençons pas regarder la définition de page suivante:

```
 <page view-id="/auction.xhtml">
  <param name="id" value="#{auctionDetail.selectedAuctionId}"/>

  <navigation from-action="#{bidAction.placeBid}">
   <redirect view-id="/bid.xhtml"/>
  </navigation>
 </page
>
```

A partir de là, nous pouvons voir que l'invocation de l'action `#{bidAction.placeBid}` pour notre vue auction (à partir de là, tous ces exemples sont prit de l'exemple seamBat dans Seam), qui va être redirigier vers `/bid.xhtml`, qui , comme nous l'avons vu précédemment, est configuré avec la conversation explicite `PlaceBid`. La déclaration de notre méthode action devrait ressemble à ceci:

```
  @Begin(join = true)
  public void placeBid()
```

Quand les conversations nommées sont spécifiées dans l'élément `<page/>`, la redirection vers la conversion nommé intervient comme une des règles de navigations, après que la méthode d'action ai été invoquée. Ceci est un problème quand la redirection vers une conversation existante, quand la redirection a besoin d'intervenir après que la méthode d'action ai été invoquée. C'est pourquoi il est nécéssaire de spécifier le nom de la conversation quand l'action est invoquée. Une façon de faire cela est en utilisant la balise `s:conversationName` :

```
<h:commandButton id="placeBidWithAmount" styleClass="placeBid"
action="#{bidAction.placeBid}">
  <s:conversationName value="PlaceBid"/>
 </h:commandButton
>
```

Une autre manière de spécifier l'attribut `conversationName` est en utilisant aussi bien `s:link` que `s:button`:

```
<s:link value="Place Bid" action="#{bidAction.placeBid}" conversationName="PlaceBid"/>
```

7.10. Le gestionnaire d'espace de travail

Le gestionnaire de l'espace de travail est capable de "basculer" les conversations dans une seule fenètre. Seam rend la gestion des espaces de travail complètement transparente au niveau du code Java. Pour activer le gestionnaire de l'espace de travail, tout ce que vous avez à faire est :

- Fournir un texte de *description*pour chaque identifiant de vue (avec l'utilisation de JSF ou des règles de navigation Seam) ou un noeud de page (avec l'utilisation des enchainements de page jPDL). Ce texte descriptif est afficher à l'utilisateur par le commutateur d'espace de travail.
- Inclure un ou plusieurs commutateur d'espace de travail standards JSP ou fragment de facelets dans vos pages. Les fragments standards supportent le gestionnaire d'espace de travail via un menu par liste sélectionnable, ou par fil d'ariane.

7.10.1. Le gestionaire d'espace de trvail et la navigation JSF

Quand vous utiliser les règles de navigation Seam ou JSF, Seam bascule d'une conversation en restorant le `view-id` courrant pour cette conversation. Le texte descriptif pour l'espace de travail est défini dans un fichier appelé `pages.xml` ainsi Seam s'attend à trouver dans le dossier `WEB-INF`, tout à côté de `faces-config.xml`:

```
<pages>
  <page view-id="/main.xhtml">
    <description
```

```
>Search hotels: #{hotelBooking.searchString}</description>
   </page>
   <page view-id="/hotel.xhtml">
      <description
>View hotel: #{hotel.name}</description>
   </page>
   <page view-id="/book.xhtml">
      <description
>Book hotel: #{hotel.name}</description>
   </page>
   <page view-id="/confirm.xhtml">
      <description
>Confirm: #{booking.description}</description>
   </page>
   </pages
>
```

Botez que si le fichier est manquant, l'application Seam continura de fonctionner parfaitement! La seule fonctionnalité manquante sera la capacité de basculer dans les espaces de travail.

7.10.2. Le gestionnaire d'espace de travail et l'enchainement de page jPDL

Quand vous utilisez une définition d'enchainement de page jPDL, Seam bascule vers une conversation en restorant l'état du processus jBPM courant. Ceci est beaucoup plus flexible depuis qu'il permet au même `view-id` d'avoir plusieurs descriptions selon le noeud `<page>` courrant. Le texte descriptif est défini dans le noeud `<page>`:

```
<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
>
```

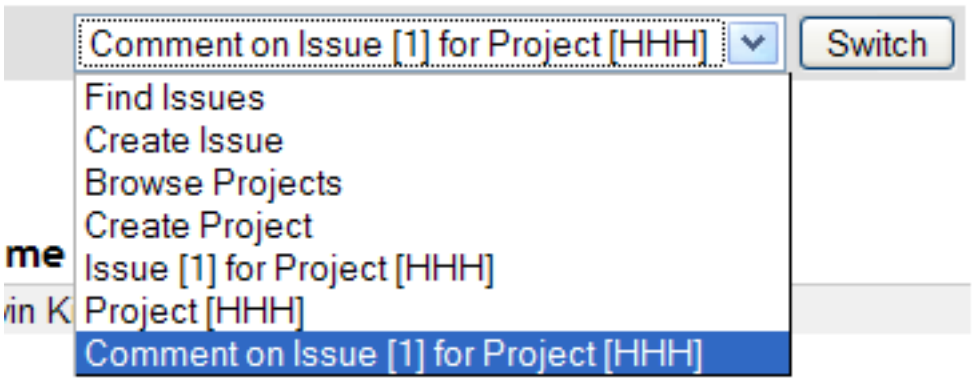
7.10.3. Le commutateur de conversation

Inclure les fragments suivant dans votre page facelets ou JSP pour avoir le menu à liste sélectionnable qui vous permet de basculer vers n'importe quelle conversation courante, ou vers n'importe quelle page de l'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"/>
```

Dans cet exemple, nous avons un menu qui inclus un élément pour chaque conversation, regroupé avec deux éléments additionnels qui permet à l'utilisateur de commencer une nouvelle conversation.

Seulement les conversations avec une description (spécifiée dans `pages.xml`) seront incluses dans le menu à liste sélectionnable.



7.10.4. La liste de conversation

La liste des conversations est vraiment similaire au commutateur de conversation, exception faite qu'elle est affiché comme un tableau:

```
<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
>
```

Nous imaginons que nous allons vouloir la personnaliser pour notre application.

Workspace	Workspace activity	Action
Comment on Issue [1] for Project [HHH]	01:18 PM - 01:18 PM	Switch Destroy
Issue [1] for Project [HHH]	01:18 PM - 01:18 PM	Switch Destroy
Project [HHH]	01:18 PM - 01:18 PM	Switch Destroy

Seule les conversations avec une description seront incluse dans la liste.

Il faut noter que la liste de conversation permet à l'utilisateur de détruire les espaces de travail.

7.10.5. Le fil d'ariane

Les fils d'ariane sont vraiment très utile dans les applications qui utilisent un modèle de conversations liée. Le fil d'ariane est une list de liens vers les conversations dans la pile de conversation courante:

```
<ui:repeat value="#{conversationStack}" var="entry">
   <h:outputText value=" | "/>
   <h:commandLink value="#{entry.description}" action="#{entry.select}"/>
</ui:repeat
```

7.11. Les composants conversationnels et la liaison avec les composants JSF

Les composants conversationnels ont une limitation mineure; ils ne peuvent être utilisé pour conserver une liaison vers des composants JSF. (Nous préférons générallement ne pas utiliser cette fonctionnalité avec JSF à moins que cela ne soit absolument nécssaire, car cela créer une dépendance forte de la logique applicative vers la vue). Avec une requête de retour,la liaison de composant est mise à jours pendant la phase de restoration de la vue, avant que le contexte conversationnel de Sean ne soit restoré.

Pour contourner ceci, utilisez un composant d'étendue évènement pour conserver la liaison du composant et l'injecter dans le composant d'étendue conversationnel qui en a besoin.

```
@Name("grid")
@Scope(ScopeType.EVENT)
public class Grid
{
   private HtmlPanelGrid htmlPanelGrid;

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

```
@Name("gridEditor")
@Scope(ScopeType.CONVERSATION)
```

```
public class GridEditor
{
   @In(required=false)
   private Grid grid;

   ...
}
```

De même, vous ne pouvez pas injecter un composant d'étendue conversationnel dans un composant d'étendue d'évènement que vous liez avec un contrôle JSF. Ceci inclu les composants livrés de Seam comme `facesMessages`.

L'alternative est que vous pouvez accéder à l'arbre de composant JSF au traver d'une référence `uiComponent` implicite. L'exemple suivant accède `getRowIndex()` du composant `UIData` qui conserve le tableau de données pendant l'itération, il affiche le numéro de ligne courrant:

```
<h:dataTable id="lineItemTable" var="lineItem" value="#{orderHome.lineItems}">
  <h:column>
    Row: #{uiComponent['lineItemTable'].rowIndex}
  </h:column>
  ...
</h:dataTable
>
```

Les composant UI de JSF sont disponibles avec leurs identifiants de client dans cette table de hachage.

7.12. Les appels concurentiels de composants conversationnels

Une discussion général d'appel concurentiel des composants de Seam peut être trouvée dans *Section 4.1.10, « Modèle de concurrence »*. Ici, nous allons discuter de la situation la plus courante où vous allez rencontrer de la concurence — l'accès aux composants conversationnel depuis les requêtes AJAX. Nous aloons discuter des options que les bibliothèque AJAX cliente devraient fournir pour controler les évènements provenant du client — et nous allons voir l'option que RichFaces vous offre.

Les composants conversationnels ne permettent pas un accès réellement concurentiel à moins que Seam ne mette en file d'attente chaque requête pour les réaliser séquenciellement. Ceci permet à chaque requête d'être exécutée de façon déterministe. Cependant une simple file d'attente n'est pas aussi géniale — en premier lieu, si uen méthode prend, pour une raison quelconque, un très long moment pour se terminer, exécuter encore une par dessus , et encore

une autre, tant et plus que le client génère une requête est une mauvaise idée (attaque potentielle par Dénie de Service) et en second, AJAX est souvent utilisé pour fournir une mise à jours rapide d'un status à l'utilisateur, donc continuer à exécuter l'action après un bout de temps n'est pas très utile.

Ainsi, quand vous travaillez à l'intérieur d'une conversation à exécution longue, Seam met en file d'attente les évènements d'action pour une période de temps (le temps de péremption de la requête concurentielle); si il peut exécuter l'évènement dans les temps, il créé une conversation temporaire et affiche le message pour que l'utilisateur sache ce qu'il se passe. Il est donc très important de ne pas saturer le serveur avec des évènements AJAX!

Nous pouvons définir par défaut et précisément le temps de péremption de la requête concurentielle (en ms) dans components.xml:

```
<core:manager concurrent-request-timeout="500" />
```

Nous pouvons aussi configurer finement le temps de péremption de la requête concurente sur une base de page-par-page:

```
<page view-id="/book.xhtml"
      conversation-required="true"
      login-required="true"
      concurrent-request-timeout="2000" />
```

D'aussi loin que nous discutons des requêtes AJAX qui apparaisent sérialisée à l'utilisateur - le client indique au serveur qu'un évènement intervient, et ensuite re-rends de la partie de la page basé sur le résultat. Cette approche est géniale quand la requête AJAX est légère (la méthode appelé est simple, par exemple, le calcul de la somme d'une colonne de nombre). Mais si nous avons besoin de faire un calcul complexe qui va prendre une minute?

Pour les calculs lourds, vous devrions utiliser une approche basé sur un groupement — le client envoie une requête AJAX vers le serveur, ce qui fait que l'action soit exécuté assynchronement sur le serveur (la response vers le client est immédiate) et le client ensuite regoupe les mises à jours vers le serveur. Ceci est une bonne approche si vous avez une action à exécution longue pour laquelle il est important que chaque action soit exécuté (vous ne voulez pas de péremption).

7.12.1. Comment allons nous construire notre application AJAX conversationnelle?

En premier lieu, vous avez besoin de choisir si vous voulez utiliser la requête la plus simple en "série" ou vous voulez utiliser une approche par regroupement.

SI vous vous décidez pour des requêtes en "séries", alors vous avez besoin d'estimer combien de temps va durer votre requête pour se terminer - est ce plus court que le temps de péremption de

la requête concurentielle? Ci ce n'est pas le cas, vous voulez probablement modifier le temps de péremption de la requête concurentielle (comme vu précédemment). Vous voulez probablement mettre en fil d'attente côté client pour prévenir la saturation du serveur avec les requêtes. Si l'évènement intervient souvent (par exemple, pression sur une touche, au survol de champs de saisie) et la mise à jours immédiate du client n'est pas une priorité, vous devriez définir un delais de la requête côté client. Quand le delais de la requête sera passé, tenir compte que l'évènement sera aussi mis en file d'attente du côté serveur.

Au final, la bibliothèque cliente peut fournir une option pour intérrompre une requête de duplication en cours au profit d'une plus récente.

L'utilisation d'un design de style regroupement nécéssite un réglage moins précis. Vous avez juste à indiquer votre méthode d'action avec `@Asynchronous` et décider d'un intervale de regroupement:

```
int total;

// This method is called when an event occurs on the client
// It takes a really long time to execute
@Asynchronous
public void calculateTotal() {
  total = someReallyComplicatedCalculation();
}

// This method is called as the result of the poll
// It's very quick to execute
public int getTotal() {
  return total;
}
```

7.12.2. La gestion des erreurs

Quelquesoit le cas, définissez votre application avec attention pour mettre les requêtes concurentes en file d'attente vers votre composant conversationnel, il ya une risque que le serveur devienne trop surchargé pour être capable d'exécuter toutes les requêtes avant que la requête n'ai à attendre plus longtemps que `concurrent-request-timeout`. Dans ce cas, Seam déclenchera un `ConcurrentRequestTimeoutException` qui peut être gérer dans `pages.xml`. Nous recommendons d'envoyer une erreur HTTP 503:

```
  <exception class="org.jboss.seam.ConcurrentRequestTimeoutException" log-level="trace">
    <http-error error-code="503" />
  </exception
>
```

503 Service Unavailable (HTTP/1.1 RFC)

Le serveur est habituellement incapable de gérer la requête à cause de la surcharge temporaire ou de la maintenance du serveur. L'implication est que ceci est une condition temporaire qui sera reduite après un délais.

Autrement vous pouvez rediriger vers une page d'erreur:

```
<exception class="org.jboss.seam.ConcurrentRequestTimeoutException" log-level="trace">
  <end-conversation/>
  <redirect view-id="/error.xhtml">
    <message
>The server is too busy to process your request, please try again later</message>
  </redirect>
</exception
>
```

ICEfaces, RichFaces Ajax et Seam Remoting peuvent gérer les codes d'erreurs HTTP. Seam Remoting va faire surgir une boite de dialogue montrant l'erreur HTTP. ICEfaces va indiquer l'erreur dans son composant de status de connection. RichFaces fourni le support le plus complet pour la gestion des erreurs HTTP en fournissant une fonction de rappel définissable pour l'utilisateur. Par exemple, pour afficher le message d'erreur pour l'utilisateur:

```
<script type="text/javascript">
  A4J.AJAX.onError = function(req,status,message) {
    alert("An error occurred");
  };
</script
>
```

Si au lieu d'un code d'erreur, le serveur rapporte que la vue a expirée, peut être à cause du temps de péremption de la session, vous pouvez utiliser une fonction de rappel séparée dans RichFaces pour gérer ce scénario.

```
<script type="text/javascript">
  A4J.AJAX.onExpired = function(loc,message) {
    alert("View expired");
  };
</script
```

```
>
```

Autre alternative, vous pouvez permettre à RichFaces de gérer l'erreur, dans ce cas l'utilisateur aura un message interogatif qui indiquera "View state could't be restored - reload page?" Vous pouvez personnaliser ce message globallement en définissant la clef du message suivant dans le fichier de ressource livré dans l'application.

```
AJAX_VIEW_EXPIRED=View expired. Please reload the page.
```

7.12.3. RichFaces (Ajax4jsf)

RichFaces (Ajax4jsf) est une bibliothèque Ajax la plus communément utilisée dans Seam, et fourni tous les controls discutés ci-dessous:

- `eventsQueue` — fourni une file d'attente où les évènements sont placés. Tous les évènements sont mis en attente et les requêtes sont envoyées vers le serveur en série. Ceci est utile si la requête vers le serveur peut prendre un certain temps à s'exécuter(par exemple. un calcul lourd, retrouver de l'information d'une source lente) si el serveur n'est pas saturé.
- `ignoreDupResponses` — ignore la réponse produite par la requête si une requête récente et 'similaire' est déjà dans la file d'attente. ignoreDupResponses="true" *ne peut annuler* l'exécution de la requête du côté serveur — simplement prévenir les mise à jours inutile du côté client.

  Cette option devrait être utilisé avec précaution avec les conversations de Seam qui permettent de faire des requêtes concurentielle multiples.
- `requestDelay` — défini le temps (en ms.) que la requête va rester dans la file d'attente. Si la requête n'est pas exécuté après ce delais, la requête sera envoyée (sans regarder si la réponse a été reçu) ou annuler (si il y a une requête similaire plus récente dans la file d'attente).

  Cette option devrait être utilisée avec précaution avec les conversations de Seam qui permettent de faire des requêtes concurentielles multiples. Vous devez être sur que le delais défini (en combinaison avec le temps de péremption de requête concurentielle) est plus long que l'action ne va prendre pour s'exécuter.
- `<a:poll reRender="total" interval="1000" />` — Regroupe le serveur et re-rends une zone nécéssaire

L'enchainement des pages et les processus métiers

JBoss jBPM est un moteur de gestion du processus métier pour tout environement Java SE ou EE. jBPM vous permet de vous représenter un processus métier ou une intéraction utilisateur comme un graphe de noeud représentant les états d'attente, les décisions, les tâches, les pages web, etc. Le graphe est défini en utilisant un simple et facile à lire dialecte XML appelé jPDL, et peut être édité et visualisé graphiquement en utilisant un plugin d'Eclipse. jPDL est un language extensible et il est adapté pour une grande variété de problème, depuis la définition de l'enchainement de page d'une application web jusqu'à la gestion des traditionnels enchainement de tâches, tout cela dans un esprit d'orchestration des services dans un environnement SOA.

Les applications Seam utilise jBPM pour deux types de problèmes différents:

- Définition d'un enchainement de page inclus dans de complexes intéractions de l'utilisateur. Une définition de processus jPDL définie l'enchainement de page pour une seule conversation. Une conversation Seam est considérée pour être une intérraction à relativement courte exécution avec un seul utilisateur.
- Définition d'un processus métier hypercintré. Le processus métier peu s'étendre sur de multiples conversations avec de multiples utilisateurs. Son état est persistant dans la base de données jBPM, donc il est considéré comme à longue exécution. La coordination des activités de multiples utilisateurs est un problème beaucoup plus complexe que l'écriture d'une intéraction avec un seul utilisateur, donc jBPM offre des fonctionnalités sophistiquées pour la gestion des tâches et la considération de multiples chemins d'éxécutions concurents.

Ne vous laisser pas embrouillés par ces deux choses ! Elle fonctionne à des niveaux ou de granularité très différents. *L'enchainement de page* , *de conversation* et des *tâches* toutes ce réfèrent à une seule intéraction avec un seul utilisateur. Un processus métier s'étends sur plusieurs tâches. En outre, les deux applications de jBPM sont totalement ortogonales. Vous pouvez les utiliser ensemble ou indépendamment ou pas du tout.

Vous n'avez pas besoin de connaitre jDPL pour utiliser Seam. Si vous êtes parfaitement heureux de définir l'enchainement de page en utilisant JSF ou les règles de navigation de Seam, et si votre application est plus à connatation données qu'à connotation processus, vous n'avez probablement pas besoin de jBPM. Mais nous allons trouver que penser l'interaction utilisateur en terme de représentation graphique bien définie nous aide à construire des applications plus robustes.

8.1. L'enchainement de page dans Seam

Il y a deux façon de définir un enchainement de page dans Seam:

- Utiliser les règles de navigation de JSF ou de Seam - le *modèle de navigation sans état*
- Utiliser le jPDL - le *modèle de navigation avec état*

De très simple applications n'ont seulement besoin que du modèle de navigation sans état. Des applications très complexes auront besoins des deux modèles à des endroits différents. Chaque modèle a ses forces et ses faiblesses!

8.1.1. Les deux modèles de navigation

Le modèle sans état défini une relation depuis un groupe de résultat nomé et la logique d'un évènement directement dans la page résultat de la vue. Les règles de navigation sont entièrement inconnues des autres états inclus dans l'application à part de la page qui a été la source de l'évènement. Cela signifie que les méthodes d'écoute de l'action doivent parfois prendre la décicsion à propos de l'enchainement de page, alors qu'elles n'ont accès qu'à l'état courrant de l'application.

Voici un exemple de définition d'un enchainement de page en utilisant les règles de navigation de JSF:

```
<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
```

```
>
```

Voici un exemple de définition d'un enchainement de page en utilisant les règles de navigation de Seam:

```
<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>

</page
>
```

Si vous trouvez que les règles de navigation beaucoup trop verbeuses, vous pouvez retourner l'identifiant de la vue directement depuis la méthode d'écouteur d'action:

```
public String guess() {
   if (guess==randomNumber) return "/win.jsp";
   if (++guessCount==maxGuesses) return "/lose.jsp";
   return null;
}
```

Notez que cela résulte dans une redirection. Vous pouvez même spécifier les paramètres à utiliser pour la redirection:

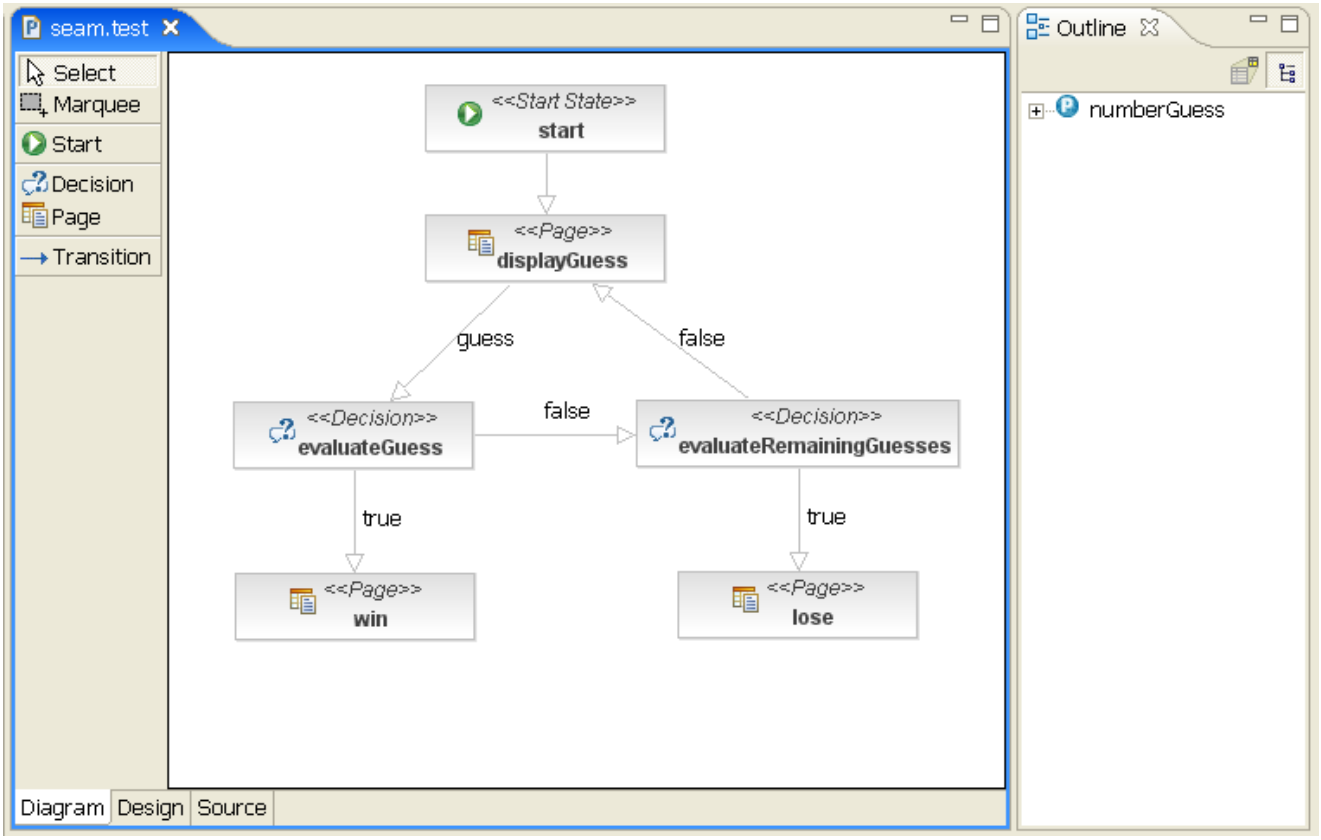
```
public String search() {
   return "/searchResults.jsp?searchPattern=#{searchAction.searchPattern}";
}
```

Le modèle avec état défini un groupe de transition entre un groupe d'état de l'application logique et nomé. Dans ce modèle, il est possible d'exprimer l'enchainement des interactions utilisateur

entièrement dans une définition d'enchainement de page en jPDL et écrire les méthode d'écouteur d'action qui sont complètement ignare du flot d'interaction.

Voici un exemple de définition du flot de page en utilisant 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
>
```



Il y a deux choses que nous pouvons noter immédiatement ici:

- Les règles de navigation JSF/Seam sont beaucoup *plus* simples. (Néanmoins, cela masque le fait que le code Java sous-entendu est plus complexe.)
- Le jPDL rend les interactions utilisateurs immédiatement compréhensible, sans qu'il soit même nécéssaire de lire le code JSP ou Java.

En plus, le modèle à état est plus *constraint*. Pour chaque état logique (chaque étape dans l'enchainement de page), il y a un contrainte sur un groupe de transition possible vers les autres états. Le modèle sans état est un modèle *ad hoc* qui est efficace pour la navigation relativement sans contrainte, libre de formulaire quand l'utilisateur décide là ou il/elle veux aller ensuite, pas l'application.

La distinction entre navigation avec/sans état est assez similaire à la traditionnel vue de l'intération avec/sans modèle. Maintenant, les applications de Seam ne sont pas en général modale au sens strict de ce mot - en fait, éviter la fonctionnalité modal de l'application est une des raisons principale pour avoir les conversations! Malgré tout, les applications Seam peuvent être, et souvent le sont, modale au niveau de conversation particulière. Il est bien connu que la fonctionnalité modale est quelquechose à éviter autant que possible; il est très difficile de prédire dans quel ordre vos utilisateurs vont vouloir faire les choses! Ainsi, il n'y a pas de doutes que le modèle avec état a sa place.

Le plus grand contraste entre les deux modèles est la fonction du bouton-précédent.

8.1.2. Seam et le bouton précédent

Quand les règles de navigation JSF et Seam sont utilisées, Seam laisse l'utilisateur naviger librement avec les boutons précédent, suivant et rafraichir. Il est de la responsabilité de l'application de s'assurer que l'état conversationnel reste en interne consistant quand cela arrive. L'experience avec la combinaisond'application web comme Struts ou WebWork - qui ne supporte pas le modèle conversationnel - et les modèles de composants sans état comme les beans de session sans état EJB ou le serveur d'application Spring a appris à beaucoup de développers qu'il est presque impossible de le faire! Ainsi, notre expérience est que c'est dans le contexte de Seam qu'il y a un modèle conversaionnel bien définie, soutenu par les beans de de sessions avec état, il est actuellement presque prêt. Habituellement, il est presque aussi simple de combiner l'utilisation `no-conversation-view-id` avec la vérification de null au démarrage des méthodes d'écouteur d'action. Nous considerons le support de la navigation libre de formulaire presque aussi souhaitable.

Dans ce cas, la déclaration de `no-conversation-view-id` va dans `pages.xml`. Il indique a Seam de rediriger vers une page différente si une requête originaire d'une page rendue pendant une conversation et que cette conversation n'existe plus:

```
<page view-id="/checkout.xhtml"
      no-conversation-view-id="/main.xhtml"/>
```

D'un autre côté, dans le modèle avec état, le bouton "précédent" est interprété comme un retour de transition non-définie vers l'état précédent. Depuis que le modèle avec état se renforce comme un groupe définie de transition depuis l'état courrant, le bouton "précédent" est par défaut interdit dans le modèle avec état! Seam de manière transparent détecte l'utilisation du bouton précédent et bloque toute tentative de réaliser une action depuis une page précédente "périmée" et redirige simplement l'utilisateur vers la page "courante" (et affiche un message faces). Que vous considériez comme une fonctionnalité ou une limitation le modèle sans état cela dépend de votre point de vue: comme un développeur d'application, c'est une fonctionnalité, comme un utilisateur, cela peut être frustrant. Vous pouvez activer la navigation avec le bouton "précédent" depuis un noeud de page particulier en indiquant `back="enabled"`.

```
<page name="checkout"
      view-id="/checkout.xhtml"
      back="enabled">
   <redirect/>
   <transition to="checkout"/>
   <transition name="complete" to="complete"/>
</page
>
```

Cela autorise le bouton "précédent" *depuis* l'état checkout vers *tout état précédent!*

Note

Si une page est définie pour rediriger après une transition, il n'est pas possible d'utiliser le bouton précédent pour retourner à la page même quand le bouton précédent est sur une page plus tard dans l'enchainement. La raison est que parce que Seam stocke les informations à propos de l'enchainement de page dans l'étendue page donc le bouton précédent doit résulter dans une requête POST pour que l'information soit restorée (autrement dit, une requête Faces). Une redirection fournie ce type de liaison.

Bien sûr, nous avons toujours besoin de définir ce qu'il arrive si une requête originiaire d'une page rendue pendant l'enchainement de page, et la conversation ave l'enchainement de page n'existe plus. Dans ce cas, la déclaration `no-conversation-view-id` va dans la définition d'enchainement de page:

```
<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
>
```

En pratique, les deux modèles de navigation ont leur spécificité et vous allez rapidement apprendre à les reconnaitre quand vous préférerez un modèle plutôt que l'autre.

8.2. Utilisation des enchainements de page jPDL

8.2.1. Installation des enchainements de page

Nous avons besoin d'installer les composants jBPM-style de Seam et leurs dirent où ils vont trouver la définition de l'enchainement de page. Nous pouvons spécifier cette configuration de Seam dans components.xml.

```
<bpm:jbpm />
```

Vous pouvez aussi explicitement indiquer à Seam où trouver la définition d'enchainement de page. Nous spécifions ceci dans `components.xml`:

```
<bpm:jbpm>
   <bpm:pageflow-definitions>
      <value
>pageflow.jpdl.xml</value>
   </bpm:pageflow-definitions>
</bpm:jbpm
>
```

8.2.2. Démarrage des enchainements de pages

Nous "démarrons" un enchainement de pages jPDL-style en indiquant le nom de la définition du processus en utilisation une annotation `@Begin`, `@BeginTask` ou `@StartTask`:

```
@Begin(pageflow="numberguess")
public void begin() { ... }
```

De manière alternative, nous pouvons démarrer un enchainement de page en utilisant pages.xml:

```
<page>
      <begin-conversation pageflow="numberguess"/>
   </page
>
```

Si nous commençons un enchainement de page pendant la phase `RRENDER_RESPONSE` — pendant qu'une méthode `@Factory` ou `@Create`, par exemple — nous considerons nous même être à la page qui doit être rendue, et utilisons un noeud `<start-page>` comme premier noeud dans l'enchainement de page, comme le montre l'exemple ci dessous.

Mais si l'enchainement de page a commencé comme le resulteat de l'invocation d'un écouteur d'action, le retour de lécouteur d'action détermine quel est la première page à être rendue. Dans ca cas, nous utilisons un `<start-state>` comme premier noeud de l'enchainement de page et déclarons une transition pour chaque sortie possible:

```
<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
>
```

8.2.3. Les noeuds de page et transitions

Chaque noeud `<page>` représente un état où le système est en train d'attendre une saisie de l'utilisateur:

```
<page name="displayGuess" view-id="/numberGuess.jsp">
    <redirect/>
    <transition name="guess" to="evaluateGuess">
        <action expression="#{numberGuess.guess}" />
    </transition>
</page
>
```

Le `view-id` est l'identifiant de la vue JSF. L'élément `<redirect/>` a le même effet que `<redirect/>` dans une règle de navigation : à savoir, une fonctionnalité poster-puis-rediriger, pour contourner les problème avec le bouton rafraichir du navigateur. (Notez que Seam propage les contextes de conversation au travers de ces redirection de navigateur. Il n'y a donc pas besoin d'un fabrique de style Ruby on Rails "flash" dans Seam!)

Le nom de transition est le nom de la sortie JSF déclenché en cliquant sur le bouton de commande ou le lien de commande dans `numberGuess.jsp`.

```
<h:commandButton type="submit" value="Guess" action="guess"/>
```

Quand la transition est déclenchée par le clic sur le bouton, jBPM ira activer l'action de transition en appelant la méthode *guess()* du composant *numberGuess*. Notez que la syntaxe utilisée pour spécifier les actions dans le jPDL est juste une expression familière JSF EL, et que l'action de transation est juste une méthode d'une composant Seam dans le contexte courrant de Seam.

Donc, nous avons exactement le même modèle d'évènement pour jBPM que nous avons déjà pour les évènements JSF! (Le principe *d'Un Seul Genre de Truc.*)

Dans le cas d'un résultat null (par exemple, un bouton de command sans aucune action définie), Seam ira signaler la transition sans aucun nom si une existe, ou sinon simplement réaffiche la page si toutes les transition ont un nom. Donc nous pourrions légèrement simplifier notre exemple d'enchainement de page et ce bouton:

```
<h:commandButton type="submit" value="Guess"/>
```

Qui va déclencher la transition sans-nom suivante:

```
<page name="displayGuess" view-id="/numberGuess.jsp">
   <redirect/>
   <transition to="evaluateGuess">
      <action expression="#{numberGuess.guess}" />
   </transition>
</page
>
```

Il est même possible d'avoir un bouton qui appele une méthode d'action, dans ce cas le résultat de l'action va déterminer la tranistion qui doit être prise:

```
<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
>
```

Malgré cela, il est est à considérer comme un style inférieur, car il déplace la responsabilité du contrôl du flot à l'extérieur de la définition de l'enchainement de page et le place dans un autre composants. Il est franchement meilleur de centraliser ce qui s'y rapporte dans l'enchainement de pages lui même.

8.2.4. Controler le flot

Habituellement, nous n'avons pas besoin des fonctionnalités les plus puissantes de jPDL pendant la définition des enchainements de pages. Nous avons besoin du noeud `<decision>`, malgré tout:

```
<decision name="evaluateGuess" expression="#{numberGuess.correctGuess}">
   <transition name="true" to="win"/>
   <transition name="false" to="evaluateRemainingGuesses"/>
</decision
>
```

Une décision est faite en évaluant une expression JSF EL dans les contextes de Seam.

8.2.5. Terminer le flot

Nous finissons la conversation en utilisant `<end-conversation>` ou `@End`. (Dans les faits, pour la lisibilité, l'utilisation des deux est encouragé.)

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

De manière optionnel, nous pouvons finir une tâche, spécifiant un nom de `transition` jBPM. Dans ce cas, Seam va signaler la fin de la tâche courrante dans le processus métier hypercintré.

```
<page name="win" view-id="/win.jsp">
   <redirect/>
   <end-task transition="success"/>
</page
>
```

8.2.6. La composition de l'enchainement de page

Il ets possible de composer des enchainements de page et d'avoir une pause dans l'enchainement de page alors qu'un autre enchainement s'exécute. Le noeud `<process-state>` met en pausse l'enchainement de page indiqué et comme l'exécution de l'enchainement de page appelé:

```
<process-state name="cheat">
   <sub-process name="cheat"/>
   <transition to="displayGuess"/>
</process-state
>
```

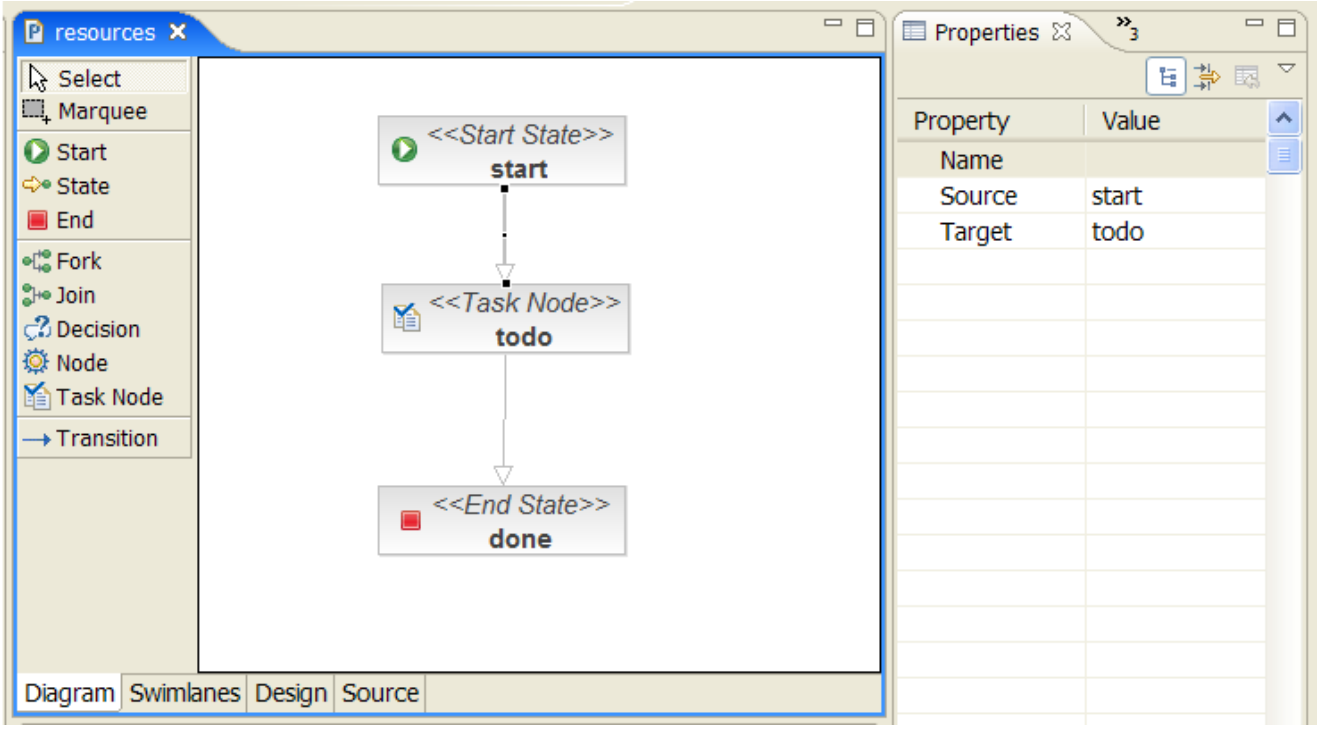
Le flot inclus commence en exécutant un noeud `<start-state>`. Quand il atteind un noeud `<end-state>`, l'exécution du flot inclus s'arrête et l'exécution du flot englobant recommence avec la transition définie dans l'élement `<process-state>`.

8.3. La gestion des processus métier dans Seam

Un processus métier est un groupe de tâche bien définis qui doivent être réaliser par des utilisateurs ou des systèmes logiciels en accord avec des règles bien définies à propos de *qui* peut réaliser une tâche, et *quand* elle devrait être réalisée. L'intégration de JBPM dans Seam rend facile d'afficher la liste des tâches des utilisateurs et les laisser gérer leurs tâches. Seam permet aussi à l'application stocker l'état associé avec le processus métier dans le contexte `BUSINESS_PROCESS` et avoir cet état qui soit persistant via les variables du jBPM.

Une simple définition de processus métier ressemble assez à la définition de l'enchainement de page (*Un Seul Genre de Truc*), exception sauf qu'au lieu d'un noeud `<page>` , nous avons des noeuds `<task-node>`. Dans un procéssus métier à exécution longue, les état d'atente sont là où le système est en attente qu'un utilisateur se connecte pour réaliser une tâche.

```
<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
>
```



Il est parfaitement possible que nous ayons à la fois des définition de processus métier en jPDL et des définitions d'enchainements de pages en jPDL dans le même projet. Et donc, la relation entre les deux est qu'un seul `<task>` dans un processus métier correspond à l'ensemble d'un enchainement de page `<pageflow-definition>`

8.4. Utilisation des définitions de processus métier jPDL

8.4.1. Installation des définition de processus

Nous avons besoin d'installer un jBPM, et de lui dire où trouver les définition de processus métier :

```
<bpm:jbpm>
   <bpm:process-definitions>
      <value
>todo.jpdl.xml</value>
   </bpm:process-definitions>
</bpm:jbpm
>
```

Les processus jBPM sont persistant au travers du redémarrage de l'application, avec l'utilisation de Seam dans un environement de production vous ne voudriez pas avoir à installer la définition de processus à chaque fois que l'application démarre. Cependant, dans un environement de production vous allez avoir besoin de détruire le processus jBPM à l'extérieur de Seam. En

d'autres termes, installez seulement le processus de définitions dans `components.xml` pendant le développement de votre application.

8.4.2. Initialisation des identifiants des acteurs

Nous avons toujours besoin de connaitre quel utilisateur est actuellement connecté. jBPM "connait" les utilisateur par *leur identifiant d'acteur* et par *leurs identifiants de groupe d'acteur*. Nous spécifions les identifiant d'acteur courrant en utilisant le composant livré dans Seam nommé `actor`:

```
@In Actor actor;

public String login() {
   ...
   actor.setId( user.getUserName() );
   actor.getGroupActorIds().addAll( user.getGroupNames() );
   ...
}
```

8.4.3. Initialisation d'un processus métier

Pour initialiser une instance d'un processus métier, nous utilisons l'annotation `@CreateProcess`:

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

Autre alternative, nous pouvons initialiser un processus métier en utilisant pages.xml:

```
<page>
   <create-process definition="todo" />
</page
>
```

8.4.4. Assigner une tâche

Quand un processus démarre, les instances de tâches sont créés. Ils doivent être assignés aux utilisateurs et aux groupes d'utilisateurs. Nous pouvons coder en durs les idenfiants d'acteur ou le déléguer à un composant de Seam:

```
<task name="todo" description="#{todoList.description}">
```

```
   <assignment actor-id="#{actor.id}"/>
</task
>
```

Dans ce cas, nous avons simplement assigné la tâche à l'utilisateur courrant. Nous pouvons aussi assigné les tâches à un groupement:

```
<task name="todo" description="#{todoList.description}">
   <assignment pooled-actors="employees"/>
</task
>
```

8.4.5. Liste des tâches

Plusieurs composant livrés dans Seam rendent facile l'affichage de listes de tâches. Le `pooledTaskInstanceList` est une liste de tâches en groupement que les utilisateurs peuvent assigner à eux-même:

```
<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
>
```

Notez qu'au lieu de `<s:link>` nous pouvons utiliser un `<h:commandLink>` en pur JSF :

```
<h:commandLink action="#{pooledTask.assignToCurrentActor}"
>
   <f:param name="taskId" value="#{task.id}"/>
</h:commandLink
>
```

Le composant `pooledTask` est un composant livré qui assigne simplement la tâche à l'utilisateur courrant.

Le composant `taskInstanceListForType` inclus les tâches d'un type particulier qui sont assigné à l'utilisateur courrant:

```
<h:dataTable value="#{taskInstanceListForType['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
>
```

8.4.6. Réalisation d'une tâche

Pour commencer le travail sur une tâche, nous utilisons aussi bien `@StartTask` ou `@BeginTask` sur la méthode d'écoute:

```
@StartTask
public String start() { ... }
```

Autre alternative, nous pouvons commencer le travail sur une tâche en utilisant pages.xml:

```
<page>
   <start-task />
</page
>
```

Ces annotations commencent une type spécial de conversation qui ont de l'importance en terme de processus métier hypercintré. Le travail fait par cette conversation à accès à l'état contenue dans le processus métier dans le contexte de processus métier.

Si nous finisons la conversation en utilisant `@EndTask`, Seam va signaler la réalisation de la tâche:

```
@EndTask(transition="completed")
```

```
public String completed() { ... }
```

Autre alternative, nous pouvons utiliser pages.xml:

```
<page>
   <end-task transition="completed" />
</page
>
```

Vous pouvez aussi utiliser une EL pour spécifier la transition dans pages.xml.

A ce moment, jBPM prends le contrôle et continue l'exécution de la définition du processus métier. (Dans des processus plus complexes, plusieurs tâches peuvent avoir besoin d'être réalisés avant que l'exécution du processus ne puisse reprendre.)

Merci de vous référer à la document de jBPM pour un apperçu plus détaillé des fonctionnalités sophistiquées que le jBPM fournir pour la gestion des processus métier complexes.

Seam and Object/Relational Mapping

Seam provides extensive support for the two most popular persistence architectures for Java: Hibernate3, and the Java Persistence API introduced with EJB 3.0. Seam's unique state-management architecture allows the most sophisticated ORM integration of any web application framework.

9.1. Introduction

Seam grew out of the frustration of the Hibernate team with the statelessness typical of the previous generation of Java application architectures. The state management architecture of Seam was originally designed to solve problems relating to persistence — in particular problems associated with *optimistic transaction processing*. Scalable online applications always use optimistic transactions. An atomic (database/JTA) level transaction should not span a user interaction unless the application is designed to support only a very small number of concurrent clients. But almost all interesting work involves first displaying data to a user, and then, slightly later, updating the same data. So Hibernate was designed to support the idea of a persistence context which spanned an optimistic transaction.

Unfortunately, the so-called "stateless" architectures that preceded Seam and EJB 3.0 had no construct for representing an optimistic transaction. So, instead, these architectures provided persistence contexts scoped to the atomic transaction. Of course, this resulted in many problems for users, and is the cause of the number one user complaint about Hibernate: the dreaded `LazyInitializationException`. What we need is a construct for representing an optimistic transaction in the application tier.

EJB 3.0 recognizes this problem, and introduces the idea of a stateful component (a stateful session bean) with an *extended persistence context* scoped to the lifetime of the component. This is a partial solution to the problem (and is a useful construct in and of itself) however there are two problems:

- The lifecycle of the stateful session bean must be managed manually via code in the web tier (it turns out that this is a subtle problem and much more difficult in practice than it sounds).
- Propagation of the persistence context between stateful components in the same optimistic transaction is possible, but tricky.

Seam solves the first problem by providing conversations, and stateful session bean components scoped to the conversation. (Most conversations actually represent optimistic transactions in the data layer.) This is sufficient for many simple applications (such as the Seam booking demo) where persistence context propagation is not needed. For more complex applications, with many loosly-interacting components in each conversation, propagation of the persistence context across components becomes an important issue. So Seam extends the persistence context management model of EJB 3.0, to provide conversation-scoped extended persistence contexts.

9.2. 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 some Seam applications.

However, there is a problem with this approach. A Seam application may not perform all data access for a request from a single method call to a session bean.

- The request might require processing by several loosely-coupled components, each of which is called independently from the web layer. It is common to see several or even many calls per request from the web layer to EJB components in Seam.
- Rendering of the view might require lazy fetching of associations.

The more transactions per request, the more likely we are to encounter atomicity and isolation problems when our application is processing many concurrent requests. Certainly, all write operations should occur in the same transaction!

Hibernate users developed the *"open session in view"* pattern to work around this problem. In the Hibernate community, "open session in view" was historically even more important because frameworks like Spring use transaction-scoped persistence contexts. So rendering the view would cause `LazyInitializationException`s when unfetched associations were accessed.

This pattern is usually implemented as a single transaction which spans the entire request. There are several problems with this implementation, the most serious being that we can never be sure that a transaction is successful until we commit it — but by the time the "open session in view" transaction is committed, the view is fully rendered, and the rendered response may already have been flushed to the client. How can we notify the user that their transaction was unsuccessful?

Seam solves both the transaction isolation problem and the association fetching problem, while working around the problems with "open session in view". The solution comes in two parts:

- use an extended persistence context that is scoped to the conversation, instead of to the transaction
- use two transactions per request; the first spans the beginning of the restore view phase (some transaction managers begin the transaction later at the beginning of the apply request vaues phase) until the end of the invoke application phase; the second spans the render response phase

In the next section, we'll tell you how to set up a conversation-scope persistence context. But first we need to tell you how to enable Seam transaction management. Note that you can use conversation-scoped persistence contexts without Seam transaction management, and there are good reasons to use Seam transaction management even when you're not using Seam-managed

persistence contexts. However, the two facilities were designed to work together, and work best when used together.

Seam transaction management is useful even if you're using EJB 3.0 container-managed persistence contexts. But it is especially useful if you use Seam outside a Java EE 5 environment, or in any other case where you would use a Seam-managed persistence context.

9.2.1. Disabling Seam-managed transactions

Seam transaction management is enabled by default for all JSF requests. If you want to *disable* this feature, you can do it in `components.xml`:

```
<core:init transaction-management-enabled="false"/>

<transaction:no-transaction />
```

9.2.2. Configuring a Seam transaction manager

Seam provides a transaction management abstraction for beginning, committing, rolling back, and synchronizing with a transaction. By default Seam uses a JTA transaction component that integrates with Container Managed and programmatic EJB transactions. If you are working in a Java EE 5 environment, you should install the EJB synchronization component in `components.xml`:

```
<transaction:ejb-transaction />
```

However, if you are working in a non EE 5 container, Seam will try auto detect the transaction synchronization mechanism to use. However, if Seam is unable to detect the correct transaction synchronization to use, you may find you need configure one of the following:

- JPA RESOURCE_LOCAL transactions with the `javax.persistence.EntityTransaction` interface. `EntityTransaction` begins the transaction at the beginning of the apply request values phase.
- Hibernate managed transactions with the `org.hibernate.Transaction` interface. `HibernateTransaction` begins the transaction at the beginning of the apply request values phase.
- Spring managed transactions with the `org.springframework.transaction.PlatformTransactionManager` interface. The Spring `PlatformTransactionManagement` manager may begin the transaction at the beginning of the apply request values phase if the `userConversationContext` attribute is set.
- Explicitly disable Seam managed transactions

Configure JPA RESOURCE_LOCAL transaction management by adding the following to your components.xml where `#{em}` is the name of the `persistence:managed-persistence-context` component. If your managed persistence context is named `entityManager`, you can opt to leave out the `entity-manager` attribute. (see *Seam-managed persistence contexts*)

```
<transaction:entity-transaction entity-manager="#{em}"/>
```

To configure Hibernate managed transactions declare the following in your components.xml where `#{hibernateSession}` is the name of the project's `persistence:managed-hibernate-session` component. If your managed hibernate session is named `session`, you can opt to leave out the `session` attribute. (see *Seam-managed persistence contexts*)

```
<transaction:hibernate-transaction session="#{hibernateSession}"/>
```

To explicitly disable Seam managed transactions declare the following in your components.xml:

```
<transaction:no-transaction />
```

For configuring Spring managed transactions see *using Spring PlatformTransactionManagement* .

9.2.3. Transaction synchronization

Transaction synchronization provides callbacks for transaction related events such as `beforeCompletion()` and `afterCompletion()`. By default, Seam uses it's own transaction synchronization component which requires explicit use of the Seam transaction component when committing a transaction to ensure synchronization callbacks are correctly executed. If in a Java EE 5 environment the `<transaction:ejb-transaction/>` component should be be declared in `components.xml` to ensure that Seam synchronization callbacks are correctly called if the container commits a transaction outside of Seam's knowledge.

9.3. Seam-managed persistence contexts

If you're using Seam outside of a Java EE 5 environment, you can't rely upon the container to manage the persistence context lifecycle for you. Even if you are in an EE 5 environment, you might have a complex application with many loosly coupled components that collaborate together in the scope of a single conversation, and in this case you might find that propagation of the persistence context between component is tricky and error-prone.

In either case, you'll need to use a *managed persistence context* (for JPA) or a *managed session* (for Hibernate) in your components. A Seam-managed persistence context is just a built-in Seam component that manages an instance of `EntityManager` or `Session` in the conversation context. You can inject it with `@In`.

Seam-managed persistence contexts are extremely efficient in a clustered environment. Seam is able to perform an optimization that EJB 3.0 specification does not allow containers to use for container-managed extended persistence contexts. Seam supports transparent failover of extended persisence contexts, without the need to replicate any persistence context state between nodes. (We hope to fix this oversight in the next revision of the EJB spec.)

9.3.1. Using a Seam-managed persistence context with JPA

Configuring a managed persistence context is easy. In `components.xml`, we can write:

```
<persistence: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;
```

If you are using EJB3 and mark your class or method `@TransactionAttribute(REQUIRES_NEW)` then the transaction and persistence context shouldn't be propagated to method calls on this object. However as the Seam-managed persistence context is propagated to any component within the conversation, it will be propagated to methods marked `REQUIRES_NEW`. Therefore, if you mark a method `REQUIRES_NEW` then you should access the entity manager using @PersistenceContext.

9.3.2. Using a Seam-managed Hibernate session

Seam-managed Hibernate sessions are similar. In `components.xml`:

```
<persistence:hibernate-session-factory name="hibernateSessionFactory"/>
```

```
<persistence: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

property>
                                                                    <property

property>
   <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;
```

9.3.3. Seam-managed persistence contexts and atomic conversations

Persistence contexts scoped to the conversation allows you to program optimistic transactions that span multiple requests to the server without the need to use the `merge()` operation , without the need to re-load data at the beginning of each request, and without the need to wrestle with the `LazyInitializationException` or `NonUniqueObjectException`.

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();
}
```

Of course, you could set the `flushMode` to `MANUAL` from pages.xml, for example in a navigation rule:

```
<begin-conversation flush-mode="MANUAL" />
```

You can set any Seam Managed Persistence Context to use manual flush mode:

```
<components xmlns="http://jboss.com/products/seam/components"
  xmlns:core="http://jboss.com/products/seam/core">
  <core:manager conversation-timeout="120000" default-flush-mode="manual" />
</components>
```

9.4. Using the JPA "delegate"

The `EntityManager` interface lets you access a vendor-specific API via the `getDelegate()` method. Naturally, the most interesting vendor is Hibernate, and the most powerful delegate interface is `org.hibernate.Session`. You'd be nuts to use anything else. Trust me, I'm not biased at all. If you must use a different JPA provider see *Using Alternate JPA Providers*.

But regardless of whether you're using Hibernate (genius!) or something else (masochist, or just not very bright), you'll almost certainly want to use the delegate in your Seam components from time to time. One approach would be the following:

```
@In EntityManager entityManager;

@Create
public void init() {
   ( (Session) entityManager.getDelegate() ).enableFilter("currentVersions");
}
```

But typecasts are unquestionably the ugliest syntax in the Java language, so most people avoid them whenever possible. Here's a different way to get at the delegate. First, add the following line to `components.xml`:

```
<factory name="session"
      scope="STATELESS"
      auto-create="true"
      value="#{entityManager.delegate}"/>
```

Now we can inject the session directly:

```
@In Session session;
```

```
@Create
public void init() {
   session.enableFilter("currentVersions");
}
```

9.5. Using EL in EJB-QL/HQL

Seam proxies the `EntityManager` or `Session` object whenever you use a Seam-managed persistence context or inject a container managed persistence context using `@PersistenceContext`. This lets you use EL expressions in your query strings, safely and efficiently. For example, this:

```
User user = em.createQuery("from User where username=#{user.username}")
      .getSingleResult();
```

is equivalent to:

```
User user = em.createQuery("from User where username=:username")
      .setParameter("username", user.getUsername())
      .getSingleResult();
```

Of course, you should never, ever write it like this:

```
User user = em.createQuery("from User where username=" + user.getUsername()) //BAD!
      .getSingleResult();
```

(It is inefficient and vulnerable to SQL injection attacks.)

9.6. 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.)

```
<persistence:filter name="regionFilter">
   <persistence:name>region</persistence:name>
   <persistence:parameters>
      <key>regionCode</key>
      <value>#{region.code}</value>
   </persistence:parameters>
</persistence:filter>

<persistence:filter name="currentFilter">
   <persistence:name>current</persistence:name>
   <persistence:parameters>
      <key>date</key>
      <value>#{currentDate}</value>
   </persistence:parameters>
</persistence:filter>

<persistence:managed-persistence-context name="personDatabase"
   persistence-unit-jndi-name="java:/EntityManagerFactories/personDatabase">
   <persistence:filters>
      <value>#{regionFilter}</value>
      <value>#{currentFilter}</value>
   </persistence:filters>
</persistence:managed-persistence-context>
```

JSF form validation in Seam

In plain JSF, validation is defined in the view:

```
<h:form>
   <h:messages/>

   <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>

   <h:commandButton/>
</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>
    <h:messages/>

    <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>
```

```
   <h:commandButton/>

</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 approach *defines* constraints on the model, and *presents* constraint violations in the view — a significantly better design.

However, it is not much less verbose than what we started with, so let's try `<s:validateAll>`:

```
<h:form>

   <h:messages/>

   <s:validateAll>

      <div>
         Country:
         <h:inputText value="#{location.country}" required="true"/>
      </div>

      <div>
         Zip code:
         <h:inputText value="#{location.zip}" required="true"/>
      </div>

      <h:commandButton/>

   </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. In order for the user to correlate the message with an input, you need to define a label using the standard `label` attribute on the input component.

```
<h:inputText value="#{location.zip}" required="true" label="Zip:">
   <s:validate/>
</h:inputText>
```

You can then inject this value into the message string using the placeholder {0} (the first and only parameter passed to a JSF message for a Hiberate Validator restriction). See the internationalization section for more information regarding where to define these messages.

```
validator.length={0} length must be between {min} and {max}
```

What we would really like to do, though, is display the message next to the field with the error (this is possible in plain JSF), highlight the field and label (this is not possible) and, for good measure, display some image next to the field (also not possible). We also want to display a little colored asterisk next to the label for each required form field. Using this approach, the identifying label is not necessary.

That's quite a lot of functionality we need for each field of our form. We wouldn't want to have to specify higlighting and the layout of the image, message and input field for every field on the form. So, instead, we'll specify the common layout in a facelets template:

```
<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"
                xmlns:s="http://jboss.com/products/seam/taglib">

   <div>

      <s:label styleClass="#{invalid?'error':''}">
         <ui:insert name="label"/>
         <s:span styleClass="required" rendered="#{required}">*</s:span>
      </s:label>

      <span class="#{invalid?'error':''}">
         <h:graphicImage value="/img/error.gif" rendered="#{invalid}"/>
         <s:validateAll>
            <ui:insert/>
         </s:validateAll>
      </span>

      <s:message styleClass="error"/>
```

```
    </div>

</ui:composition>
```

We can include this template for each of our form fields using `<s:decorate>`.

```
<h:form>

    <h:messages globalOnly="true"/>

    <s:decorate template="edit.xhtml">
        <ui:define name="label">Country:</ui:define>
        <h:inputText value="#{location.country}" required="true"/>
    </s:decorate>

    <s:decorate template="edit.xhtml">
        <ui:define name="label">Zip code:</ui:define>
        <h:inputText value="#{location.zip}" required="true"/>
    </s:decorate>

    <h:commandButton/>

</h:form>
```

Finally, we can use RichFaces Ajax to display validation messages as the user is navigating around the form:

```
<h:form>

    <h:messages globalOnly="true"/>

    <s:decorate id="countryDecoration" template="edit.xhtml">
        <ui:define name="label">Country:</ui:define>
        <h:inputText value="#{location.country}" required="true">
            <a:support event="onblur" reRender="countryDecoration" bypassUpdates="true"/>
        </h:inputText>
    </s:decorate>

    <s:decorate id="zipDecoration" template="edit.xhtml">
        <ui:define name="label">Zip code:</ui:define>
        <h:inputText value="#{location.zip}" required="true">
```

```
            <a:support event="onblur" reRender="zipDecoration" bypassUpdates="true"/>
        </h:inputText>
    </s:decorate>

    <h:commandButton/>

</h:form>
```

It's better style to define explicit ids for important controls on the page, especially if you want to do automated testing for the UI, using some toolkit like Selenium. If you don't provide explicit ids, JSF will generate them, but the generated values will change if you change anything on the page.

```
<h:form id="form">

    <h:messages globalOnly="true"/>

    <s:decorate id="countryDecoration" template="edit.xhtml">
        <ui:define name="label">Country:</ui:define>
        <h:inputText id="country" value="#{location.country}" required="true">
            <a:support event="onblur" reRender="countryDecoration" bypassUpdates="true"/>
        </h:inputText>
    </s:decorate>

    <s:decorate id="zipDecoration" template="edit.xhtml">
        <ui:define name="label">Zip code:</ui:define>
        <h:inputText id="zip" value="#{location.zip}" required="true">
            <a:support event="onblur" reRender="zipDecoration" bypassUpdates="true"/>
        </h:inputText>
    </s:decorate>

    <h:commandButton/>

</h:form>
```

And what if you want to specify a different message to be displayed when validation fails? You can use the Seam message bundle (and all it's goodies like el expressions inside the message, and per-view message bundles) with the Hibernate Validator:

```
public class Location {
    private String name;
    private String zip;
```

```
    // Getters and setters for name

    @NotNull
    @Length(max=6)
    @ZipCode(message="#{messages['location.zipCode.invalid']}")
    public String getZip() { return zip; }
    public void setZip(String z) { zip = z; }
}
```

```
location.zipCode.invalid = The zip code is not valid for #{location.name}
```

L'intégration avec Groovy

Un des aspects de JBoss Seam est dans sa capacité RAD (Rapid Application Development, NdT Développement d'Application Rapide). Sans synonimie avec RAD, l'outil intéréssant dans ce cas est dans les langages dynamiques. Jusqu'à présent, choisir un langage dynamique était un choix qui été requis par une plateforme de développement complètement différente (une plateforme de développement avec un groupe d'APIs et un runtime si génial que vous ne devriez plus vouloir utiliser vos bonnes vieille APIs Java [sic] après, ce qui est une chance car vous ne devriez plus être forcé d'utiliser ces APIs propriétaire maintenant). Les langages dynamiques sont construis au dessus de la Machine Virtuelle Java et *Groovy* [http://groovy.codehaus.org] en particulier brise cette approche en silos.

JBoss Seam unifie maintenant le monde des langages dynamiques avec le monde Java EE en intégrant sans couture les deux langages statique et dynamique. JBoss Seam permet au développer d'application d'utiliser le meilleur outil pour la tâche, sans basculer de contexte. Ecrire les composants dynamiques de Seam est exactement comme écrire des composants classiques de Seam. Vous utilisez les même annotations, les même APIs, les mêmes tout.

11.1. Introduction à Groovy

Groovy est un langage dynamique agile basé sur le langage Java avec des fonctionnalité additionnelles inspirée par Python, Ruby et Smalltalk. La force de Groovy est dans deux points:

- La syntaxe Java est supportée dans Groovy: le code Java est le code Groovy, ce qui rend la courbe d'apprentissage très douce
- Les objets Groovy sont des objets Java, et les classes Groovy sont des classes Java: Groovy s'intègre facilement avec les bibliothèques Java et les serveurs d'applications.

A FAIRE: ecrire un rapide apperçu de l'ajout de syntaxe Groovy

11.2. L'écriture d'applications Seam en Groovy

Il n'y a pas grand chose à dire à propos de cela. Si un objet Groovy est un objet Java, vous pouvez virtuellement écrire tout composant Seam, ou toute classe pour ce qui est prévu, en Groovy et la déployer. Vous pouvez aussi mélanger les classes Groovy et Java dans la même application.

11.2.1. Ecriture de composants en Groovy

Comme vous devriez avoir noté maintenant, Seam utilise lourdement les annotations. Soyez sûr d'utiliser Groovy 1.1 ou supérieur pour le support des annotations. Voici un exemple de code en groovy utilisé dans une application Seam.

11.2.1.1. Entité

```
@Entity
```

```
@Name("hotel")
class Hotel implements Serializable
{
    @Id @GeneratedValue
    Long id

    @Length(max=50) @NotNull
    String name

    @Length(max=100) @NotNull
    String address

    @Length(max=40) @NotNull
    String city

    @Length(min=2, max=10) @NotNull
    String state

    @Length(min=4, max=6) @NotNull
    String zip

    @Length(min=2, max=40) @NotNull
    String country

    @Column(precision=6, scale=2)
    BigDecimal price

    @Override
    String toString()
    {
        return "Hotel(${name},${address},${city},${zip})"
    }
}
```

Groovy supporte nativement la notion de propriété (asseseurs), donc il n'y a pas besoin d'écrire de verbeux assesseurs: dans l'exemple précédent, la classe hotel peut être accédé depuis Java comme `hotel.getCity()`, les asseseurs peuvent être générés par le compilateur Groovy. Le type de la syntaxe pur sucre rends le code de l'entité très concise.

11.2.1.2. Le composant Seam

Ecrire les composants Seam en Groovy n'est pas vraiment différent qu'en Java: les annotations sont utilisées pour marquer la classe comme un composant de Seam.

```
@Scope(ScopeType.SESSION)
@Name("bookingList")
class BookingListAction implements Serializable
{
   @In EntityManager em
   @In User user
   @DataModel List<Booking> bookings
   @DataModelSelection Booking booking
   @Logger Log log

   @Factory public void getBookings()
   {
      bookings = em.createQuery("'
            select b from Booking b
            where b.user.username = :username
            order by b.checkinDate"')
         .setParameter("username", user.username)
         .getResultList()
   }

   public void cancel()
   {
      log.info("Cancel booking: #{bookingList.booking.id} for #{user.username}")
      Booking cancelled = em.find(Booking.class, booking.id)
      if (cancelled != null) em.remove( cancelled )
      getBookings()
                  FacesMessages.instance().add("Booking cancelled for confirmation number
 #{bookingList.booking.id}", new Object[0])
   }
}
```

11.2.2. seam-gen

Seam-gen as une intégration transparente avec Groovy. Vous pouvez écrire le code Groovy dans un projet géré par seam-gen sans aucun prérequis d'infrastructure additionnel. Quand à écrire une entité Groovy, placez simplement vos fichiers `.groovy` dans `src/main`. Sans surprise, avec l'écriture d'une action, placez simplement vos fichiers `.groovy` dans `src/hot`.

11.3. Le déploiement

Le déploiement des classes Groovy est assez proche du déploiement des classes Java (sans surprise, pas besoin d'écrire ni de se conformer avec la spécification composite en 3 lettres pour supportert un serveur d'application multi langage).

Au delà des déploiement standard, JBoss Seam a la capacité ,au moment du développement, de redéployer les classes de composants Seam JavaBeans sans avoir à redémarrer l'application, préservant beaucoup de temps de développement dans le cycle développement/test. Le même support est fourni par les composants de Seam de GroovyBeans quand les fichiers `.groovy` sont déployés.

11.3.1. Le déploiement de code Groovy

Une classe Groovy *est* une classe Java avec une représentation en bytecode tout comme une classe Java. Pour déployer une entité Groovy, un bean de session Groovy ou un composant Seam Groovy, une étape de compilation est nécéssaire. Une approche commune est d'utiliser la tache ant `groovyc`. Une fois compilé, une classe Groovy n'est en acune manière différente d'une classe Java et le serveur d'application la traitera à équalité. Notez que cela autorise un mélange intégré de Groovy et de code Java.

11.3.2. Le déploiement de fichier .groovy natif au moment du déploiement

JBoss Seam supporte nativement le déploiement de fichier `.groovy` f (autrement dit sans compilation) dans le mode de redéploiement à chaud incrémental (seulement au développement). Cela rend possible un cycle edition/test très rapide. Pour configurer le déploiement de .groovy, suivez la configuration de la *Section 2.8, « Seam et le déploiement incrémental à chaud »* et deployez votre code Groovy (les fichiers `.groovy`) dans le dossier `WEB-INF/dev`. Les composants GroovyBean vont être pris incrémentallement sans avoir besoin de redémarrer l'application (ni heureusement le serveur d'application).

Soyez informé que le déploiement de fichier .groovy natif souffre des limitations identique du redéploiement à chaud classique de Seam:

- Les composants doivent être des JavaBeans ou des GroovyBeans. Ils ne peuvent pas être des beans EJB3
- Les entités ne peuvent pas redéployées à chaud
- Le déploiement à chaud des composants ne sera pas visible pour toute classes deployé à l'extérieur de `WEB-INF/dev`
- Le mode debug de Seam doit être activé

11.3.3. seam-gen

Seam-gen supporte de manière transparent le déploiement et la compilation de fichiers Groovy. Cela inclus le déploiement de fichier `.groovy` natif dans le mode développement (sans compilation). Si vous créez un projet seam-gen de type WAR, les classes Java et Groovy dans `src/hot` vont être automatiquement candidate pour le déploiement à chaud incrémental. Si

vous êtes en mode de production, les fichiers Groovy vont être simplement compilés avant le déploiement.

Vous trouverez un exemple en ligne de la démonstration de la Réservation d'Hotel écrit totalement en Groovy et supportant le déploiement incrémental à chaud dans `examples/groovybooking`.

Writing your presentation layer using Apache Wicket

Seam supports Wicket as an alternative presentation layer to JSF. Take a look at the `wicket` example in Seam which shows the Booking Example ported to Wicket.

Note

Wicket support is new to Seam, so some features which are available in JSF are not yet available when you use Wicket (e.g. pageflow). You'll also notice that the documentation is very JSF-centric and needs reorganization to reflect the first class support for Wicket.

12.1. Adding Seam to your wicket application

The features added to your Wicket application can be split into two categories: bijection and orchestration; these are discussed in detail below.

Extensive use of inner classes is common when building Wicket applications, with the component tree being built in the constructor. Seam fully supports the use of annotation based control in inner classes and constructors (unlike regular Seam components).

Annotations are processed *after* any call to a superclass. This mean's that any injected attributes cannot be passed as an argument in a call to `this()` or `super()`.

Note

We are working to improve this.

When a method is called in an inner class, bijection occurs for any class which encloses it. This allows you to place your bijected variables in the outer class, and refer to them in any inner class.

12.1.1. Bijection

A Seam enabled Wicket application has full access to the all the standard Seam contexts (`EVENT`, `CONVERSATION`, `SESSION`, `APPLICATION` and `BUSINESS_PROCESS`).

To access Seam component's from Wicket, you just need to inject it using `@In`:

```
@In(create=true)
private HotelBooking hotelBooking;
```

Astuce

As your Wicket class isn't a full Seam component, there is no need to annotate it `@Name`.

You can also outject an object into the Seam contexts from a Wicket component:

```
@Out(scope=ScopeType.EVENT, required=false)
private String verify;
```

TODO Make this more use case driven

12.1.2. Orchestration

You can secure a Wicket component by using the `@Restrict` annotation. This can be placed on the outer component or any inner components. If `@Restrict` is specified, the component will automatically be restricted to logged in users. You can optionally use an EL expression in the `value` attribute to specify a restriction to be applied. For more refer to the *Chapitre 15, Security*.

For example:

```
@Restrict
public class Main extends WebPage {

  ...
```

Astuce

Seam will automatically apply the restriction to any nested classes.

You can demarcate conversations from within a Wicket component through the use of `@Begin` and `@End`. The semantics for these annotations are the same as when used in a Seam component. You can place `@Begin` and `@End` on any method.

Note

The deprecated `ifOutcome` attribute is not supported.

For example:

```
item.add(new Link("viewHotel") {

  @Override
  @Begin
  public void onClick() {
    hotelBooking.selectHotel(hotel);
    setResponsePage(org.jboss.seam.example.wicket.Hotel.class);
  }
};
```

You may have pages in your application which can only be accessed when the user has a long-running conversation active. To enforce this you can use the `@NoConversationPage` annotation:

```
@Restrict
@NoConversationPage(Main.class)
public class Hotel extends WebPage {
```

If you want to further decouple your application classes, you can use Seam events. Of course, you can raise an event using `Events.instance().raiseEvent("foo")`. Alternatively, you can annotate a method `@RaiseEvent("foo")`; if the method returns a non-null outcome without exception, the event will be raised.

You can also control tasks and processes in Wicket classes through the use of `@CreateProcess`, `@ResumeTask`, `@BeginTask`, `@EndTask`, `@StartTask` and `@Transition`.

TODO - Implement BPM control - JBSEAM-3194

12.2. Setting up your project

Seam needs to instrument the bytecode of your Wicket classes to be able to intercept the annotations you use. The first decision to make is: do you want your code instrumented at runtime as your app is running, or at compile time? The former requires no integration with your build environment, but has a performance penalty when loading each instrumented class for the first time. The latter is faster, but requires you to integrate this instrumentation into your build environment.

12.2.1. Runtime instrumentation

There are two ways to achieve runtime instrumentation. One relies on placing wicket components to be instrumented in a special folder in your WAR deployment. If this is not acceptable or possible, you can also use an instrumentation "agent," which you specify in the command line for launching your container.

12.2.1.1. Location-specific instrumentation

Any classes placed in the `WEB-INF/wicket` folder within your WAR deployment will be automatically instrumented by the seam-wicket runtime. You can arrange to place your wicket pages and components here by specifying a separate output folder for those classes in your IDE, or through the use of ant scripts.

12.2.1.2. Runtime instrumentation agent

The jar file `jboss-seam-wicket.jar` can be used as an iinstrumentation agent through the Java Instrumentation api. This is accomplished through the following steps:

- Arrange for the `jboss-seam-wicket.jar` file to live in a location for which you have an absolute path, as the Java Instrumentation API does not allow relative paths when specifying the location of an agent lib.
- Add `javaagent:/path/to/jboss-seam-wicket.jar` to the command line options when launching your webapp container:
- In addition, you will need to add an environment variable that specifies packages that the agent should instrument. This is accomplished by a comma separated list of package names:

```
-Dorg.jboss.seam.wicket.instrumented-packages=my.package.one,my.other.package
```

Note that if a package A is specified, classes in subpackages of A are also examined. The classes chosen for instrumentation can be further limited by specifying:

```
-Dorg.jboss.seam.wicket.scanAnnotations=true
```

and then marking instrumentable classes with the `@SeamWicketComponent` annotation, see *Section 12.2.3, « The `@SeamWicketComponent` annotation »*.

12.2.2. Compile-time instrumentation

Seam supports instrumentation at compile time through either Apache Ant or Apache Maven.

12.2.2.1. Instrumenting with ant

Seam provides an ant task in the `jboss-seam-wicket-ant.jar` . This is used in the following manner:

```
<taskdef name="instrumentWicket"
  classname="org.jboss.seam.wicket.ioc.WicketInstrumentationTask">
 <classpath>
```

```
    <pathelement location="lib/jboss-seam-wicket-ant.jar"/>
    <pathelement location="web/WEB-INF/lib/jboss-seam-wicket.jar"/>
    <pathelement location="lib/javassist.jar"/>
    <pathelement location="lib/jboss-seam.jar"/>
  </classpath>
</taskdef>

<instrumentWicket outputDirectory="${build.instrumented}" useAnnotations="true">
  <classpath refid="build.classpath"/>
  <fileset dir="${build.classes}" includes="**/*.class"/>
</instrumentWicket>
```

This results in the instrumented classes being placed in the directory specified by `${build.instrumented}`. You will then need to instruct ant to copy these classes into `WEB-INF/classes`. If you want to hot deploy the Wicket components, you can copy the instrumented classes to `WEB-INF/dev`; if you use hot deploy, make sure that your `WicketApplication` class is also hot-deployed. Upon a reload of hot-deployed classes, the entire WicketApplication instance has to be re-initialized, in order to pick up new references to the classes of mounted pages.

The `useAnnotations` attribute is used to make the ant task only include classes that have been marked with the `@SeamWicketComponent` annotation, see *Section 12.2.3, « The `@SeamWicketComponent` annotation »*.

12.2.2.2. Instrumenting with maven

The jboss maven repository `repository.jboss.org` provides a plugin named `seam-instrument-wicket` with a `process-classes` mojo. An example configuration in your pom.xml might look like:

```
<build>
        <plugins>
         <plugin>
            <groupId>org.jboss.seam</groupId>
            <artifactId>seam-instrument-wicket</artifactId>
            <version>2.2.0</version>
            <configuration>
                <scanAnnotations>true</scanAnnotations>
                <includes>
                    <include>your.package.name</include>
                </includes>
            </configuration>
            <executions>
                <execution>
                    <id>instrument</id>
```

```
                        <phase>process-classes</phase>
                        <goals>
                            <goal>instrument</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
</build>
```

The above example illustrates that the instrumentation is limited to classes specified by the `includes` element. In this example, the `scanAnnotations` is specified, see *Section 12.2.3, « The `@SeamWicketComponent` annotation »*.

12.2.3. The `@SeamWicketComponent` annotation

Classes placed in WEB-INF/wicket will unconditionally be instrumented. The other instrumentation mechanisms all allow you to specify that instrumentation should only be applied to classes annotated with the `@SeamWicketComponent` annotation. This annotation is inherited, which means all subclasses of an annotated class will also be instrumented. An example usage is:

```
import org.jboss.seam.wicket.ioc.SeamWicketComponent;
@SeamWicketComponent
public class MyPage extends WebPage{
 ...
}
```

12.2.4. Defining the Application

A Wicket web application which uses Seam should use `SeamWebApplication` as the base class; this creates hooks into the Wicket lifecycle allowing Seam to automagically propagate the conversation as needed. It also adds status messages to the page.

For example:

The `SeamAuthorizationStrategy` delegates authorization to Seam Security, allowing the use of `@Restrict` on Wicket components. `SeamWebApplication` installs the authorization strategy for you. You can specify the login page by implementing the `getLoginPage()` method.

You'll also need to set the home page of the application by implementing the `getHomePage()` method.

```
public class WicketBookingApplication extends SeamWebApplication {
```

```
  @Override
  public Class getHomePage() {
    return Home.class;
  }

  @Override
  protected Class getLoginPage() {
    return Home.class;
  }

}
```

Seam automatically installs the Wicket filter for you (ensuring that it is inserted in the correct place for you). But you still need to tell Wicket which `WebApplication` class to use.

```
<components xmlns="http://jboss.com/products/seam/components"
 xmlns:wicket="http://jboss.com/products/seam/wicket"
 xsi:schemaLocation=
  "http://jboss.com/products/seam/wicket
   http://jboss.com/products/seam/wicket-2.2.xsd">

  <wicket:web-application
    application-class="org.jboss.seam.example.wicket.WicketBookingApplication" />
</components
```

In addition, if you plan to use JSF-based pages in the same application as wicket pages, you'll need to ensure that the jsf exception filter is only enabled for jsf urls:

```
<components xmlns="http://jboss.com/products/seam/components"
 xmlns:web="http://jboss.com/products/seam/web"
 xmlns:wicket="http://jboss.com/products/seam/wicket"
 xsi:schemaLocation=
  "http://jboss.com/products/seam/web
   http://jboss.com/products/seam/web-2.2.xsd">

   <!-- Only map the seam jsf exception filter to jsf paths, which we identify with the *.seam path -->
       <web:exception-filter url-pattern="*.seam"/>
</components
```

Astuce

Take a look at the Wicket documentation for more on authorization strategies and other methods you can override on the `Application` class.

Le serveur d'application Seam

Seam rends vraiment plus simple de créer des applications en écrivant les pures-classes Java avec les annotations, qui n'ont pas besoin d'être étendue avec des interfaces spéciales ou des super-classes. Mais nous pouvons simplifier beaucoup plus des tâches communes de programmation en fournissant un groupe de composant pré-livrés qui peuvent être réutilisés en les configurant dans `components.xml` (pour les cas très simple) ou par extention.

Le *Serveur d'Application Seam* peut réduire la quantité de code que vous avez besoin d'écrire quand vous voulez faire les accès classiques à la base de données dans une application web, en utilisant aussi bien Hibernate ou JPA.

Vous devriez apprécier que le serveur d'application soit extrèmement simple, juste un bagage de classes simples qui sont faciles à comprendre et à étendre. La "magie" est dans Seam lui-même — la même magie que vous utiliser quand vous créez tout application Seam même sans utiliser le serveur d'application.

13.1. Introduction

Les composants fournies par le serveur d'application de Seam peuvent être utilisés d'une ou de deux manières différentes. La première façon est d'installer et de configurer une instance de composant dans `components.xml`, tout comme nous avons fait avec les autres types de composants livrés par Seam. Par exemple, le fragment suivant de `components.xml` installe un composant qui va réaliser les opérations basiques de CRUD pour l'entité `Person`:

```
<framework:entity-home name="personHome"
                       entity-class="eg.Person"
                       entity-manager="#{personDatabase}">
   <framework:id
>#{param.personId}</framework:id>
</framework:entity-home
>
```

i cela ressemble un peu beaucoup à "programmation en XML" à votre goût, vous pouvez utiliser une extention à la place:

```
@Name("personHome")
public class PersonHome extends EntityHome<Person
> {

   @In EntityManager personDatabase;
```

```
  public EntityManager getEntityManager() {
    return personDatabase;
  }

}
```

La seconde approche a un énorme avantage; vous pouvez facilement ajouter des fonctionnalités additionnelles, et surcharger les fonctionnalitées livrées par défaut (les classes du serveurd'applications sont précisément conçues pour l'extention et la personalisation).

Un deuxième avantage est que vos classes peuvent être des beans de session EJB avec état, si vous préférez (Ils ne sont pas obligés de l'être, ils peuvent être de pur composants JavaBean si vous préférez.) Si vous utiliser JBoss AS, vous allez avoir besoin de la 4.2.2GA ou supérieure:

```
@Stateful
@Name("personHome")
public class PersonHome extends EntityHome<Person
> implements LocalPersonHome {

}
```

Vous pouvez aussi faire de vos classes des beans de sessions sans état. Dans ce cas, vous *devez* utiliser l'injection pour fournir le contexte de persistance, meême s'il ya un appel à `entityManager`:

```
@Stateless
@Name("personHome")
public class PersonHome extends EntityHome<Person
> implements LocalPersonHome {

  @In EntityManager entityManager;

  public EntityManager getPersistenceContext() {
    entityManager;
  }

}
```

A cette étape, le Serveur d'application de Seam fourni tout juste quatre composants livrés: `EntityHome`et `HibernateEntityHome` pour le CRUD, avec `EntityQuery` et `HibernateEntityQuery` pour les requêtes.

es composants Home et Query sont écris de façon qu' ils puissent fonctionner dans une étendue de session, d'évènement ou de conversation. L'étendue que vous utiliserez dépendra du modèle à état que vous souhaiterez utiliser dans votre application.

Le Serveur d'application de Seam fonctionnement seulement dans les contextes de persistances gérés par Seam. Par défaut, les composants vont chercher un contexte de persistance nommé `entityManager`.

13.2. Les objets Home

Un objet Home fourni les opérations de persistance pour une classe d'entité particulière. Supposons que nous avons notre classe valide `Person` class:

```
@Entity
public class Person {
   @Id private Long id;
   private String firstName;
   private String lastName;
   private Country nationality;

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

Nous pouvons définir un composant `personHome` via la configuration suivante:

```
<framework:entity-home name="personHome" entity-class="eg.Person" />
```

Ou via une extension:

```
@Name("personHome")
public class PersonHome extends EntityHome<Person
> {}
```

Un objet Home fourni les opérations suivantes: `persist()`, `remove()`, `update()` et `getInstance()`. Avant que vous puissiez appeler les opérations `remove()`, ou `update()` vous devez en premier définir l'identification de l'objet dont vous êtes en train de vous intéresser en utilisant la méthode s `setId()`.

Vous pouvez utiliser un Home directement depuis une page JSF, pour exemple:

```
<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
>
```

Habituellement, il est plus jolie d'être capable de se référer à la `Person`presque comme une `person`, rendons cela possible en ajoutant une ligne à `components.xml`:

```
<factory name="person"
        value="#{personHome.instance}"/>

<framework:entity-home name="personHome"
                    entity-class="eg.Person" />
```

(Si nous utilisons la configuration). Ou en ajoutant la méthode `@Factory` a `PersonHome`:

```
@Name("personHome")
public class PersonHome extends EntityHome<Person
> {

   @Factory("person")
   public Person initPerson() { return getInstance(); }

}
```

(Si vous utilisons l'extention.) Cette modification simplifie notre page JSF comme ci-dessous:

```
<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
>
```

Et bien, cela nous permet de créer de nouvelles entrées `Person` . Oui, c'est bien tout le code nécéssaire! Maintenant, si vous voulont être capable d'afficher, de modifier, d'éffacer des `Person` préexistantes dans la base de données, nous avons besoin de passer l'identifiant de l'entrée à `PersonHome`. Les paramètres de pages sont une excelente manière de faire cela:

```
<pages>
   <page view-id="/editPerson.jsp">
      <param name="personId" value="#{personHome.id}"/>
   </page>
</pages
>
```

Maintenant, nous pouvons ajouter les opérations additionnelles à notre page JSF:

```
<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.managed}"/>
                  <h:commandButton value="Update Person" action="#{personHome.update}"
 rendered="#{personHome.managed}"/>
                  <h:commandButton value="Delete Person" action="#{personHome.remove}"
 rendered="#{personHome.managed}"/>
   </div>
</h:form
>
```

Quand nous lions notre page sans paramètres de requêtes, la page va être affiché comme une page "Création d'une personne". Quand nous fournissons une valeur au paramètre de requête `personId`, cela va être une page "Edition d'une personne".

Supposons que nous avons besoin de créer une entrée `Person` avec sa nationnalité initialisée. Nous pouvons faire cela facilement via la 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
>
```

Ou par extention:

```
@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);
    }

}
```

Bien sur, le `Country` peut être un objet géré par un autre objet Home, par exemple, `CountryHome`.

Pour ajouter des opérations plus sophistiquées (gestion d'association, etc.), nous pouvons juste ajouter les méthodes à `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();
   }

}
```

L'objet Home déclenche un évènement `org.jboss.seam.afterTransactionSuccess` quand une transaction réussie (quand un appel à `persist()`, `update()` ou `remove()` réussit). En observant cet évènement nous pouvons rafraichir nos requêtes quand une entitées sousjacentes se modifie. Si nous voulons seulement rafraichir certains requêtes quand un entité particulière est persistée, mise-à-jour ou retirée, nous pouvons observer l'évènement `org.jboss.seam.afterTransactionSuccess.<name>` (avec `<name>` qui est le nom simple de l'entité par exemple une entité appelée "org.foo.myEntity" a comme nom simple "myEntity").

L'objet Home affiche automatiquement les messages faces quand une opération est réussit. Pour personnaliser ces messages, nous pouvons, encore, utiliser la 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:created-message>
   <framework:deleted-message
>Person #{person.firstName} #{person.lastName} deleted</framework:deleted-message>
   <framework:updated-message
```

```
>Person #{person.firstName} #{person.lastName} updated</framework:updated-message>
</framework:entity-home>

<component name="newPerson"
        class="eg.Person">
   <property name="nationality"
>#{country}</property>
</component
>
```

Ou l'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 createValueExpression("New person
#{person.firstName} #{person.lastName} created"); }
         protected String getUpdatedMessage() { return createValueExpression("Person
#{person.firstName} #{person.lastName} updated"); }
         protected String getDeletedMessage() { return createValueExpression("Person
#{person.firstName} #{person.lastName} deleted"); }

}
```

Mais la meilleure façon pour spécifier le message est de le mettre dans le lot de ressource connu par Seam (le lot nommé `messages`, par défaut).

```
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
```

Cela active l'internationnalisation et converse votre code et votre configuration propre de ce qui concerne la présentation.

L'étape finale est d'ajouter la fonctionnalité de validation à la page, en utilisant `<s:validateAll>` et `<s:decorate>`, mais je vais vous laisser trouver.

13.3. Le objets Query

Si nous avons besoin d'une liste de toutes les instance de `Person` dans la base de données, nous pouvons utiliser un objet Query object. Par exemple:

```
<framework:entity-query name="people"
                        ejbql="select p from Person p"/>
```

Nous pouvons l'utiliser dans une page JSF:

```
<h1
>List of people</h1>
<h:dataTable value="#{people.resultList}" var="person">
   <h:column>
      <s:link view="/editPerson.jsp" value="#{person.firstName} #{person.lastName}">
         <f:param name="personId" value="#{person.id}"/>
      </s:link>
   </h:column>
</h:dataTable
>
```

Nous avons probablement besoin de supporter la pagination:

```
<framework:entity-query name="people"
                        ejbql="select p from Person p"
                        order="lastName"
                        max-results="20"/>
```

Nous allons utiliser un paramètre de page pour déterminer la page à afficher:

```
<pages>
   <page view-id="/searchPerson.jsp">
      <param name="firstResult" value="#{people.firstResult}"/>
   </page>
```

```
</pages
>
```

Le code JSF pour le contrôle de la pagination est un peu verbeux mais gérable:

```
<h1
>Search for people</h1>
<h:dataTable value="#{people.resultList}" var="person">
   <h:column>
      <s:link view="/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
>
```

Les écrans de recherche réel permettent à l'utilisateur d'entrer un tas de critères de recherches optionnels pour réduire la liste des résultats retournés. L'objet Query vous permet d'indiquer les "restrictions" optionneles pour supporter ce cas d'utilisation important:

```
<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( concat(#{examplePerson.firstName},'%') )</value>
      <value
>lower(lastName) like lower( concat(#{examplePerson.lastName},'%') )</value>
   </framework:restrictions>
</framework:entity-query
>
```

Notez l'utilisation d'un objet "exemple".

```
<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="/editPerson.jsp" value="#{person.firstName} #{person.lastName}">
         <f:param name="personId" value="#{person.id}"/>
      </s:link>
   </h:column>
</h:dataTable
>
```

Pour rafraichir la requête quand les entitées sousjacentes changent, nous oberservons l'évènement `org.jboss.seam.afterTransactionSuccess`:

```
<event type="org.jboss.seam.afterTransactionSuccess">
   <action execute="#{people.refresh}" />
</event
>
```

Ou juste pour rafraichir la requête quand l'entité personne est peristée, mise-à-jours ou retirée au travers de `PersonHome`:

```
<event type="org.jboss.seam.afterTransactionSuccess.Person">
   <action execute="#{people.refresh}" />
   </event
>
```

Malheureusement, les objets Query ne fonctionnent pas bien avec les reuqêtes *join fetch* - l'utilisation de la pagination avec ces requêtes n'est pas recommandée, et vous allez devoir implémenter votre propre méthode pour calculer le nombre total de résultats (en surchargeant `getCountEjbql()`).

Les exemples de cette section qui ont été montré utilisent la configuration. Cependant les utiliser par l'extension est égale possible pour les objets Query.

13.4. Les objets Controleur

Une partie complètement optionnelle du Serveur d'Application Seam est la classe `Controller` et ses sousclasses `EntityController` `HibernateEntityController` et `BusinessProcessController`. Ces classes ne fournissent rien de plus que des méthodes pratiques pour acceder aux composants livré utilisé communément. Ils permettent de d'éviter quelques frappes au claviers (les touches peuvent se reposer) et fournissent une bonne base de lancement pour les nouveaux utilisateurs afin qu'ils explorent les fonctionnalités riches livrées dans Seam.

Par exemple, ici c'est le `RegisterAction` de l'exemple de reservation de Seam devrait ressemble :

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

  @In private User user;

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

    if ( existing.size()==0 )
    {
      persist(user);
      info("Registered new user #{user.username}");
      return "/registered.jspx";
```

```
    }
    else
    {
      addFacesMessage("User #{user.username} already exists");
      return null;
    }
  }

}
```

Comme vous pouvez le voir, ce n'est pas une ammélioration stupéfiante...

Seam et Jboss Rules

Seam rends facile l'appel aux règles de bases de JBoss Rules (Drools) depuis les composants Seam ou la définition de processus jBPM.

14.1. Installation des règles

La première étape est de rendre une instance de `org.drools.RuleBase` disponible dans une variable de contexte de Seam. Pour les besoins de tests, Seam fourni un composant livré qui compile un groupe de règles statiques depuis le classpath. Vous pouvez installer ce composant via `components.xml`:

```
<drools:rule-base name="policyPricingRules">
   <drools:rule-files>
      <value
>policyPricingRules.drl</value>
   </drools:rule-files>
</drools:rule-base
>
```

Ce composant compiles les règles depuis un groupe de fichiers (`.drl`) ou d'un fichier de table de décision (`.xls`) et met en cache une instance de `org.drools.RuleBase` dans le contexte `APPLICATION` de Seam. Notez que c'est presque comme cela que vous allez devoir installer les bases de règles multiples dans une application conduite par les règles.

Si vous voulez utiliser un Drools DSL, vous avez aussi besoin de spécifier la définition DSL :

```
<drools:rule-base name="policyPricingRules" dsl-file="policyPricing.dsl">
   <drools:rule-files>
      <value
>policyPricingRules.drl</value>
   </drools:rule-files>
</drools:rule-base
>
```

Le support de Drools RuleFlow est aussi disponible et peut vous simplifier l'ajout de `.rf` ou de `.rfm` comme morceau de vos fichiers de règles comme:

```
<drools:rule-base name="policyPricingRules" rule-files="policyPricingRules.drl,
policyPricingRulesFlow.rf"/>
```

Notez qu'avec l'utilisation du format Drools 4.x RuleFlow (`.rfm`), vous allez avoir besoin de spécifier la propriété système -Ddrools.ruleflow.port=true au démarrage du serveur. Ceci est cependant toujours une fonctionnalité expérimentale et nous conseillons d'utiliser le format Drools5 (`.rf`) si possible.

Si vous voulez enregistrer un gestionnaire d'exception de conséquence particulière ay travers de RuleBaseConfiguration, vous allez devoir écrire le gestionnaire, par exemple:

```
@Scope(ScopeType.APPLICATION)
@Startup
@Name("myConsequenceExceptionHandler")
public class MyConsequenceExceptionHandler implements ConsequenceExceptionHandler,
 Externalizable {

  public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
  }

  public void writeExternal(ObjectOutput out) throws IOException {
  }

  public void handleException(Activation activation,
                              WorkingMemory workingMemory,
                              Exception exception) {
     throw new ConsequenceException( exception,
                                     activation.getRule() );
  }

}
```

et l'enregistrer:

```
<drools:rule-base name="policyPricingRules" dsl-file="policyPricing.dsl" consequence-
exception-handler="#{myConsequenceExceptionHandler}">
   <drools:rule-files>
      <value
>policyPricingRules.drl</value>
   </drools:rule-files>
</drools:rule-base
>
```

Dans la plus part des applications conduites par les règles, les règles doivent être déployable dynamiquement pour que l'application en production qui veux utiliser un Drools RuleAgent pour gérer la RuleBase. Le RuleAgent peut se connecter au seveur de règle Drools (BRMS) ou aux packages de règles déployés à chauds depuis le référenciel de fichier local. La RuleBase gérant le RulesAgent-managed est aussi configurable dans `components.xml`:

```
<drools:rule-agent name="insuranceRules"
                   configurationFile="/WEB-INF/deployedrules.properties" />
```

Le fichier de propriété contient les propriétés spécifiques au RulesAgent. Voici un exemple de fichier de configuration depuis la distribution exemple de Drools.

```
newInstance=true
url=http://localhost:8080/drools-jbrms/org.drools.brms.JBRMS/package/org.acme.insurance/
fmeyer
localCacheDir=/Users/fernandomeyer/projects/jbossrules/drools-examples/drools-examples-
brms/cache
poll=30
name=insuranceconfig
```

Il est aussi possible de configurer les options du composant directement, en outrepassant le fichier de configuration.

```
<drools:rule-agent name="insuranceRules"
    url="http://localhost:8080/drools-jbrms/org.drools.brms.JBRMS/package/org.acme.insurance/
fmeyer"
                   local-cache-dir="/Users/fernandomeyer/projects/jbossrules/drools-examples/drools-
examples-brms/cache"
   poll="30"
   configuration-name="insuranceconfig" />
```

Ensuite, vous allons avoir besoin de faire une instance de `org.drools.WorkingMemory` disponible pour chaque conversation. (Chaque `WorkingMemory` accumule les faits relatif à la conversation courante.)

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

Notes que nous avons donné au `policyPricingWorkingMemory` une référence en retour vers notre base de règles via la propriété de configuration `ruleBase`.

Nous pouvons aussi en faire plus pour être informé des évènements du moteur de règles, ce qui inclus le déclenche de règles, le fait que des objets soient injecté, etc. en ajoutant des écouteur d'évènements au WorkingMemory.

```
<drools:managed-working-memory  name="policyPricingWorkingMemory"  auto-create="true"
 rule-base="#{policyPricingRules}">
      <drools:event-listeners>
         <value
>org.drools.event.DebugWorkingMemoryEventListener</value>
      <value
>org.drools.event.DebugAgendaEventListener</value>
      </drools:event-listeners>
</drools:managed-working-memory
>
```

14.2. L'utilisation des règles d'un composant de Seam

Nous pouvons injecter notre `WorkingMemory` dans tout composant de Seam, insérer des faits, et déclencher des règles:

```
@In WorkingMemory policyPricingWorkingMemory;

@In Policy policy;
@In Customer customer;

public void pricePolicy() throws FactException
{
      policyPricingWorkingMemory.insert(policy);
      policyPricingWorkingMemory.insert(customer);
      // if we have a ruleflow, start the process
      policyPricingWorkingMemory.startProcess(startProcessId)
   policyPricingWorkingMemory.fireAllRules();
}
```

14.3. L'utilisation des règles depuis une définition de processus jBPM

Nous pouvons même autoriser une base de règle à agir comme un gestionnaire d'action jBPM, un gestionnaire de décision, ou un gestionnaire d'affectation — à la fois dans un enchainement de page ou dans la définition de processus métier.

```
<decision name="approval">

   <handler class="org.jboss.seam.drools.DroolsDecisionHandler">
      <workingMemoryName
>orderApprovalRulesWorkingMemory</workingMemoryName>
      <!-- if a ruleflow was added -->
      <startProcessId
>approvalruleflowid</startProcessId>
      <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
```

```
>
```

L'élément `<assertObjects>` spécifie des expressions EL qui retourne un objet ou une collection d'objjets à insérer dans les faits dans le `WorkingMemory`.

L'élément `<retractObjects>` d'un autre côté spécifie les expressions EL qui retourne un objet ou une collection d'objets à être extrait depuis le `WorkingMemory`.

Il y a aussi le support pour l'utilisation de Drool pour l'affectation des tâches jBPM

```
<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
>
```

Certains objets sont disponibles pour les règles comme Drools globals, dénommé le jBPM `Assignable`, comme `assignable` et un objet `Decision` de Seam, comme `décision`. Les règles qui gèrent les décisions devraient s'appeler `decision.setOutcome("result")` pour déterminer le résultat de la décision. Les règles qui réalise l'affectation devrait définir l'identifiant de l'acteur en utilisant `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
```

Note

Vous pouvez trouver beaucoup plus sur les Drools sur *http://www.drools.org*

Attention

Seam vient avec suffisant de dépendances de Drools pour implémenter quelques règles simples. Si vous voulez ajouter des fonctionnalités additionnelles à Drools vous devriez télécharger une distribution complète et ajouter des dépendances additionnelles selon les besoins.

Security

15.1. Overview

The Seam Security API provides a multitude of security-related features for your Seam-based application, covering such areas as:

- Authentication - an extensible, JAAS-based authentication layer that allows users to authenticate against any security provider.
- Identity Management - an API for managing a Seam application's users and roles at runtime.
- Authorization - an extremely comprehensive authorization framework, supporting user roles, persistent and rule-based permissions, and a pluggable permission resolver for easily implementing customised security logic.
- Permission Management - a set of built-in Seam components to allow easy management of an application's security policy.
- CAPTCHA support - to assist in the prevention of automated software/scripts abusing your Seam-based site.
- And much more

This chapter will cover each of these features in detail.

15.2. Disabling Security

In some situations it may be necessary to disable Seam Security, for instances during unit tests or because you are using a different approach to security, such as native JAAS. Simply call the static method `Identity.setSecurityEnabled(false)` to disable the security infrastructure. Of course, it's not very convenient to have to call a static method when you want to configure the application, so as an alternative you can control this setting in components.xml:

- Entity Security
- Hibernate Security Interceptor
- Seam Security Interceptor
- Page restrictions
- Servlet API security integration

Assuming you are planning to take advantage of what Seam Security has to offer, the rest of this chapter documents the plethora of options you have for giving your user an identity in the eyes of the security model (authentication) and locking down the application by establishing constraints

(authorization). Let's begin with the task of authentication since that's the foundation of any security model.

15.3. Authentication

The authentication features provided by Seam Security are built upon JAAS (Java Authentication and Authorization Service), and as such provide a robust and highly configurable API for handling user authentication. However, for less complex authentication requirements Seam offers a much more simplified method of authentication that hides the complexity of JAAS.

15.3.1. Configuring an Authenticator component

Note

If you use Seam's Identity Management features (discussed later in this chapter) then it is not necessary to create an authenticator component (and you can skip this section).

The simplified authentication method provided by Seam uses a built-in JAAS login module, `SeamLoginModule`, which delegates authentication to one of your own Seam components. This login module is already configured inside Seam as part of a default application policy and as such does not require any additional configuration files. It allows you to write an authentication method using the entity classes that are provided by your own application, or alternatively to authenticate with some other third party provider. Configuring this simplified form of authentication requires the `identity` component to be configured in `components.xml`:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core"
            xmlns:security="http://jboss.com/products/seam/security"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation=
                "http://jboss.com/products/seam/components http://jboss.com/products/seam/
components-2.2.xsd
                 http://jboss.com/products/seam/security http://jboss.com/products/seam/security-
2.2.xsd">

   <security:identity authenticate-method="#{authenticator.authenticate}"/>

</components>
```

The EL expression `#{authenticator.authenticate}` is a method binding that indicates the `authenticate` method of the `authenticator` component will be used to authenticate the user.

15.3.2. Writing an authentication method

The `authenticate-method` property specified for `identity` in `components.xml` specifies which method will be used by `SeamLoginModule` to authenticate users. This method takes no parameters, and is expected to return a boolean, which indicates whether authentication is successful or not. The user's username and password can be obtained from `Credentials.getUsername()` and `Credentials.getPassword()`, respectively (you can get a reference to the `credentials` component via `Identity.instance().getCredentials()`). Any roles that the user is a member of should be assigned using `Identity.addRole()`. Here's a complete example of an authentication method inside a POJO component:

```
@Name("authenticator")
public class Authenticator {
  @In EntityManager entityManager;
  @In Credentials credentials;
  @In Identity identity;

  public boolean authenticate() {
    try {
      User user = (User) entityManager.createQuery(
        "from User where username = :username and password = :password")
        .setParameter("username", credentials.getUsername())
        .setParameter("password", credentials.getPassword())
        .getSingleResult();

      if (user.getRoles() != null) {
        for (UserRole mr : user.getRoles())
          identity.addRole(mr.getName());
      }

      return true;
    }
    catch (NoResultException ex) {
      return false;
    }

  }

}
```

In the above example, both `User` and `UserRole` are application-specific entity beans. The `roles` parameter is populated with the roles that the user is a member of, which should be added to the `Set` as literal string values, e.g. "admin", "user". In this case, if the user record is not

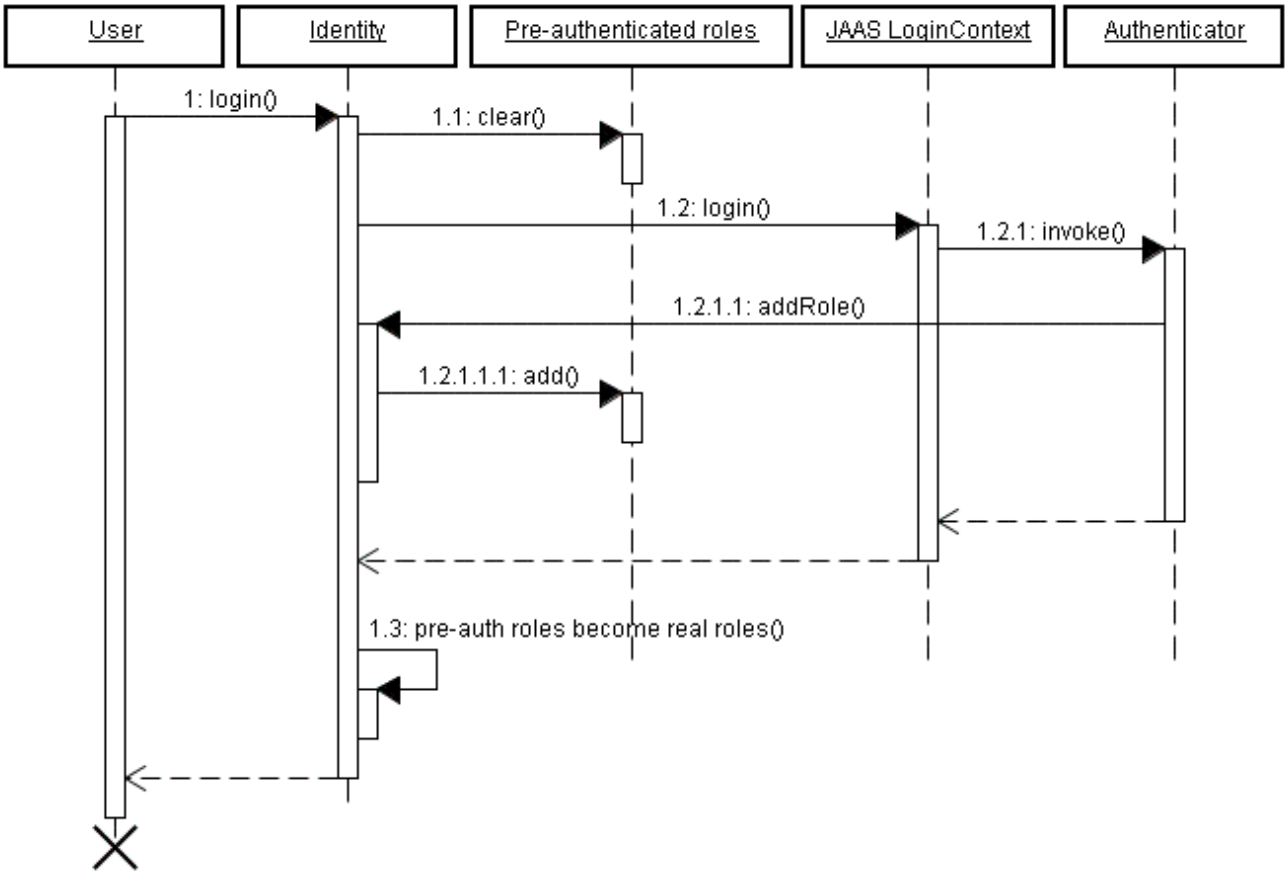
found and a `NoResultException` thrown, the authentication method returns `false` to indicate the authentication failed.

Astuce

When writing an authenticator method, it is important that it is kept minimal and free from any side-effects. This is because there is no guarantee as to how many times the authenticator method will be called by the security API, and as such it may be invoked multiple times during a single request. Because of this, any special code that should execute upon a successful or failed authentication should be written by implementing an event observer. See the section on Security Events further down in this chapter for more information about which events are raised by Seam Security.

15.3.2.1. Identity.addRole()

The `Identity.addRole()` method behaves differently depending on whether the current session is authenticated or not. If the session is not authenticated, then `addRole()` should *only* be called during the authentication process. When called here, the role name is placed into a temporary list of pre-authenticated roles. Once authentication is successful, the pre-authenticated roles then become "real" roles, and calling `Identity.hasRole()` for those roles will then return true. The following sequence diagram represents the list of pre-authenticated roles as a first class object to show more clearly how it fits in to the authentication process.



If the current session is already authenticated, then calling `Identity.addRole()` will have the expected effect of immediately granting the specified role to the current user.

15.3.2.2. Writing an event observer for security-related events

Say for example, that upon a successful login that some user statistics must be updated. This would be done by writing an event observer for the `org.jboss.seam.security.loginSuccessful` event, like this:

```
@In UserStats userStats;

@Observer("org.jboss.seam.security.loginSuccessful")
public void updateUserStats()
{
  userStats.setLastLoginDate(new Date());
  userStats.incrementLoginCount();
}
```

This observer method can be placed anywhere, even in the Authenticator component itself. You can find more information about security-related events later in this chapter.

15.3.3. Writing a login form

The `credentials` component provides both `username` and `password` properties, catering for the most common authentication scenario. These properties can be bound directly to the username and password fields on a login form. Once these properties are set, calling `identity.login()` will authenticate the user using the provided credentials. Here's an example of a simple login form:

```
<div>
   <h:outputLabel for="name" value="Username"/>
   <h:inputText id="name" value="#{credentials.username}"/>
</div>

<div>
   <h:outputLabel for="password" value="Password"/>
   <h:inputSecret id="password" value="#{credentials.password}"/>
</div>

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

Similarly, logging out the user is done by calling `#{identity.logout}`. Calling this action will clear the security state of the currently authenticated user, and invalidate the user's session.

15.3.4. Configuration Summary

So to sum up, there are the three easy steps to configure authentication:

- Configure an authentication method in `components.xml`.
- Write an authentication method.
- Write a login form so that the user can authenticate.

15.3.5. Remember Me

Seam Security supports the same kind of "Remember Me" functionality that is commonly encountered in many online web-based applications. It is actually supported in two different "flavours", or modes - the first mode allows the username to be stored in the user's browser as a cookie, and leaves the entering of the password up to the browser (many modern browsers are capable of remembering passwords).

The second mode supports the storing of a unique token in a cookie, and allows a user to authenticate automatically upon returning to the site, without having to provide a password.

Avertissement

Automatic client authentication with a persistent cookie stored on the client machine is dangerous. While convenient for users, any cross-site scripting security hole in your website would have dramatically more serious effects than usual. Without the authentication cookie, the only cookie to steal for an attacker with XSS is the cookie of the current session of a user. This means the attack only works when the user has an open session - which should be a short timespan. However, it is much more attractive and dangerous if an attacker has the possibility to steal a persistent Remember Me cookie that allows him to login without authentication, at any time. Note that this all depends on how well you protect your website against XSS attacks - it's up to you to make sure that your website is 100% XSS safe - a non-trival achievement for any website that allows user input to be rendered on a page.

Browser vendors recognized this issue and introduced a "Remember Passwords" feature - today almost all browsers support this. Here, the browser remembers the login username and password for a particular website and domain, and fills out the login form automatically when you don't have an active session with the website. If you as a website designer then offer a convenient login keyboard shortcut, this approach is almost as convenient as a "Remember Me" cookie and much safer. Some browsers (e.g. Safari on OS X) even store the login form data in the encrypted global operation system keychain. Or, in a networked environment, the keychain can be transported with the user (between laptop and desktop for example), while browser cookies are usually not synchronized.

To summarize: While everyone is doing it, persistent "Remember Me" cookies with automatic authentication are a bad practice and should not be used. Cookies that "remember" only the users login name, and fill out the login form with that username as a convenience, are not an issue.

To enable the remember me feature for the default (safe, username only) mode, no special configuration is required. In your login form, simply bind the remember me checkbox to `rememberMe.enabled`, like in the following example:

```
<div>
  <h:outputLabel for="name" value="User name"/>
  <h:inputText id="name" value="#{credentials.username}"/>
</div>

<div>
  <h:outputLabel for="password" value="Password"/>
  <h:inputSecret id="password" value="#{credentials.password}" redisplay="true"/>
```

```
</div>

<div class="loginRow">
  <h:outputLabel for="rememberMe" value="Remember me"/>
  <h:selectBooleanCheckbox id="rememberMe" value="#{rememberMe.enabled}"/>
</div>
```

15.3.5.1. Token-based Remember-me Authentication

To use the automatic, token-based mode of the remember me feature, you must first configure a token store. The most common scenario is to store these authentication tokens within a database (which Seam supports), however it is possible to implement your own token store by implementing the `org.jboss.seam.security.TokenStore` interface. This section will assume you will be using the provided `JpaTokenStore` implementation to store authentication tokens inside a database table.

The first step is to create a new Entity which will contain the tokens. The following example shows a possible structure that you may use:

```
@Entity
public class AuthenticationToken implements Serializable {
  private Integer tokenId;
  private String username;
  private String value;

  @Id @GeneratedValue
  public Integer getTokenId() {
    return tokenId;
  }

  public void setTokenId(Integer tokenId) {
    this.tokenId = tokenId;
  }

  @TokenUsername
  public String getUsername() {
    return username;
  }

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

  @TokenValue
```

```
  public String getValue() {
    return value;
  }

  public void setValue(String value) {
    this.value = value;
  }
}
```

As you can see from this listing, a couple of special annotations, `@TokenUsername` and `@TokenValue` are used to configure the username and token properties of the entity. These annotations are required for the entity that will contain the authentication tokens.

The next step is to configure `JpaTokenStore` to use this entity bean to store and retrieve authentication tokens. This is done in `components.xml` by specifying the `token-class` attribute:

```
<security:jpa-token-store token-class="org.jboss.seam.example.seamspace.AuthenticationToken" />
```

Once this is done, the last thing to do is to configure the `RememberMe` component in `components.xml` also. Its `mode` should be set to `autoLogin`:

```
<security:remember-me mode="autoLogin"/>
```

That is all that is required - automatic authentication will now occur for users revisiting your site (as long as they check the "remember me" checkbox).

To ensure that users are automatically authenticated when returning to the site, the following section should be placed in components.xml:

```
<event type="org.jboss.seam.security.notLoggedIn">
   <action execute="#{redirect.captureCurrentView}"/>
   <action execute="#{identity.tryLogin()}"/>
 </event>
 <event type="org.jboss.seam.security.loginSuccessful">
   <action execute="#{redirect.returnToCapturedView}"/>
 </event
```

```
>
```

15.3.6. Handling Security Exceptions

To prevent users from receiving the default error page in response to a security error, it's recommended that `pages.xml` is configured to redirect security errors to a more "pretty" page. The two main types of exceptions thrown by the security API are:

- `NotLoggedInException` - This exception is thrown if the user attempts to access a restricted action or page when they are not logged in.
- `AuthorizationException` - This exception is only thrown if the user is already logged in, and they have attempted to access a restricted action or page for which they do not have the necessary privileges.

In the case of a `NotLoggedInException`, it is recommended that the user is redirected to either a login or registration page so that they can log in. For an `AuthorizationException`, it may be useful to redirect the user to an error page. Here's an example of a `pages.xml` file that redirects both of these security exceptions:

```
<pages>

   ...

   <exception class="org.jboss.seam.security.NotLoggedInException">
      <redirect view-id="/login.xhtml">
         <message>You must be logged in to perform this action</message>
      </redirect>
   </exception>

   <exception class="org.jboss.seam.security.AuthorizationException">
      <end-conversation/>
      <redirect view-id="/security_error.xhtml">
         <message>You do not have the necessary security privileges to perform this action.</
message>
      </redirect>
   </exception>

</pages>
```

Most web applications require even more sophisticated handling of login redirection, so Seam includes some special functionality for handling this problem.

15.3.7. Login Redirection

You can ask Seam to redirect the user to a login screen when an unauthenticated user tries to access a particular view (or wildcarded view id) as follows:

```
<pages login-view-id="/login.xhtml">

   <page view-id="/members/*" login-required="true"/>

   ...

</pages>
```

Astuce

This is less of a blunt instrument than the exception handler shown above, but should probably be used in conjunction with it.

After the user logs in, we want to automatically send them back where they came from, so they can retry the action that required logging in. If you add the following event listeners to `components.xml`, attempts to access a restricted view while not logged in will be remembered, so that upon the user successfully logging in they will be redirected to the originally requested view, with any page parameters that existed in the original request.

```
<event type="org.jboss.seam.security.notLoggedIn">
   <action execute="#{redirect.captureCurrentView}"/>
</event>

<event type="org.jboss.seam.security.postAuthenticate">
   <action execute="#{redirect.returnToCapturedView}"/>
</event>
```

Note that login redirection is implemented as a conversation-scoped mechanism, so don't end the conversation in your `authenticate()` method.

15.3.8. HTTP Authentication

Although not recommended for use unless absolutely necessary, Seam provides means for authenticating using either HTTP Basic or HTTP Digest (RFC 2617) methods. To use either form of authentication, the `authentication-filter` component must be enabled in components.xml:

```
<web:authentication-filter url-pattern="*.seam" auth-type="basic"/>
```

To enable the filter for basic authentication, set `auth-type` to `basic`, or for digest authentication, set it to `digest`. If using digest authentication, the `key` and `realm` must also be set:

```
<web:authentication-filter url-pattern="*.seam" auth-type="digest" key="AA3JK34aSDlkj"
realm="My App"/>
```

The `key` can be any String value. The `realm` is the name of the authentication realm that is presented to the user when they authenticate.

15.3.8.1. Writing a Digest Authenticator

If using digest authentication, your authenticator class should extend the abstract class `org.jboss.seam.security.digest.DigestAuthenticator`, and use the `validatePassword()` method to validate the user's plain text password against the digest request. Here is an example:

```
public boolean authenticate()
{
  try
  {
    User user = (User) entityManager.createQuery(
      "from User where username = :username")
      .setParameter("username", identity.getUsername())
      .getSingleResult();

    return validatePassword(user.getPassword());
  }
  catch (NoResultException ex)
  {
    return false;
  }
}
```

15.3.9. Advanced Authentication Features

This section explores some of the advanced features provided by the security API for addressing more complex security requirements.

15.3.9.1. Using your container's JAAS configuration

If you would rather not use the simplified JAAS configuration provided by the Seam Security API, you may instead delegate to the default system JAAS configuration by providing a `jaas-config-name` property in `components.xml`. For example, if you are using JBoss AS and wish to use the `other` policy (which uses the `UsersRolesLoginModule` login module provided by JBoss AS), then the entry in `components.xml` would look like this:

```
<security:identity jaas-config-name="other"/>
```

Please keep in mind that doing this does not mean that your user will be authenticated in whichever container your Seam application is deployed in. It merely instructs Seam Security to authenticate itself using the configured JAAS security policy.

15.4. Identity Management

Identity Management provides a standard API for the management of a Seam application's users and roles, regardless of which identity store (database, LDAP, etc) is used on the backend. At the center of the Identity Management API is the `identityManager` component, which provides all the methods for creating, modifying and deleting users, granting and revoking roles, changing passwords, enabling and disabling user accounts, authenticating users and listing users and roles.

Before it may be used, the `identityManager` must first be configured with one or more `IdentityStores`. These components do the actual work of interacting with the backend security provider, whether it be a database, LDAP server, or something else.

15.4.1. Configuring IdentityManager

The `identityManager` component allows for separate identity stores to be configured for authentication and authorization operations. This means that it is possible for users to be

authenticated against one identity store, for example an LDAP directory, yet have their roles loaded from another identity store, such as a relational database.

Seam provides two `IdentityStore` implementations out of the box; `JpaIdentityStore` uses a relational database to store user and role information, and is the default identity store that is used if nothing is explicitly configured in the `identityManager` component. The other implementation that is provided is `LdapIdentityStore`, which uses an LDAP directory to store users and roles.

There are two configurable properties for the `identityManager` component - `identityStore` and `roleIdentityStore`. The value for these properties must be an EL expression referring to a Seam component implementing the `IdentityStore` interface. As already mentioned, if left unconfigured then `JpaIdentityStore` will be assumed by default. If only the `identityStore` property is configured, then the same value will be used for `roleIdentityStore` also. For example, the following entry in `components.xml` will configure `identityManager` to use an `LdapIdentityStore` for both user-related and role-related operations:

```
<security:identity-manager identity-store="#{ldapIdentityStore}"/>
```

The following example configures `identityManager` to use an `LdapIdentityStore` for user-related operations, and `JpaIdentityStore` for role-related operations:

```
<security:identity-manager
  identity-store="#{ldapIdentityStore}"
  role-identity-store="#{jpaIdentityStore}"/>
```

The following sections explain both of these identity store implementations in greater detail.

15.4.2. JpaIdentityStore

This identity store allows for users and roles to be stored inside a relational database. It is designed to be as unrestrictive as possible in regards to database schema design, allowing a great deal of flexibility in the underlying table structure. This is achieved through the use of a set of special annotations, allowing entity beans to be configured to store user and role records.

15.4.2.1. Configuring JpaIdentityStore

`JpaIdentityStore` requires that both the `user-class` and `role-class` properties are configured. These properties should refer to the entity classes that are to be used to store both user and role records, respectively. The following example shows the configuration from `components.xml` in the SeamSpace example:

```
<security:jpa-identity-store
  user-class="org.jboss.seam.example.seamspace.MemberAccount"
  role-class="org.jboss.seam.example.seamspace.MemberRole"/>
```

15.4.2.2. Configuring the Entities

As already mentioned, a set of special annotations are used to configure entity beans for storing users and roles. The following table lists each of the annotations, and their descriptions.

Tableau 15.1. User Entity Annotations

Annotation	Status	Description
`@UserPrincipal`	Required	This annotation marks the field or method containing the user's username.

`@UserPassword`	Required	This annotation marks the field or method containing the user's password. It allows a `hash` algorithm to be specified for password hashing. Possible values for `hash` are `md5`, `sha` and `none`. E.g:
		`@UserPassword (hash="md5")`
		`public String getPasswordHash() {`
		`  return passwordHash;`
		`}`
		If an application requires a hash algorithm that isn't supported natively by Seam, it is possible to extend the `PasswordHash` component to implement other hashing algorithms.

`@UserFirstName`	Optional	This annotation marks the field or method containing the user's first name.
`@UserLastName`	Optional	This annotation marks the field or method containing the user's last name.
`@UserEnabled`	Optional	This annotation marks the field or method containing the enabled status of the user. This should be a boolean property, and if not present then all user accounts are assumed to be enabled.
`@UserRoles`	Required	

Annotation	Status	Description
		This annotation marks the field or method containing the roles of the user. This property will be described in more detail further down.

Tableau 15.2. Role Entity Annotations

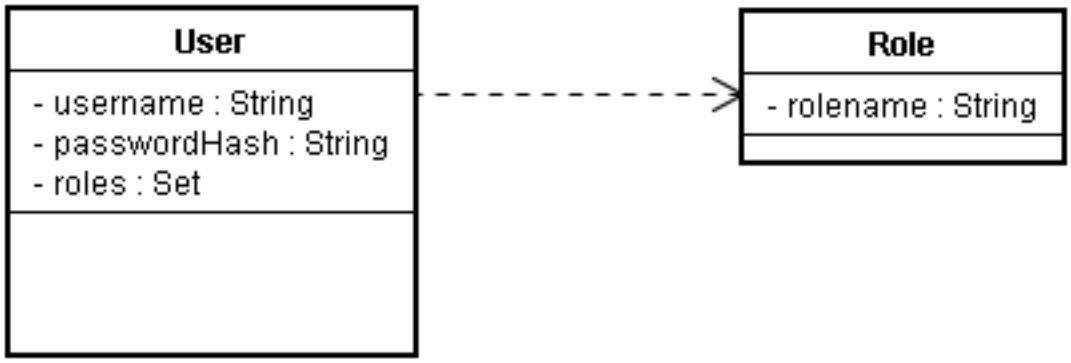
Annotation	Status	Description
`@RoleName`	Required	This annotation marks the field or method containing the name of the role.
`@RoleGroups`	Optional	This annotation marks the field or method containing the group memberships of the role.
`@RoleConditional`	Optional	This annotation marks the field or method indicating whether the role is conditional or not. Conditional roles are explained later in this chapter.

15.4.2.3. Entity Bean Examples

As mentioned previously, `JpaIdentityStore` is designed to be as flexible as possible when it comes to the database schema design of your user and role tables. This section looks at a number of possible database schemas that can be used to store user and role records.

15.4.2.3.1. Minimal schema example

In this bare minimal example, a simple user and role table are linked via a many-to-many relationship using a cross-reference table named `UserRoles`.



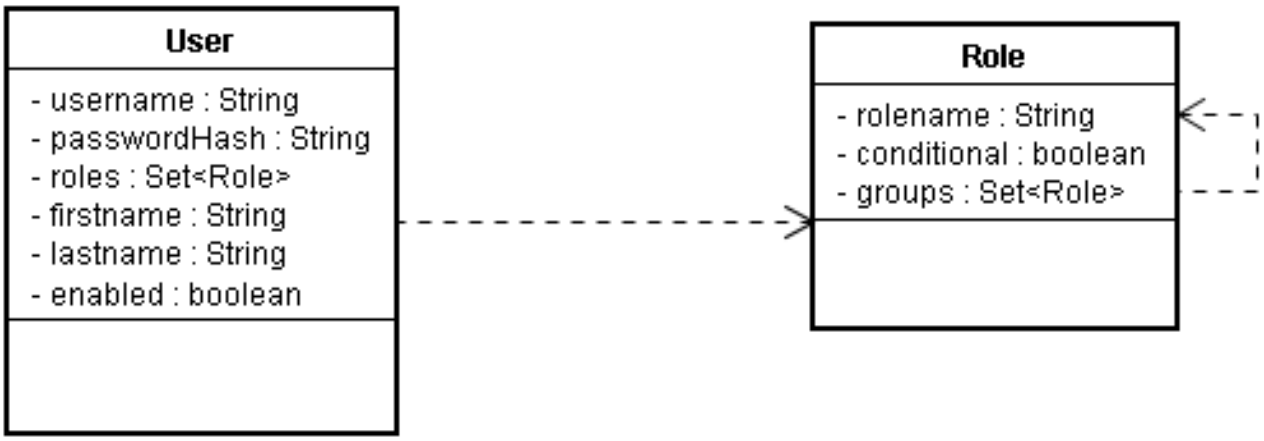

```
@Entity
public class User {
  private Integer userId;
  private String username;
  private String passwordHash;
```

```
  private Set<Role> roles;

  @Id @GeneratedValue
  public Integer getUserId() { return userId; }
  public void setUserId(Integer userId) { this.userId = userId; }

  @UserPrincipal
  public String getUsername() { return username; }
  public void setUsername(String username) { this.username = username; }

  @UserPassword(hash = "md5")
  public String getPasswordHash() { return passwordHash; }
  public void setPasswordHash(String passwordHash) { this.passwordHash = passwordHash; }

  @UserRoles
  @ManyToMany(targetEntity = Role.class)
  @JoinTable(name = "UserRoles",
    joinColumns = @JoinColumn(name = "UserId"),
    inverseJoinColumns = @JoinColumn(name = "RoleId"))
  public Set<Role> getRoles() { return roles; }
  public void setRoles(Set<Role> roles) { this.roles = roles; }
}
```

```
@Entity
public class Role {
  private Integer roleId;
  private String rolename;

  @Id @Generated
  public Integer getRoleId() { return roleId; }
  public void setRoleId(Integer roleId) { this.roleId = roleId; }

  @RoleName
  public String getRolename() { return rolename; }
  public void setRolename(String rolename) { this.rolename = rolename; }
}
```

15.4.2.3.2. Complex Schema Example

This example builds on the above minimal example by including all of the optional fields, and allowing group memberships for roles.




```
@Entity
public class User {
  private Integer userId;
  private String username;
  private String passwordHash;
  private Set<Role> roles;
  private String firstname;
  private String lastname;
  private boolean enabled;

  @Id @GeneratedValue
  public Integer getUserId() { return userId; }
  public void setUserId(Integer userId) { this.userId = userId; }

  @UserPrincipal
  public String getUsername() { return username; }
  public void setUsername(String username) { this.username = username; }

  @UserPassword(hash = "md5")
  public String getPasswordHash() { return passwordHash; }
  public void setPasswordHash(String passwordHash) { this.passwordHash = passwordHash; }

  @UserFirstName
  public String getFirstname() { return firstname; }
  public void setFirstname(String firstname) { this.firstname = firstname; }

  @UserLastName
  public String getLastname() { return lastname; }
  public void setLastname(String lastname) { this.lastname = lastname; }
```

```
  @UserEnabled
  public boolean isEnabled() { return enabled; }
  public void setEnabled(boolean enabled) { this.enabled = enabled; }

  @UserRoles
  @ManyToMany(targetEntity = Role.class)
  @JoinTable(name = "UserRoles",
    joinColumns = @JoinColumn(name = "UserId"),
    inverseJoinColumns = @JoinColumn(name = "RoleId"))
  public Set<Role> getRoles() { return roles; }
  public void setRoles(Set<Role> roles) { this.roles = roles; }
}
```

```
@Entity
public class Role {
  private Integer roleId;
  private String rolename;
  private boolean conditional;

  @Id @Generated
  public Integer getRoleId() { return roleId; }
  public void setRoleId(Integer roleId) { this.roleId = roleId; }

  @RoleName
  public String getRolename() { return rolename; }
  public void setRolename(String rolename) { this.rolename = rolename; }

  @RoleConditional
  public boolean isConditional() { return conditional; }
  public void setConditional(boolean conditional) { this.conditional = conditional; }

  @RoleGroups
  @ManyToMany(targetEntity = Role.class)
  @JoinTable(name = "RoleGroups",
    joinColumns = @JoinColumn(name = "RoleId"),
    inverseJoinColumns = @JoinColumn(name = "GroupId"))
  public Set<Role> getGroups() { return groups; }
  public void setGroups(Set<Role> groups) { this.groups = groups; }

}
```

15.4.2.4. JpaIdentityStore Events

When using `JpaIdentityStore` as the identity store implementation with `IdentityManager`, a few events are raised as a result of invoking certain `IdentityManager` methods.

15.4.2.4.1. JpaIdentityStore.EVENT_PRE_PERSIST_USER

This event is raised in response to calling `IdentityManager.createUser()`. Just before the user entity is persisted to the database, this event will be raised passing the entity instance as an event parameter. The entity will be an instance of the `user-class` configured for `JpaIdentityStore`.

Writing an observer for this event may be useful for setting additional field values on the entity, which aren't set as part of the standard `createUser()` functionality.

15.4.2.4.2. JpaIdentityStore.EVENT_USER_CREATED

This event is also raised in response to calling `IdentityManager.createUser()`. However, it is raised after the user entity has already been persisted to the database. Like the `EVENT_PRE_PERSIST_USER` event, it also passes the entity instance as an event parameter. It may be useful to observe this event if you also need to persist other entities that reference the user entity, for example contact detail records or other user-specific data.

15.4.2.4.3. JpaIdentityStore.EVENT_USER_AUTHENTICATED

This event is raised when calling `IdentityManager.authenticate()`. It passes the user entity instance as the event parameter, and is useful for reading additional properties from the user entity that is being authenticated.

15.4.3. LdapIdentityStore

This identity store implementation is designed for working with user records stored in an LDAP directory. It is very highly configurable, allowing great flexibility in how both users and roles are stored in the directory. The following sections describe the configuration options for this identity store, and provide some configuration examples.

15.4.3.1. Configuring LdapIdentityStore

The following table describes the available properties that can be configured in `components.xml` for `LdapIdentityStore`.

Tableau 15.3. LdapIdentityStore Configuration Properties

Property	Default Value	Description
`server-address`	`localhost`	The address of the LDAP server.
`server-port`	`389`	The port number that the LDAP server is listening on.

Property	Default Value	Description
`user-context-DN`	`ou=Person,dc=acme,dc=com`	The Distinguished Name (DN) of the context containing user records.
`user-DN-prefix`	`uid=`	This value is prefixed to the front of the username to locate the user's record.
`user-DN-suffix`	`,ou=Person,dc=acme,dc=com`	This value is appended to the end of the username to locate the user's record.
`role-context-DN`	`ou=Role,dc=acme,dc=com`	The DN of the context containing role records.
`role-DN-prefix`	`cn=`	This value is prefixed to the front of the role name to form the DN for locating the role record.
`role-DN-suffix`	`,ou=Roles,dc=acme,dc=com`	This value is appended to the role name to form the DN for locating the role record.
`bind-DN`	`cn=Manager,dc=acme,dc=com`	This is the context used to bind to the LDAP server.
`bind-credentials`	`secret`	These are the credentials (the password) used to bind to the LDAP server.
`user-role-attribute`	`roles`	This is the name of the attribute of the user record that contains the list of roles that the user is a member of.

Property	Default Value	Description
role-attribute-is-DN	true	This boolean property indicates whether the role attribute of the user record is itself a distinguished name.
user-name-attribute	uid	Indicates which attribute of the user record contains the username.
user-password-attribute	userPassword	Indicates which attribute of the user record contains the user's password.
first-name-attribute	null	Indicates which attribute of the user record contains the user's first name.
last-name-attribute	sn	Indicates which attribute of the user record contains the user's last name.
full-name-attribute	cn	Indicates which attribute of the user record contains the user's full (common) name.
enabled-attribute	null	Indicates which attribute of the user record determines whether the user is enabled.
role-name-attribute	cn	Indicates which attribute of the role record contains the name of the role.
object-class-attribute	objectClass	Indicates which attribute determines the class of an object in the directory.

Property	Default Value	Description
`role-object-classes`	`organizationalRole`	An array of the object classes that new role records should be created as.
`user-object-classes`	`person,uidObject`	An array of the object classes that new user records should be created as.
`security-authentication-type`	`simple`	The security level to use. Possible values are "none", "simple" and "strong".

15.4.3.2. LdapIdentityStore Configuration Example

The following configuration example shows how `LdapIdentityStore` may be configured for an LDAP directory running on fictional host `directory.mycompany.com`. The users are stored within this directory under the context `ou=Person,dc=mycompany,dc=com`, and are identified using the `uid` attribute (which corresponds to their username). Roles are stored in their own context, `ou=Roles,dc=mycompany,dc=com` and referenced from the user's entry via the `roles` attribute. Role entries are identified by their common name (the `cn` attribute) , which corresponds to the role name. In this example, users may be disabled by setting the value of their `enabled` attribute to false.

```
<security:ldap-identity-store
  server-address="directory.mycompany.com"
  bind-DN="cn=Manager,dc=mycompany,dc=com"
  bind-credentials="secret"
  user-DN-prefix="uid="
  user-DN-suffix=",ou=Person,dc=mycompany,dc=com"
  role-DN-prefix="cn="
  role-DN-suffix=",ou=Roles,dc=mycompany,dc=com"
  user-context-DN="ou=Person,dc=mycompany,dc=com"
  role-context-DN="ou=Roles,dc=mycompany,dc=com"
  user-role-attribute="roles"
  role-name-attribute="cn"
  user-object-classes="person,uidObject"
  enabled-attribute="enabled"
  />
```

15.4.4. Writing your own IdentityStore

Writing your own identity store implementation allows you to authenticate and perform identity management operations against security providers that aren't supported out of the box by Seam. Only a single class is required to achieve this, and it must implement the `org.jboss.seam.security.management.IdentityStore` interface.

Please refer to the JavaDoc for `IdentityStore` for a description of the methods that must be implemented.

15.4.5. Authentication with Identity Management

If you are using the Identity Management features in your Seam application, then it is not required to provide an authenticator component (see previous Authentication section) to enable authentication. Simply omit the `authenticate-method` from the `identity` configuration in `components.xml`, and the `SeamLoginModule` will by default use `IdentityManager` to authenticate your application's users, without any special configuration required.

15.4.6. Using IdentityManager

The `IdentityManager` can be accessed either by injecting it into your Seam component as follows:

```
@In IdentityManager identityManager;
```

or by accessing it through its static `instance()` method:

```
IdentityManager identityManager = IdentityManager.instance();
```

The following table describes `IdentityManager`'s API methods:

Tableau 15.4. Identity Management API

Method	Returns	Description
`createUser(String name, String password)`	`boolean`	Creates a new user account, with the specified name and password. Returns `true` if successful, or `false` if not.
`deleteUser(String name)`	`boolean`	Deletes the user account with the specified name.

Method	Returns	Description
		Returns `true` if successful, or `false` if not.
`createRole(String role)`	`boolean`	Creates a new role, with the specified name. Returns `true` if successful, or `false` if not.
`deleteRole(String name)`	`boolean`	Deletes the role with the specified name. Returns `true` if successful, or `false` if not.
`enableUser(String name)`	`boolean`	Enables the user account with the specified name. Accounts that are not enabled are not able to authenticate. Returns `true` if successful, or `false` if not.
`disableUser(String name)`	`boolean`	Disables the user account with the specified name. Returns `true` if successful, or `false` if not.
`changePassword(String name, String password)`	`boolean`	Changes the password for the user account with the specified name. Returns `true` if successful, or `false` if not.
`isUserEnabled(String name)`	`boolean`	Returns `true` if the specified user account is enabled, or `false` if it isn't.
`grantRole(String name, String role)`	`boolean`	Grants the specified role to the specified

Method	Returns	Description
		user or role. The role must already exist for it to be granted. Returns `true` if the role is successfully granted, or `false` if it is already granted to the user.
`revokeRole(String name, String role)`	`boolean`	Revokes the specified role from the specified user or role. Returns `true` if the specified user is a member of the role and it is successfully revoked, or `false` if the user is not a member of the role.
`userExists(String name)`	`boolean`	Returns `true` if the specified user exists, or `false` if it doesn't.
`listUsers()`	`List`	Returns a list of all user names, sorted in alpha-numeric order.
`listUsers(String filter)`	`List`	Returns a list of all user names filtered by the specified filter parameter, sorted in alpha-numeric order.
`listRoles()`	`List`	Returns a list of all role names.
`getGrantedRoles(String name)`	`List`	Returns a list of the names of all the roles explicitly granted to the specified user name.
`getImpliedRoles(String name)`	`List`	Returns a list of the names of all the roles implicitly granted to the specified user name. Implicitly

Method	Returns	Description
		granted roles include those that are not directly granted to a user, rather they are granted to the roles that the user is a member of. For example, is the `admin` role is a member of the `user` role, and a user is a member of the `admin` role, then the implied roles for the user are both the `admin`, and `user` roles.
`authenticate(String name, String password)`	`boolean`	Authenticates the specified username and password using the configured Identity Store. Returns `true` if successful or `false` if authentication failed. Successful authentication implies nothing beyond the return value of the method. It does not change the state of the `Identity` component - to perform a proper Seam login the `Identity.login()` must be used instead.
`addRoleToGroup(String role, String group)`	`boolean`	Adds the specified role as a member of the specified group. Returns true if the operation is successful.
`removeRoleFromGroup(String role, String group)`	`boolean`	Removes the specified role from

Method	Returns	Description
		the specified group. Returns true if the operation is successful.
`listRoles()`	`List`	Lists the names of all roles.

Using the Identity Management API requires that the calling user has the appropriate authorization to invoke its methods. The following table describes the permission requirements for each of the methods in `IdentityManager`. The permission targets listed below are literal String values.

Tableau 15.5. Identity Management Security Permissions

Method	Permission Target	Permission Action
`createUser()`	`seam.user`	`create`
`deleteUser()`	`seam.user`	`delete`
`createRole()`	`seam.role`	`create`
`deleteRole()`	`seam.role`	`delete`
`enableUser()`	`seam.user`	`update`
`disableUser()`	`seam.user`	`update`
`changePassword()`	`seam.user`	`update`
`isUserEnabled()`	`seam.user`	`read`
`grantRole()`	`seam.user`	`update`
`revokeRole()`	`seam.user`	`update`
`userExists()`	`seam.user`	`read`
`listUsers()`	`seam.user`	`read`
`listRoles()`	`seam.role`	`read`
`addRoleToGroup()`	`seam.role`	`update`
`removeRoleFromGroup()`	`seam.role`	`update`

The following code listing provides an example set of security rules that grants access to all Identity Management-related methods to members of the `admin` role:

```
rule ManageUsers
  no-loop
  activation-group "permissions"
when
  check: PermissionCheck(name == "seam.user", granted == false)
```

```
  Role(name == "admin")
then
  check.grant();
end

rule ManageRoles
  no-loop
  activation-group "permissions"
when
  check: PermissionCheck(name == "seam.role", granted == false)
  Role(name == "admin")
then
  check.grant();
end
```

15.5. Error Messages

The security API produces a number of default faces messages for various security-related events. The following table lists the message keys that can be used to override these messages by specifying them in a `message.properties` resource file. To suppress the message, just put the key with an empty value in the resource file.

Tableau 15.6. Security Message Keys

Message Key	Description
`org.jboss.seam.loginSuccessful`	This message is produced when a user successfully logs in via the security API.
`org.jboss.seam.loginFailed`	This message is produced when the login process fails, either because the user provided an incorrect username or password, or because authentication failed in some other way.
`org.jboss.seam.NotLoggedIn`	This message is produced when a user attempts to perform an action or access a page that requires a security check, and the user is not currently authenticated.
`org.jboss.seam.AlreadyLoggedIn`	This message is produced when a user that is already authenticated attempts to log in again.

15.6. Authorization

There are a number of authorization mechanisms provided by the Seam Security API for securing access to components, component methods, and pages. This section describes each of these. An important thing to note is that if you wish to use any of the advanced features (such as rule-based

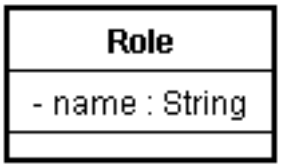
permissions) then your `components.xml` may need to be configured to support this - see the Configuration section above.

15.6.1. Core concepts

Seam Security is built around the premise of users being granted roles and/or permissions, allowing them to perform operations that may not otherwise be permissible for users without the necessary security privileges. Each of the authorization mechanisms provided by the Seam Security API are built upon this core concept of roles and permissions, with an extensible framework providing multiple ways to secure application resources.

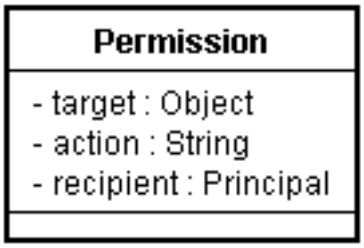
15.6.1.1. What is a role?

A role is a *group*, or *type*, of user that may have been granted certain privileges for performing one or more specific actions within an application. They are simple constructs, consisting of just a name such as "admin", "user", "customer", etc. They can be granted either to users (or in some cases to other roles), and are used to create logical groups of users for the convenient assignment of specific application privileges.



15.6.1.2. What is a permission?

A permission is a privilege (sometimes once-off) for performing a single, specific action. It is entirely possible to build an application using nothing but permissions, however roles offer a higher level of convenience when granting privileges to groups of users. They are slightly more complex in structure than roles, essentially consisting of three "aspects"; a target, an action, and a recipient. The target of a permission is the object (or an arbitrary name or class) for which a particular action is allowed to be performed by a specific recipient (or user). For example, the user "Bob" may have permission to delete customer objects. In this case, the permission target may be "customer", the permission action would be "delete" and the recipient would be "Bob".



Within this documentation, permissions are generally represented in the form `target:action` (omitting the recipient, although in reality one is always required).

15.6.2. Securing components

Let's start by examining the simplest form of authorization, component security, starting with the `@Restrict` annotation.

@Restrict vs Typesafe security annotations

While using the `@Restrict` annotation provides a powerful and flexible method for security component methods due to its ability to support EL expressions, it is recommended that the typesafe equivalent (described later) be used, at least for the compile-time safety it provides.

15.6.2.1. The @Restrict annotation

Seam components may be secured either at the method or the class level, using the `@Restrict` annotation. If both a method and it's declaring class are annotated with `@Restrict`, the method restriction will take precedence (and the class restriction will not apply). If a method invocation fails a security check, then an exception will be thrown as per the contract for `Identity.checkRestriction()` (see Inline Restrictions). A `@Restrict` on just the component class itself is equivalent to adding `@Restrict` to each of its methods.

An empty `@Restrict` implies a permission check of `componentName:methodName`. Take for example the following component method:

```
@Name("account")
public class AccountAction {
   @Restrict public void delete() {
     ...
   }
}
```

In this example, the implied permission required to call the `delete()` method is `account:delete`. The equivalent of this would be to write `@Restrict("#{s:hasPermission('account','delete')}")`. Now let's look at another example:

```
@Restrict @Name("account")
public class AccountAction {
   public void insert() {
     ...
   }
   @Restrict("#{s:hasRole('admin')}")
```

```
    public void delete() {
      ...
    }
}
```

This time, the component class itself is annotated with `@Restrict`. This means that any methods without an overriding `@Restrict` annotation require an implicit permission check. In the case of this example, the `insert()` method requires a permission of `account:insert`, while the `delete()` method requires that the user is a member of the `admin` role.

Before we go any further, let's address the `#{s:hasRole()}` expression seen in the above example. Both `s:hasRole` and `s:hasPermission` are EL functions, which delegate to the correspondingly named methods of the `Identity` class. These functions can be used within any EL expression throughout the entirety of the security API.

Being an EL expression, the value of the `@Restrict` annotation may reference any objects that exist within a Seam context. This is extremely useful when performing permission checks for a specific object instance. Look at this example:

```
@Name("account")
public class AccountAction {
    @In Account selectedAccount;
    @Restrict("#{s:hasPermission(selectedAccount,'modify')}")
    public void modify() {
        selectedAccount.modify();
    }
}
```

The interesting thing to note from this example is the reference to `selectedAccount` seen within the `hasPermission()` function call. The value of this variable will be looked up from within the Seam context, and passed to the `hasPermission()` method in `Identity`, which in this case can then determine if the user has the required permission for modifying the specified `Account` object.

15.6.2.2. Inline restrictions

Sometimes it might be desirable to perform a security check in code, without using the `@Restrict` annotation. In this situation, simply use `Identity.checkRestriction()` to evaluate a security expression, like this:

```
public void deleteCustomer() {
    Identity.instance().checkRestriction("#{s:hasPermission(selectedCustomer,'delete')}");
}
```

If the expression specified doesn't evaluate to `true`, either

- if the user is not logged in, a `NotLoggedInException` exception is thrown or
- if the user is logged in, an `AuthorizationException` exception is thrown.

It is also possible to call the `hasRole()` and `hasPermission()` methods directly from Java code:

```
if (!Identity.instance().hasRole("admin"))
    throw new AuthorizationException("Must be admin to perform this action");

if (!Identity.instance().hasPermission("customer", "create"))
    throw new AuthorizationException("You may not create new customers");
```

15.6.3. Security in the user interface

One indication of a well designed user interface is that the user is not presented with options for which they don't have the necessary privileges to use. Seam Security allows conditional rendering of either 1) sections of a page or 2) individual controls, based upon the privileges of the user, using the very same EL expressions that are used for component security.

Let's take a look at some examples of interface security. First of all, let's pretend that we have a login form that should only be rendered if the user is not already logged in. Using the `identity.isLoggedIn()` property, we can write this:

```
<h:form class="loginForm" rendered="#{not identity.loggedIn}">
```

If the user isn't logged in, then the login form will be rendered - very straight forward so far. Now let's pretend there is a menu on the page that contains some actions which should only be accessible to users in the `manager` role. Here's one way that these could be written:

```
<h:outputLink action="#{reports.listManagerReports}" rendered="#{s:hasRole('manager')}">
   Manager Reports
</h:outputLink>
```

This is also quite straight forward. If the user is not a member of the `manager` role, then the outputLink will not be rendered. The `rendered` attribute can generally be used on the control itself, or on a surrounding `<s:div>` or `<s:span>` control.

Now for something more complex. Let's say you have a `h:dataTable` control on a page listing records for which you may or may not wish to render action links depending on the user's

privileges. The `s:hasPermission` EL function allows us to pass in an object parameter which can be used to determine whether the user has the requested permission for that object or not. Here's how a dataTable with secured links might look:

```
<h:dataTable value="#{clients}" var="cl">
   <h:column>
      <f:facet name="header"
>Name</f:facet>
      #{cl.name}
   </h:column>
   <h:column>
      <f:facet name="header"
>City</f:facet>
      #{cl.city}
   </h:column>
   <h:column>
      <f:facet name="header"
>Action</f:facet>
      <s:link value="Modify Client" action="#{clientAction.modify}"
            rendered="#{s:hasPermission(cl,'modify')}"/>
      <s:link value="Delete Client" action="#{clientAction.delete}"
            rendered="#{s:hasPermission(cl,'delete')}"/>
   </h:column>
</h:dataTable
>
```

15.6.4. Securing pages

Page security requires that the application is using a `pages.xml` file, however is extremely simple to configure. Simply include a `<restrict/>` element within the `page` elements that you wish to secure. If no explicit restriction is specified by the `restrict` element, an implied permission of `/viewId.xhtml:render` will be checked when the page is accessed via a non-faces (GET) request, and a permission of `/viewId.xhtml:restore` will be required when any JSF postback (form submission) originates from the page. Otherwise, the specified restriction will be evaluated as a standard security expression. Here's a couple of examples:

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

This page has an implied permission of `/settings.xhtml:render` required for non-faces requests and an implied permission of `/settings.xhtml:restore` for faces requests.

```
<page view-id="/reports.xhtml">
   <restrict>#{s:hasRole('admin')}</restrict>
</page>
```

Both faces and non-faces requests to this page require that the user is a member of the `admin` role.

15.6.5. Securing Entities

Seam security also makes it possible to apply security restrictions to read, insert, update and delete actions for entities.

To secure all actions for an entity class, add a `@Restrict` annotation on the class itself:

```
@Entity
@Name("customer")
@Restrict
public class Customer {
  ...
}
```

If no expression is specified in the `@Restrict` annotation, the default security check that is performed is a permission check of `entity:action`, where the permission target is the entity instance, and the `action` is either `read`, `insert`, `update` or `delete`.

It is also possible to only restrict certain actions, by placing a `@Restrict` annotation on the relevent entity lifecycle method (annotated as follows):

- `@PostLoad` - Called after an entity instance is loaded from the database. Use this method to configure a `read` permission.
- `@PrePersist` - Called before a new instance of the entity is inserted. Use this method to configure an `insert` permission.
- `@PreUpdate` - Called before an entity is updated. Use this method to configure an `update` permission.
- `@PreRemove` - Called before an entity is deleted. Use this method to configure a `delete` permission.

Here's an example of how an entity would be configured to perform a security check for any `insert` operations. Please note that the method is not required to do anything, the only important thing in regard to security is how it is annotated:

```
```

```
  @PrePersist @Restrict
  public void prePersist() {}
```

Using `/META-INF/orm.xml`

You can also specify the call back method in `/META-INF/orm.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                     xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
 http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
                 version="1.0">

   <entity class="Customer">
     <pre-persist method-name="prePersist" />
   </entity>

</entity-mappings>
```

Of course, you still need to annotate the `prePersist()` method on `Customer` with `@Restrict`

And here's an example of an entity permission rule that checks if the authenticated user is allowed to insert a new `MemberBlog` record (from the seamspace example). The entity for which the security check is being made is automatically inserted into the working memory (in this case `MemberBlog`):

```
rule InsertMemberBlog
  no-loop
  activation-group "permissions"
when
  principal: Principal()
                    memberBlog:          MemberBlog(member          :          member          ->
 (member.getUsername().equals(principal.getName())))
  check: PermissionCheck(target == memberBlog, action == "insert", granted == false)
then
  check.grant();
end;
```

This rule will grant the permission `memberBlog:insert` if the currently authenticated user (indicated by the `Principal` fact) has the same name as the member for which the blog entry is being created. The "`principal: Principal()`" structure that can be seen in the example code is a variable binding - it binds the instance of the `Principal` object from the working memory (placed there during authentication) and assigns it to a variable called `principal`. Variable bindings allow the value to be referred to in other places, such as the following line which compares the member's username to the `Principal` name. For more details, please refer to the JBoss Rules documentation.

Finally, we need to install a listener class that integrates Seam security with your JPA provider.

15.6.5.1. Entity security with JPA

Security checks for EJB3 entity beans are performed with an `EntityListener`. You can install this listener by using the following `META-INF/orm.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm http://java.sun.com/
xml/ns/persistence/orm_1_0.xsd"
                 version="1.0">

   <persistence-unit-metadata>
      <persistence-unit-defaults>
         <entity-listeners>
            <entity-listener class="org.jboss.seam.security.EntitySecurityListener"/>
         </entity-listeners>
      </persistence-unit-defaults>
   </persistence-unit-metadata>

</entity-mappings>
```

15.6.5.2. Entity security with a Managed Hibernate Session

If you are using a Hibernate `SessionFactory` configured via Seam, and are using annotations, or `orm.xml`, then you don't need to do anything special to use entity security.

15.6.6. Typesafe Permission Annotations

Seam provides a number of annotations that may be used as an alternative to `@Restrict`, which have the added advantage of providing compile-time safety as they don't support arbitrary EL expressions in the same way that `@Restrict` does.

Out of the box, Seam comes with annotations for standard CRUD-based permissions, however it is a simple matter to add your own. The following annotations are provided in the `org.jboss.seam.annotations.security` package:

- @Insert
- @Read
- @Update
- @Delete

To use these annotations, simply place them on the method or parameter for which you wish to perform a security check. If placed on a method, then they should specify a target class for which the permission will be checked. Take the following example:

```
@Insert(Customer.class)
public void createCustomer() {
  ...
}
```

In this example, a permission check will be performed for the user to ensure that they have the rights to create new `Customer` objects. The target of the permission check will be `Customer.class` (the actual `java.lang.Class` instance itself), and the action is the lower case representation of the annotation name, which in this example is `insert`.

It is also possible to annotate the parameters of a component method in the same way. If this is done, then it is not required to specify a permission target (as the parameter value itself will be the target of the permission check):

```
public void updateCustomer(@Update Customer customer) {
  ...
}
```

To create your own security annotation, you simply need to annotate it with `@PermissionCheck`, for example:

```
@Target({METHOD, PARAMETER})
@Documented
@Retention(RUNTIME)
@Inherited
@PermissionCheck
```

```
public @interface Promote {
  Class value() default void.class;
}
```

If you wish to override the default permisison action name (which is the lower case version of the annotation name) with another value, you can specify it within the `@PermissionCheck` annotation:

```
@PermissionCheck("upgrade")
```

15.6.7. Typesafe Role Annotations

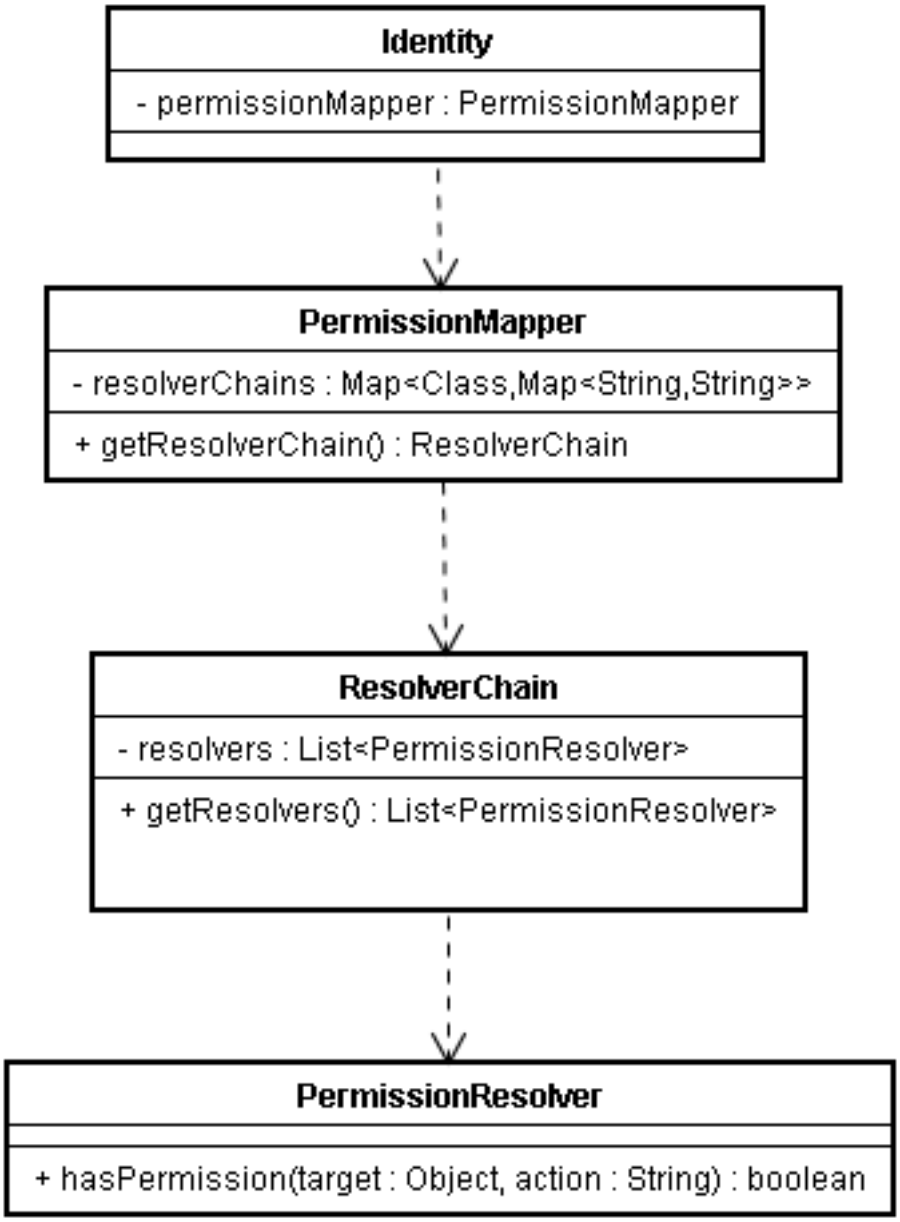
In addition to supporting typesafe permission annotation, Seam Security also provides typesafe role annotations that allow you to restrict access to component methods based on the role memberships of the currently authenticated user. Seam provides one such annotation out of the box, `org.jboss.seam.annotations.security.Admin`, used to restrict access to a method to users that are a member of the `admin` role (so long as your own application supports such a role). To create your own role annotations, simply meta-annotate them with `org.jboss.seam.annotations.security.RoleCheck`, like in the following example:

```
@Target({METHOD})
@Documented
@Retention(RUNTIME)
@Inherited
@RoleCheck
public @interface User {
}
```

Any methods subsequently annotated with the `@User` annotation as shown in the above example will be automatically intercepted and the user checked for the membership of the corresponding role name (which is the lower case version of the annotation name, in this case `user`).

15.6.8. The Permission Authorization Model

Seam Security provides an extensible framework for resolving application permissions. The following class diagram shows an overview of the main components of the permission framework:



The relevant classes are explained in more detail in the following sections.

15.6.8.1. PermissionResolver

This is actually an interface, which provides methods for resolving individual object permissions. Seam provides the following built-in `PermissionResolver` implementations, which are described in more detail later in the chapter:

- `RuleBasedPermissionResolver` - This permission resolver uses Drools to resolve rule-based permission checks.
- `PersistentPermissionResolver` - This permission resolver stores object permissions in a persistent store, such as a relational database.

15.6.8.1.1. Writing your own PermissionResolver

It is very simple to implement your own permission resolver. The `PermissionResolver` interface defines only two methods that must be implemented, as shown by the following table. By deploying your own `PermissionResolver` implementation in your Seam project, it will be automatically scanned during deployment and registered with the default `ResolverChain`.

Tableau 15.7. PermissionResolver interface

Return type	Method	Description
`boolean`	`hasPermission(Object target, String action)`	This method must resolve whether the currently authenticated user (obtained via a call to `Identity.getPrincipal()`) has the permission specified by the `target` and `action` parameters. It should return `true` if the user has the permission, or `false` if they don't.
`void`	`filterSetByAction(Set<Object> targets, String action)`	This method should remove any objects from the specified set, that would return `true` if passed to the `hasPermission()` method with the same `action` parameter value.

Note

As they are cached in the user's session, any custom `PermissionResolver` implementations must adhere to a couple of restrictions. Firstly, they may not contain any state that is finer-grained than session scope (and the scope of the component itself should either be application or session). Secondly, they must not use dependency injection as they may be accessed from multiple threads simultaneously. In fact, for performance reasons it is recommended that they are annotated with `@BypassInterceptors` to bypass Seam's interceptor stack altogether.

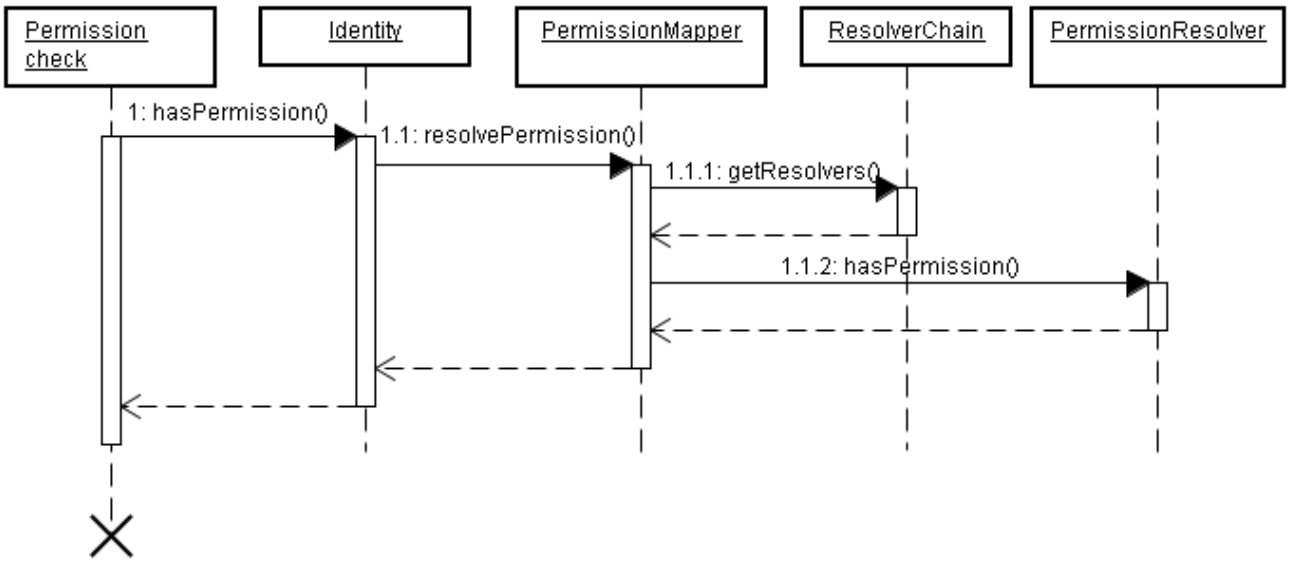
15.6.8.2. ResolverChain

A `ResolverChain` contains an ordered list of `PermissionResolvers`, for the purpose of resolving object permissions for a particular object class or permission target.

The default `ResolverChain` consists of all permission resolvers discovered during application deployment. The `org.jboss.seam.security.defaultResolverChainCreated` event is raised (and the `ResolverChain` instance passed as an event parameter) when the default

`ResolverChain` is created. This allows additional resolvers that for some reason were not discovered during deployment to be added, or for resolvers that are in the chain to be re-ordered or removed.

The following sequence diagram shows the interaction between the components of the permission framework during a permission check (explanation follows). A permission check can originate from a number of possible sources, for example - the security interceptor, the `s:hasPermission` EL function, or via an API call to `Identity.checkPermission`:



- 1. A permission check is initiated somewhere (either in code or via an EL expression) resulting in a call to `Identity.hasPermission()`.
- 1.1. `Identity` invokes `PermissionMapper.resolvePermission()`, passing in the permission to be resolved.
- 1.1.1. `PermissionMapper` maintains a `Map` of `ResolverChain` instances, keyed by class. It uses this map to locate the correct `ResolverChain` for the permission's target object. Once it has the correct `ResolverChain`, it retrieves the list of `PermissionResolvers` it contains via a call to `ResolverChain.getResolvers()`.
- 1.1.2. For each `PermissionResolver` in the `ResolverChain`, the `PermissionMapper` invokes its `hasPermission()` method, passing in the permission instance to be checked. If any of the `PermissionResolvers` return `true`, then the permission check has succeeded and the `PermissionMapper` also returns `true` to `Identity`. If none of the `PermissionResolvers` return true, then the permission check has failed.

15.6.9. RuleBasedPermissionResolver

One of the built-in permission resolvers provided by Seam, `RuleBasedPermissionResolver` allows permissions to be evaluated based on a set of Drools (JBoss Rules) security rules. A couple of the advantages of using a rule engine are 1) a centralized location for the business logic that

is used to evaluate user permissions, and 2) speed - Drools uses very efficient algorithms for evaluating large numbers of complex rules involving multiple conditions.

15.6.9.1. Requirements

If using the rule-based permission features provided by Seam Security, the following jar files are required by Drools to be distributed with your project:

- drools-api.jar
- drools-compiler.jar
- drools-core.jar
- drools-decisiontables.jar
- drools-templates.jar
- janino.jar
- antlr-runtime.jar
- mvel2.jar

15.6.9.2. Configuration

The configuration for `RuleBasedPermissionResolver` requires that a Drools rule base is first configured in `components.xml`. By default, it expects that the rule base is named `securityRules`, as per the following example:

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core"
            xmlns:security="http://jboss.com/products/seam/security"
            xmlns:drools="http://jboss.com/products/seam/drools"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation=
              "http://jboss.com/products/seam/core http://jboss.com/products/seam/core-2.2.xsd
                    http://jboss.com/products/seam/components  http://jboss.com/products/seam/
components-2.2.xsd
               http://jboss.com/products/seam/drools http://jboss.com/products/seam/drools-2.2.xsd
                   http://jboss.com/products/seam/security http://jboss.com/products/seam/security-
2.2.xsd">

    <drools:rule-base name="securityRules">
       <drools:rule-files>
          <value>/META-INF/security.drl</value>
       </drools:rule-files>
    </drools:rule-base>
```

```
</components>
```

The default rule base name can be overridden by specifying the `security-rules` property for `RuleBasedPermissionResolver`:

```
<security:rule-based-permission-resolver security-rules="#{prodSecurityRules}"/>
```

Once the `RuleBase` component is configured, it's time to write the security rules.

15.6.9.3. Writing Security Rules

The first step to writing security rules is to create a new rule file in the `/META-INF` directory of your application's jar file. Usually this file would be named something like `security.drl`, however you can name it whatever you like as long as it is configured correspondingly in `components.xml`.

So what should the security rules file contain? At this stage it might be a good idea to at least skim through the Drools documentation, however to get started here's an extremely simple example:

```
package MyApplicationPermissions;

import org.jboss.seam.security.permission.PermissionCheck;
import org.jboss.seam.security.Role;

rule CanUserDeleteCustomers
when
  c: PermissionCheck(target == "customer", action == "delete")
  Role(name == "admin")
then
  c.grant();
end
```

Let's break this down step by step. The first thing we see is the package declaration. A package in Drools is essentially a collection of rules. The package name can be anything you want - it doesn't relate to anything else outside the scope of the rule base.

The next thing we can notice is a couple of import statements for the `PermissionCheck` and `Role` classes. These imports inform the rules engine that we'll be referencing these classes within our rules.

Finally we have the code for the rule. Each rule within a package should be given a unique name (usually describing the purpose of the rule). In this case our rule is called

`CanUserDeleteCustomers` and will be used to check whether a user is allowed to delete a customer record.

Looking at the body of the rule definition we can notice two distinct sections. Rules have what is known as a left hand side (LHS) and a right hand side (RHS). The LHS consists of the conditional part of the rule, i.e. a list of conditions which must be satisfied for the rule to fire. The LHS is represented by the `when` section. The RHS is the consequence, or action section of the rule that will only be fired if all of the conditions in the LHS are met. The RHS is represented by the `then` section. The end of the rule is denoted by the `end` line.

If we look at the LHS of the rule, we see two conditions listed there. Let's examine the first condition:

```
c: PermissionCheck(target == "customer", action == "delete")
```

In plain english, this condition is stating that there must exist a `PermissionCheck` object with a `target` property equal to "customer", and an `action` property equal to "delete" within the working memory.

So what is the working memory? Also known as a "stateful session" in Drools terminology, the working memory is a session-scoped object that contains the contextual information that is required by the rules engine to make a decision about a permission check. Each time the `hasPermission()` method is called, a temporary `PermissionCheck` object, or *Fact*, is inserted into the working memory. This `PermissionCheck` corresponds exactly to the permission that is being checked, so for example if you call `hasPermission("account", "create")` then a `PermissionCheck` object with a `target` equal to "account" and `action` equal to "create" will be inserted into the working memory for the duration of the permission check.

Besides the `PermissionCheck` facts, there is also a `org.jboss.seam.security.Role` fact for each of the roles that the authenticated user is a member of. These `Role` facts are synchronized with the user's authenticated roles at the beginning of every permission check. As a consequence, any `Role` object that is inserted into the working memory during the course of a permission check will be removed before the next permission check occurs, if the authenticated user is not actually a member of that role. Besides the `PermissionCheck` and `Role` facts, the working memory also contains the `java.security.Principal` object that was created as a result of the authentication process.

It is also possible to insert additional long-lived facts into the working memory by calling `RuleBasedPermissionResolver.instance().getSecurityContext().insert()`, passing the object as a parameter. The exception to this is `Role` objects, which as already discussed are synchronized at the start of each permission check.

Getting back to our simple example, we can also notice that the first line of our LHS is prefixed with `c:`. This is a variable binding, and is used to refer back to the object that is matched by the condition (in this case, the `PermissionCheck`). Moving on to the second line of our LHS, we see this:

```
Role(name == "admin")
```

This condition simply states that there must be a `Role` object with a `name` of "admin" within the working memory. As already mentioned, user roles are inserted into the working memory at the beginning of each permission check. So, putting both conditions together, this rule is essentially saying "I will fire if you are checking for the `customer:delete` permission and the user is a member of the `admin` role".

So what is the consequence of the rule firing? Let's take a look at the RHS of the rule:

```
c.grant()
```

The RHS consists of Java code, and in this case is invoking the `grant()` method of the `c` object, which as already mentioned is a variable binding for the `PermissionCheck` object. Besides the `name` and `action` properties of the `PermissionCheck` object, there is also a `granted` property which is initially set to `false`. Calling `grant()` on a `PermissionCheck` sets the `granted` property to `true`, which means that the permission check was successful, allowing the user to carry out whatever action the permission check was intended for.

15.6.9.4. Non-String permission targets

So far we have only seen permission checks for String-literal permission targets. It is of course also possible to write security rules for permission targets of more complex types. For example, let's say that you wish to write a security rule to allow your users to create blog comments. The following rule demonstrates how this may be expressed, by requiring the target of the permission check to be an instance of `MemberBlog`, and also requiring that the currently authenticated user is a member of the `user` role:

```
rule CanCreateBlogComment
  no-loop
  activation-group "permissions"
when
  blog: MemberBlog()
  check: PermissionCheck(target == blog, action == "create", granted == false)
  Role(name == "user")
then
  check.grant();
end
```

15.6.9.5. Wildcard permission checks

It is possible to implement a wildcard permission check (which allows all actions for a given permission target), by omitting the `action` constraint for the `PermissionCheck` in your rule, like this:

```
rule CanDoAnythingToCustomersIfYouAreAnAdmin
when
  c: PermissionCheck(target == "customer")
  Role(name == "admin")
then
  c.grant();
end;
```

This rule allows users with the `admin` role to perform *any* action for any `customer` permission check.

15.6.10. PersistentPermissionResolver

Another built-in permission resolver provided by Seam, `PersistentPermissionResolver` allows permissions to be loaded from persistent storage, such as a relational database. This permission resolver provides ACL style instance-based security, allowing for specific object permissions to be assigned to individual users and roles. It also allows for persistent, arbitrarily-named permission targets (not necessarily object/class based) to be assigned in the same way.

15.6.10.1. Configuration

Before it can be used, `PersistentPermissionResolver` must be configured with a valid `PermissionStore` in `components.xml`. If not configured, it will attempt to use the default permission store, `JpaIdentityStore` (see section further down for details). To use a permission store other than the default, configure the `permission-store` property as follows:

```
<security:persistent-permission-resolver permission-store="#{myCustomPermissionStore}"/>
```

15.6.10.2. Permission Stores

A permission store is required for `PersistentPermissionResolver` to connect to the backend storage where permissions are persisted. Seam provides one `PermissionStore` implementation out of the box, `JpaPermissionStore`, which is used to store permissions inside a relational database. It is possible to write your own permission store by implementing the `PermissionStore` interface, which defines the following methods:

Tableau 15.8. PermissionStore interface

Return type	Method	Description
`List<Permission>`	`listPermissions(Object target)`	This method should return a `List` of `Permission` objects representing all the permissions granted for the specified target object.
`List<Permission>`	`listPermissions(Object target, String action)`	This method should return a `List` of `Permission` objects representing all the permissions with the specified action, granted for the specified target object.
`List<Permission>`	`listPermissions(Set<Object> targets, String action)`	This method should return a `List` of `Permission` objects representing all the permissions with the specified action, granted for the specified set of target objects.
`boolean`	`grantPermission(Permission)`	This method should persist the specified `Permission` object to the backend storage, returning true if successful.
`boolean`	`grantPermissions(List<Permission> permissions)`	This method should persist all of the `Permission` objects contained in the specified `List`, returning true if successful.
`boolean`	`revokePermission(Permission permission)`	This method should remove the specified `Permission` object

Return type	Method	Description
		from persistent storage.
`boolean`	`revokePermissions(List<Permission> permissions)`	This method should remove all of the `Permission` objects in the specified list from persistent storage.
`List<String>`	`listAvailableActions(Object target)`	This method should return a list of all the available actions (as Strings) for the class of the specified target object. It is used in conjunction with permission management to build the user interface for granting specific class permissions (see section further down).

15.6.10.3. JpaPermissionStore

This is the default `PermissionStore` implementation (and the only one provided by Seam), which uses a relational database to store permissions. Before it can be used it must be configured with either one or two entity classes for storing user and role permissions. These entity classes must be annotated with a special set of security annotations to configure which properties of the entity correspond to various aspects of the permissions being stored.

If you wish to use the same entity (i.e. a single database table) to store both user and role permissions, then only the `user-permission-class` property is required to be configured. If you wish to use separate tables for storing user and role permissions, then in addition to the `user-permission-class` property you must also configure the `role-permission-class` property.

For example, to configure a single entity class to store both user and role permissions:

```
<security:jpa-permission-store user-permission-class="com.acme.model.AccountPermission" />
```

To configure separate entity classes for storing user and role permissions:

```
<security:jpa-permission-store user-permission-class="com.acme.model.UserPermission"
   role-permission-class="com.acme.model.RolePermission" />
```

15.6.10.3.1. Permission annotations

As mentioned, the entity classes that contain the user and role permissions must be configured with a special set of annotations, contained within the `org.jboss.seam.annotations.security.permission` package. The following table lists each of these annotations along with a description of how they are used:

Tableau 15.9. Entity Permission annotations

Annotation	Target	Description
`@PermissionTarget`	`FIELD,METHOD`	This annotation identifies the property of the entity that will contain the permission target. The property should be of type `java.lang.String`.
`@PermissionAction`	`FIELD,METHOD`	This annotation identifies the property of the entity that will contain the permission action. The property should be of type `java.lang.String`.
`@PermissionUser`	`FIELD,METHOD`	This annotation identifies the property of the entity that will contain the recipient user for the permission. It should be of type `java.lang.String` and contain the user's username.
`@PermissionRole`	`FIELD,METHOD`	This annotation identifies the property of the entity that will contain the recipient role for the permission. It should be of type `java.lang.String` and contain the role name.
`@PermissionDiscriminator`	`FIELD,METHOD`	This annotation should be used when the same entity/ table is used to store both user and role permissions. It identifies the property of the entity that is used to discriminate between user and

Annotation	Target	Description
		role permissions. By default, if the column value contains the string literal `user`, then the record will be treated as a user permission. If it contains the string literal `role`, then it will be treated as a role permission. It is also possible to override these defaults by specifying the `userValue` and `roleValue` properties within the annotation. For example, to use `u` and `r` instead of `user` and `role`, the annotation would be written like this: `@PermissionDiscriminator (userValue="u", roleValue="r")`

15.6.10.3.2. Example Entity

Here is an example of an entity class that is used to store both user and role permissions. The following class can be found inside the SeamSpace example:

```
@Entity
public class AccountPermission implements Serializable {
  private Integer permissionId;
  private String recipient;
  private String target;
  private String action;
  private String discriminator;

  @Id @GeneratedValue
  public Integer getPermissionId() {
    return permissionId;
  }

  public void setPermissionId(Integer permissionId) {
    this.permissionId = permissionId;
  }
```

```
  @PermissionUser @PermissionRole
  public String getRecipient() {
    return recipient;
  }

  public void setRecipient(String recipient) {
    this.recipient = recipient;
  }

  @PermissionTarget
  public String getTarget() {
    return target;
  }

  public void setTarget(String target) {
    this.target = target;
  }

  @PermissionAction
  public String getAction() {
    return action;
  }

  public void setAction(String action) {
    this.action = action;
  }

  @PermissionDiscriminator
  public String getDiscriminator() {
    return discriminator;
  }

  public void setDiscriminator(String discriminator) {
    this.discriminator = discriminator;
  }
}
```

As can be seen in the above example, the `getDiscriminator()` method has been annotated with the `@PermissionDiscriminator` annotation, to allow `JpaPermissionStore` to determine which records represent user permissions and which represent role permissions. In addition, it can also be seen that the `getRecipient()` method is annotated with both `@PermissionUser` and `@PermissionRole` annotations. This is perfectly valid, and simply means that the `recipient`

property of the entity will either contain the name of the user or the name of the role, depending on the value of the `discriminator` property.

15.6.10.3.3. Class-specific Permission Configuration

A further set of class-specific annotations can be used to configure a specific set of allowable permissions for a target class. These permissions can be found in the `org.jboss.seam.annotation.security.permission` package:

Tableau 15.10. Class Permission Annotations

Annotation	Target	Description
`@Permissions`	`TYPE`	A container annotation, this annotation may contain an array of `@Permission` annotations.
`@Permission`	`TYPE`	This annotation defines a single allowable permission action for the target class. Its `action` property must be specified, and an optional `mask` property may also be specified if permission actions are to be persisted as bitmasked values (see next section).

Here's an example of the above annotations in action. The following class can also be found in the SeamSpace example:

```
@Permissions({
  @Permission(action = "view"),
  @Permission(action = "comment")
})
@Entity
public class MemberImage implements Serializable {
```

This example demonstrates how two allowable permission actions, `view` and `comment` can be declared for the entity class `MemberImage`.

15.6.10.3.4. Permission masks

By default, multiple permissions for the same target object and recipient will be persisted as a single database record, with the `action` property/column containing a comma-separated list of the granted actions. To reduce the amount of physical storage required to persist a large number of permissions, it is possible to use a bitmasked integer value (instead of a comma-separated list) to store the list of permission actions.

For example, if recipient "Bob" is granted both the `view` and `comment` permissions for a particular `MemberImage` (an entity bean) instance, then by default the `action` property of the permission entity will contain "`view,comment`", representing the two granted permission actions. Alternatively, if using bitmasked values for the permission actions, as defined like so:

```
@Permissions({
  @Permission(action = "view", mask = 1),
  @Permission(action = "comment", mask = 2)
})
@Entity
public class MemberImage implements Serializable {
```

The `action` property will instead simply contain "3" (with both the 1 bit and 2 bit switched on). Obviously for a large number of allowable actions for any particular target class, the storage required for the permission records is greatly reduced by using bitmasked actions.

Obviously, it is very important that the `mask` values specified are powers of 2.

15.6.10.3.5. Identifier Policy

When storing or looking up permissions, `JpaPermissionStore` must be able to uniquely identify specific object instances to effectively operate on its permissions. To achieve this, an *identifier strategy* may be assigned to each target class for the generation of unique identifier values. Each identifier strategy implementation knows how to generate unique identifiers for a particular type of class, and it is a simple matter to create new identifier strategies.

The `IdentifierStrategy` interface is very simple, declaring only two methods:

```
public interface IdentifierStrategy {
  boolean canIdentify(Class targetClass);
  String getIdentifier(Object target);
}
```

The first method, `canIdentify()` simply returns `true` if the identifier strategy is capable of generating a unique identifier for the specified target class. The second method, `getIdentifier()` returns the unique identifier value for the specified target object.

Seam provides two `IdentifierStrategy` implementations, `ClassIdentifierStrategy` and `EntityIdentifierStrategy` (see next sections for details).

To explicitly configure a specific identifier strategy to use for a particular class, it should be annotated with `org.jboss.seam.annotations.security.permission.Identifier`, and the value should be set to a concrete implementation of the `IdentifierStrategy` interface. An optional `name` property can also be specified, the effect of which is dependent upon the actual `IdentifierStrategy` implementation used.

15.6.10.3.6. ClassIdentifierStrategy

This identifier strategy is used to generate unique identifiers for classes, and will use the value of the `name` (if specified) in the `@Identifier` annotation. If there is no `name` property provided,

then it will attempt to use the component name of the class (if the class is a Seam component), or as a last resort it will create an identifier based on the name of the class (excluding the package name). For example, the identifier for the following class will be "`customer`":

```
@Identifier(name = "customer")
public class Customer {
```

The identifier for the following class will be "`customerAction`":

```
@Name("customerAction")
public class CustomerAction {
```

Finally, the identifier for the following class will be "`Customer`":

```
public class Customer {
```

15.6.10.3.7. EntityIdentifierStrategy

This identifier strategy is used to generate unique identifiers for entity beans. It does so by concatenating the entity name (or otherwise configured name) with a string representation of the primary key value of the entity. The rules for generating the name section of the identifier are similar to `ClassIdentifierStrategy`. The primary key value (i.e. the *id* of the entity) is obtained using the `PersistenceProvider` component, which is able to correctly determine the value regardless of which persistence implementation is used within the Seam application. For entities not annotated with `@Entity`, it is necessary to explicitly configure the identifier strategy on the entity class itself, for example:

```
@Identifier(value = EntityIdentifierStrategy.class)
public class Customer {
```

For an example of the type of identifier values generated, assume we have the following entity class:

```
@Entity
public class Customer {
  private Integer id;
  private String firstName;
  private String lastName;
```

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

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

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

For a `Customer` instance with an `id` value of `1`, the value of the identifier would be "`Customer:1`". If the entity class is annotated with an explicit identifier name, like so:

```
@Entity
@Identifier(name = "cust")
public class Customer {
```

Then a `Customer` with an `id` value of `123` would have an identifier value of "`cust:123`".

15.7. Permission Management

In much the same way that Seam Security provides an Identity Management API for the management of users and roles, it also provides a Permissions Management API for the management of persistent user permissions, via the `PermissionManager` component.

15.7.1. PermissionManager

The `PermissionManager` component is an application-scoped Seam component that provides a number of methods for managing permissions. Before it can be used, it must be configured with a permission store (although by default it will attempt to use `JpaPermissionStore` if it is available). To explicitly configure a custom permission store, specify the `permission-store` property in components.xml:

```
<security:permission-manager permission-store="#{ldapPermissionStore}"/>
```

The following table describes each of the available methods provided by `PermissionManager`:

Tableau 15.11. PermissionManager API methods

Return type	Method	Description
`List<Permission>`	`listPermissions(Object target, String action)`	Returns a list of `Permission` objects representing all of the permissions that have been granted for the specified target and action.
`List<Permission>`	`listPermissions(Object target)`	Returns a list of `Permission` objects representing all of the permissions that have been granted for the specified target and action.
`boolean`	`grantPermission(Permission permission)`	Persists (grants) the specified `Permission` to the backend permission store. Returns true if the operation was successful.
`boolean`	`grantPermissions(List<Permission> permissions)`	Persists (grants) the specified list

Return type	Method	Description
		of `Permissions` to the backend permission store. Returns true if the operation was successful.
`boolean`	`revokePermission(Permission permission)`	Removes (revokes) the specified `Permission` from the backend permission store. Returns true if the operation was successful.
`boolean`	`revokePermissions(List<Permission> permissions)`	Removes (revokes) the specified list of `Permissions` from the backend permission store. Returns true if the operation was successful.
`List<String>`	`listAvailableActions(Object target)`	Returns a list of the available actions for the specified target object. The actions that this method returns are

Return type	Method	Description
		dependent on the `@Permission` annotations configured on the target object's class.

15.7.2. Permission checks for PermissionManager operations

Invoking the methods of `PermissionManager` requires that the currently-authenticated user has the appropriate authorization to perform that management operation. The following table lists the required permissions that the current user must have.

Tableau 15.12. Permission Management Security Permissions

Method	Permission Target	Permission Action
`listPermissions()`	The specified `target`	`seam.read-permissions`
`grantPermission()`	The target of the specified `Permission`, or each of the targets for the specified list of `Permissions` (depending on which method is called).	`seam.grant-permission`
`grantPermission()`	The target of the specified `Permission`.	`seam.grant-permission`
`grantPermissions()`	Each of the targets of the specified list of `Permissions`.	`seam.grant-permission`
`revokePermission()`	The target of the specified `Permission`.	`seam.revoke-permission`
`revokePermissions()`	Each of the targets of the specified list of `Permissions`.	`seam.revoke-permission`

15.8. SSL Security

Seam includes basic support for serving sensitive pages via the HTTPS protocol. This is easily configured by specifying a `scheme` for the page in `pages.xml`. The following example shows how the view `/login.xhtml` is configured to use HTTPS:

```
<page view-id="/login.xhtml" scheme="https"/>
```

This configuration is automatically extended to both `s:link` and `s:button` JSF controls, which (when specifying the `view`) will also render the link using the correct protocol. Based on the previous example, the following link will use the HTTPS protocol because `/login.xhtml` is configured to use it:

```
<s:link view="/login.xhtml" value="Login"/>
```

Browsing directly to a view when using the *incorrect* protocol will cause a redirect to the same view using the *correct* protocol. For example, browsing to a page that has `scheme="https"` using HTTP will cause a redirect to the same page using HTTPS.

It is also possible to configure a *default scheme* for all pages. This is useful if you wish to use HTTPS for a only few pages. If no default scheme is specified then the normal behavior is to continue use the current scheme. So once the user accessed a page that required HTTPS, then HTTPS would continue to be used after the user navigated away to other non-HTTPS pages. (While this is good for security, it is not so great for performance!). To define HTTP as the default `scheme`, add this line to `pages.xml`:

```
<page view-id="*" scheme="http" />
```

Of course, if *none* of the pages in your application use HTTPS then it is not required to specify a default scheme.

You may configure Seam to automatically invalidate the current HTTP session each time the scheme changes. Just add this line to `components.xml`:

```
<web:session invalidate-on-scheme-change="true"/>
```

This option helps make your system less vulnerable to sniffing of the session id or leakage of sensitive data from pages using HTTPS to other pages using HTTP.

15.8.1. Overriding the default ports

If you wish to configure the HTTP and HTTPS ports manually, they may be configured in `pages.xml` by specifying the `http-port` and `https-port` attributes on the `pages` element:

```
<pages xmlns="http://jboss.com/products/seam/pages"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://jboss.com/products/seam/pages  http://jboss.com/products/
seam/pages-2.2.xsd"
```

```
    no-conversation-view-id="/home.xhtml"
    login-view-id="/login.xhtml"
    http-port="8080"
    https-port="8443">
```

15.9. CAPTCHA

Though strictly not part of the security API, Seam provides a built-in CAPTCHA (*C*ompletely *A*utomated *P*ublic *T*uring test to tell *C*omputers and *H*umans *A*part) algorithm to prevent automated processes from interacting with your application.

15.9.1. Configuring the CAPTCHA Servlet

To get up and running, it is necessary to configure the Seam Resource Servlet, which will provide the Captcha challenge images to your pages. This requires the following entry in `web.xml`:

```
<servlet>
   <servlet-name>Seam Resource Servlet</servlet-name>
   <servlet-class>org.jboss.seam.servlet.SeamResourceServlet</servlet-class>
</servlet>

<servlet-mapping>
   <servlet-name>Seam Resource Servlet</servlet-name>
   <url-pattern>/seam/resource/*</url-pattern>
</servlet-mapping>
```

15.9.2. Adding a CAPTCHA to a form

Adding a CAPTCHA challenge to a form is extremely easy. Here's an example:

```
<h:graphicImage value="/seam/resource/captcha"/>
<h:inputText id="verifyCaptcha" value="#{captcha.response}" required="true">
  <s:validate />
</h:inputText>
<h:message for="verifyCaptcha"/>
```

That's all there is to it. The `graphicImage` control displays the CAPTCHA challenge, and the `inputText` receives the user's response. The response is automatically validated against the CAPTCHA when the form is submitted.

15.9.3. Customising the CAPTCHA algorithm

You may customize the CAPTCHA algorithm by overriding the built-in component:

```
@Name("org.jboss.seam.captcha.captcha")
@Scope(SESSION)
public class HitchhikersCaptcha extends Captcha
{
  @Override @Create
  public void init()
  {
    setChallenge("What is the answer to life, the universe and everything?");
    setCorrectResponse("42");
  }

  @Override
  public BufferedImage renderChallenge()
  {
    BufferedImage img = super.renderChallenge();
    img.getGraphics().drawOval(5, 3, 60, 14); //add an obscuring decoration
    return img;
  }
}
```

15.10. Security Events

The following table describes a number of events (see *Chapitre 6, Evènements, intercepteurs et gestion des exceptions*) raised by Seam Security in response to certain security-related events.

Tableau 15.13. Security Events

Event Key	Description
`org.jboss.seam.security.loginSuccessful`	Raised when a login attempt is successful.
`org.jboss.seam.security.loginFailed`	Raised when a login attempt fails.
`org.jboss.seam.security.alreadyLoggedIn`	Raised when a user that is already authenticated attempts to log in again.
`org.jboss.seam.security.notLoggedIn`	Raised when a security check fails when the user is not logged in.
`org.jboss.seam.security.notAuthorized`	

Event Key	Description
	Raised when a security check fails when the user is logged in however doesn't have sufficient privileges.
`org.jboss.seam.security.preAuthenticate`	Raised just prior to user authentication.
`org.jboss.seam.security.postAuthenticate`	Raised just after user authentication.
`org.jboss.seam.security.loggedOut`	Raised after the user has logged out.
`org.jboss.seam.security.credentialsUpdated`	Raised when the user's credentials have been changed.
`org.jboss.seam.security.rememberMe`	Raised when the Identity's rememberMe property is changed.

15.11. Run As

Sometimes it may be necessary to perform certain operations with elevated privileges, such as creating a new user account as an unauthenticated user. Seam Security supports such a mechanism via the `RunAsOperation` class. This class allows either the `Principal` or `Subject`, or the user's roles to be overridden for a single set of operations.

The following code example demonstrates how `RunAsOperation` is used, by calling its `addRole()` method to provide a set of roles to masquerade as for the duration of the operation. The `execute()` method contains the code that will be executed with the elevated privileges.

```
new RunAsOperation() {
  public void execute() {
    executePrivilegedOperation();
  }
}.addRole("admin")
 .run();
```

In a similar way, the `getPrincipal()` or `getSubject()` methods can also be overriden to specify the `Principal` and `Subject` instances to use for the duration of the operation. Finally, the `run()` method is used to carry out the `RunAsOperation`.

15.12. Extending the Identity component

Sometimes it might be necessary to extend the Identity component if your application has special security requirements. The following example (contrived, as credentials would normally be handled by the `Credentials` component instead) shows an extended Identity component with an additional `companyCode` field. The install precendence of `APPLICATION` ensures that this extended Identity gets installed in preference to the built-in Identity.

```
@Name("org.jboss.seam.security.identity")
@Scope(SESSION)
@Install(precedence = APPLICATION)
@BypassInterceptors
@Startup
public class CustomIdentity extends Identity
{
  private static final LogProvider log = Logging.getLogProvider(CustomIdentity.class);

  private String companyCode;

  public String getCompanyCode()
  {
    return companyCode;
  }

  public void setCompanyCode(String companyCode)
  {
    this.companyCode = companyCode;
  }

  @Override
  public String login()
  {
    log.info("###### CUSTOM LOGIN CALLED ######");
    return super.login();
  }
}
```

Avertissement

Note that an `Identity` component must be marked `@Startup`, so that it is available immediately after the `SESSION` context begins. Failing to do this may render certain Seam functionality inoperable in your application.

15.13. OpenID

OpenID is a community standard for external web-based authentication. The basic idea is that any web application can supplement (or replace) its local handling of authentication by delegating responsibility to an external OpenID server of the user's chosing. This benefits the user, who no longer has to remember a name and password for every web application he uses, and the developer, who is relieved of some of the burden of maintaining a complex authentication system.

When using OpenID, the user selects an OpenID provider, and the provider assigns the user an OpenID. The id will take the form of a URL, for example `http://maximoburrito.myopenid.com` however, it's acceptable to leave off the `http://` part of the identifier when logging into a site. The web application (known as a relying party in OpenID-speak) determines which OpenID server to contact and redirects the user to the remote site for authentication. Upon successful authentication the user is given the (cryptographically secure) token proving his identity and is redirected back to the original web application.The local web application can then be sure the user accessing the application controls the OpenID he presented.

It's important to realize at this point that authentication does not imply authorization. The web application still needs to make a determination of how to use that information. The web application could treat the user as instantly logged in and give full access to the system or it could try and map the presented OpenID to a local user account, prompting the user to register if he hasn't already. The choice of how to handle the OpenID is left as a design decision for the local application.

15.13.1. Configuring OpenID

Seam uses the openid4java package and requires four additional JARs to make use of the Seam integration. These are: `htmlparser.jar`, `openid4java.jar`, `openxri-client.jar` and `openxri-syntax.jar`.

OpenID processing requires the use of the `OpenIdPhaseListener`, which should be added to your `faces-config.xml` file. The phase listener processes the callback from the OpenID provider, allowing re-entry into the local application.

```
<lifecycle>
   <phase-listener>org.jboss.seam.security.openid.OpenIdPhaseListener</phase-listener>
</lifecycle>
```

With this configuration, OpenID support is available to your application. The OpenID support component, `org.jboss.seam.security.openid.openid`, is installed automatically if the openid4java classes are on the classpath.

15.13.2. Presenting an OpenIdDLogin form

To initiate an OpenID login, you can present a simply form to the user asking for the user's OpenID. The `#{openid.id}` value accepts the user's OpenID and the `#{openid.login}` action initiates an authentication request.

```
<h:form>
   <h:inputText value="#{openid.id}" />
   <h:commandButton action="#{openid.login}" value="OpenID Login"/>
</h:form>
```

When the user submits the login form, he will be redirected to his OpenID provider. The user will eventually return to your application through the Seam pseudo-view `/openid.xhtml`, which is provided by the `OpenIdPhaseListener`. Your application can handle the OpenID response by means of a `pages.xml` navigation from that view, just as if the user had never left your application.

15.13.3. Logging in immediately

The simplest strategy is to simply login the user immediately. The following navigation rule shows how to handle this using the `#{openid.loginImmediately()}` action.

```
<page view-id="/openid.xhtml">
   <navigation evaluate="#{openid.loginImmediately()}">
      <rule if-outcome="true">
         <redirect view-id="/main.xhtml">
            <message>OpenID login successful...</message>
         </redirect>
      </rule>
      <rule if-outcome="false">
         <redirect view-id="/main.xhtml">
            <message>OpenID login rejected...</message>
         </redirect>
      </rule>
   </navigation>
</page>
```

Thie `loginImmediately()` action checks to see if the OpenID is valid. If it is valid, it adds an OpenIDPrincipal to the identity component, marks the user as logged in (i.e. `#{identity.loggedIn}` will be true) and returns true. If the OpenID was not validated, the method returns false, and the user re-enters the application un-authenticated. If the user's OpenID is valid, it will be accessible using the expression `#{openid.validatedId}` and `#{openid.valid}` will be true.

15.13.4. Deferring login

You may not want the user to be immediately logged in to your application. In that case, your navigation should check the `#{openid.valid}` property and redirect the user to a local registration or processing page. Actions you might take would be asking for more information and creating a local user account or presenting a captcha to avoid programmatic registrations. When you are done processing, if you want to log the user in, you can call the `loginImmediately` method, either through EL as shown previously or by directly interaction with the `org.jboss.seam.security.openid.OpenId` component. Of course, nothing prevents you from writing custom code to interact with the Seam identity component on your own for even more customized behaviour.

15.13.5. Logging out

Logging out (forgetting an OpenID association) is done by calling `#{openid.logout}`. If you are not using Seam security, you can call this method directly. If you are using Seam security, you should continue to use `#{identity.logout}` and install an event handler to capture the logout event, calling the OpenID logout method.

```
<event type="org.jboss.seam.security.loggedOut">
   <action execute="#{openid.logout}" />
</event>
```

It's important that you do not leave this out or the user will not be able to login again in the same session.

Internationnalisation, les langues locales et les thèmes

Seam rend facile la construction d'applications internationnalisées? En premier, regardons toutes les étapes nécéssaires pour internationnaliser et rendre avec une langue locale votre application. Ensuite nous allons regarde les composants livrés dans Seam.

16.1. Internationnalisation de votre application.

Une application JEE consisten en plusieurs composants et tous doivent être configurés de manière appropriés pour que votre application soit traduisible.

Note

Note that all i18n features in Seam work only in JSF context.

En partant du bas, la première étape est de s'assrer que le serveur de base de données et le client utilise le bon encodage de caractères pour votre langue. Normallement, vous devriez vouloir utiliser UTF-8. Comment faire cela est hors de porté de ce tutorial.

16.1.1. La configuration du serveur d'application

Pour s'assurer que le serveur d'application reçoit les paramètres dans l'encodage correct depuis les requêtes client, vous devez configurer le connecteur de tomcat. Si vous utilisez Tomcat ou JBoss AS, ajoutez l'attribut `URIEncoding="UTF-8"` à la configuration du connecteur. Pour JBoss AS 4.2, modifiez `${JBOSS_HOME}/server/(default)/deploy/jboss-web.deployer/server.xml`:

```
<Connector port="8080" URIEncoding="UTF-8"/>
```

Il ya une alternative qui est probablement meilleure. Vous pouvez dire à JBoss AS que l'encodage des paramètres des requeêtes sera prit depuis la requête:

```
<Connector port="8080" useBodyEncodingForURI="true"/>
```

16.1.2. Les chaines de caractères de l'application traduites

Vous allez avoir besoin des chaines de caractères traduites pour tous les *messages* de votre applicatin (par exemple les labels des champsd e vos vues). En premier, vous alleza avoir besoin

de vous assurer que vos fiches de ressources sont encodés avec l'encodage désiré. Par défaut, l'ASCII est utilisé. Mais l'ASCII n'est pas suffisant pour beaucoups de langues, il ne fourni pas toutes les lettres de tous les langages.

Les fichiers de ressource doivent être créés en ASCII ou utiliser le code déspécialisé Unicode pour représenter les lettres Unicode. Sinon vous ne pourez pas compiler le fichier de propriété dans le byte-code, il n'y a pas de moyen pour indiquer à la JVM quel groupe de caractères utiliser. Donc vous devez utiliser soit les caractères ASCII soi des caractères déspécialisés qui ne sont pas dans le groupe des caractères de l'ASCII. Vous pouvez représenter un caractère Unicode dans le fichier en Java en utilisant \uXXX, où XXX est la représentation hexadécimale de ce caractère.

Vous pouvez écrire votre traductions des labels (*Section 16.3, « Les labels »*) dans vos fichiers de ressource dans l'encodage natif et ensuite la conversation du fichier dans le format déspécialisé avec l'outil `native2ascii` fourni dans le JDK. Cet outil va convertir un fichier écrit dans votre encodage natif dans un nouveau où la représentation des caractères non-ASCII sera en séquence déspécialisé Unicode.

L'utilisation de cet outil est décrit dans *ici pour Java 5* [http://java.sun.com/j2se/1.5.0/docs/tooldocs/index.html#intl] ou *ici pour Java 6* [http://java.sun.com/javase/6/docs/technotes/tools/#intl]. Par exemple, pour convertir un fichier depuis UTF-8:

```
$ native2ascii -encoding UTF-8 messages_cs.properties >
 messages_cs_escaped.properties
```

16.1.3. D'autres réglages pour l'encodage

Soyez sur que les vues affichant vos informations localisées et les message utilisent le bon groupe de caractères et aussi que toute donnée soumise utilise le bon encodage.

Pour définir l'encodage de caractère à afficher, vous devez utiliser la balise `<f:view locale="cs_CZ"/>` (ici pour dire à JSF d'utiliser la langue Tchèque). Vous pouvez vouloir modifier l'encodage du document xml lui-même si vous voulez embarqué des chaines de caractères localisées en xml. Pour faire cela, modifiez l'attriut d'encodage dans la déclaration xml `<?xml version="1.0" encoding="UTF-8"?>` selon le besoin.

De plus JSF/Facelets devrait soumettre toutes les requêtes en utilisant l'encodage de caractères spécifié, mais pour êtrer sur que toutes les requêtes ne vont pas indiquer un encodage vous pouvez obliger l'encodage de la requête en utilisant un filtre servlet. Vous configurez cela dans `components.xml`:

```
<web:character-encoding-filter encoding="UTF-8"
  override-client="true"
  url-pattern="*.seam" />
```

16.2. Les locales

Chaque session de connection de l'utilisateur est associé avec une instance de `java.util.Locale` (disponible dans l'application comme un composant appelé `locale`). Dans les circonstances normales, vous n'avez pas besoin de définir la locale. Seam va simplement déléguer à JSF le fait de déterminer la locale active:

- S'il ya une langague associée avec la requête HTTP (la langue du navigateur), et que la langue est dans la liste des langues supportées dans le `faces-config.xml`, utilisez cette langue pour le reste de la session.
- Cependant, si la langue a été spécifiée dans le `faces-config.xml`, alors utilise cette langue pour le reste de la session.
- Sinon, utilise la locale par défaut du serveur.

C'est *possible* de définir la langue manuellement via les propriétés de configuration de Seam `org.jboss.seam.international.localeSelector.language`, `org.jboss.seam.international.localeSelector.country` et `org.jboss.seam.international.localeSelector.variant`, mais nous n'avons pas trouver de bonne raison d'avoir à faire cela.

C'est, cependant, utile de permettre à l'utilisateur de définir sa langue manuellement via un interface utilisateur de l'application. Seam fourni une fonctionnalité livrée pour remplacer la langue déterminée par l'algorithme ci-dessus. Tout ce que vous avez à faire est d'ajouter le fragment suivant dans un formulaire de votre page JSP ou Facelets:

```
<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']}"/>
```

Ou si vous voulez une liste de toutes les langues supportées depuis `faces-config.xml`, utilisez simplement :

```
<h:selectOneMenu value="#{localeSelector.localeString}">
   <f:selectItems value="#{localeSelector.supportedLocales}"/>
</h:selectOneMenu>
<h:commandButton action="#{localeSelector.select}"
   value="#{messages['ChangeLanguage']}"/>
```

Quand l'utilisateur sélectionne un élément depuis cette liste, ensuite il clique sur le bouton de commande, alors les lagues de Seam et de JSF vont être remplacées pour le reste de la session.

Pour répondre à la question à quel endroit les langues sont-elle définies. Typiquement, vous fournissez une liste de langues pour lesquelles vous avez un fichiers de ressources correspondant dans l'élément `<locale-config>` de fichier de configuration JSF (/META-INF/faces-config.xml). Cependant, vous avez appris à apprécier le mechanisme de configuration des composants de Seam qui est plus psuissant que celui fourni dans Java EE. Pour cette raison, vous pouvez configurer les langues supportées, et une langue par défaut sur le serveur, en utilisant le composant livré dénommé `org.jboss.seam.international.localeConfig`. Pour l'utiliser, vous déclarer en premier un espace de nommage XML pour le paquet international de Seam dans le descripteur du composant de Seam. Vous définissez ensuite une langue par défaut et les langues supportées comme ci-dessous:

```
<international:locale-config default-locale="fr_CA" supported-locales="en fr_CA fr_FR"/>
```

Naturellement, si vous déclarrez que vous supporté une langue, vous devriez fournir un fichier de ressource qui lui correspond! Ensuite, vous allez apprendre comment définir des labels spécifiques aux langues.

16.3. Les labels

JSF permet une internationnalisation des labels des interfaces utilisateurs et des texte descriptif via l'utilisation de `<f:loadBundle />`. Vous pouvez utilisez cette approche dans les application Seam. Au alternative, vous pouvez profiter du composant `messages` de Seam pour afficher les labels modèles avec les expressions EL embarquées.

16.3.1. La définition des labels

Seam fourni un`java.util.ResourceBundle` (disponible pour l'application comme un `org.jboss.seam.core.resourceBundle`). Vous allez avoir besoin de rendre les labels à internationnaliser disponible via ce fichier de ressource spécial. Par défaut, le fichier de ressource utilisé par Seam est appelé `messages` et donc vous allez avoir besoin de définir vos labels dans des fichiers appelés `messages.properties`, `messages_en.properties`, `messages_en_AU.properties`, etc. Ces fichiers vont habituellement dans le dossier `WEB-INF/classes`.

Ainsi, dans `messages_en.properties`:

```
Hello=Hello
```

Et dans `messages_en_AU.properties`:

```
Hello=G'day
```

Vous pouvez sélectionner un nom différent pour le fichier de ressource en définissant dans la propriété de configuration de Seam dénommée `org.jboss.seam.core.resourceLoader.bundleNames`. Vous pouvez même spécifie une liste de fichier de ressources à chercher (premier trouvé, premier servi) pour les messages.

```
<core:resource-loader>
   <core:bundle-names>
      <value>mycompany_messages</value>
      <value>standard_messages</value>
   </core:bundle-names>
</core:resource-loader>
```

Si vous voulez définir un message juste pour une page particulière, vous pouvez le spécifier dans le fichier de ressource avec le m^me nom que l'identifiant de la vue JSF, avec un `/` devant et en enlevant l'extension du nom de fichier. Ainsi vous pouvez mettre notre message dans `welcome/hello_en.properties` si vous avez seulement besoin d'afficher le message sur `/welcome/hello.jsp`.

Vous pouvez même spécifier un nom de fichier explicite dans `pages.xml`:

```
<page view-id="/welcome/hello.jsp" bundle="HelloMessages"/>
```

Ensuite, nous pourions utiliser les messages définie dans `HelloMessages.properties` avec `/welcome/hello.jsp`.

16.3.2. L'affichage des labels

Si vous définissez vos labels en utilisant le fichier de ressource de Seam, vous allez être capable de les utiliser sans avoir à indiquer `<f:loadBundle ... />` sur chaque page. Au lieu de cela, vous pouvez simplement indiquer:

```
<h:outputText value="#{messages['Hello']}"/>
```

ou:

```
<h:outputText value="#{messages.Hello}"/>
```

Même mieux, les mesages eux-même peuvent contenir des expressions EL:

```
Hello=Hello, #{user.firstName} #{user.lastName}
```

```
Hello=G'day, #{user.firstName}
```

Vous pouvez même utiliser les messages dans votre code:

```
@In private Map<String, String> messages;
```

```
@In("#{messages['Hello']}") private String helloMessage;
```

16.3.3. Les messages Faces

Le composant `facesMessages` est une façon super-simplifiée d'afficher un message de réussite ou d'echec à l'utilisateur. La fonctionnalité que nous avons juste à l'instant décrit fonctionne aussi avec les messages faces:

```
@Name("hello")
@Stateless
public class HelloBean implements Hello {
   @In FacesMessages facesMessages;

   public String sayIt() {
      facesMessages.addFromResourceBundle("Hello");
   }
}
```

Cela va afficher `Hello, Gavin King` ou `G'day, Gavin`, selon la locale de l'utilisateur.

16.4. Les fuseaux horaires

Il y aussi une instance d'étendue de session dénommé `java.util.Timezone`, dénommée `org.jboss.seam.international.timezone`, et un composant de Seam pour la modificaiton du fuseau horaire dénommé `org.jboss.seam.international.timezoneSelector`. Par défaut, le fuseau horaire est le fuseau horaire par défaut du serveur. Malheureusement, la spécification JSF indique que toute les dates et les heures devraient être en UTC, et affichées en UTC, à moins que

le fuseau horaire ne soit spécifié en utilisant `<f:convertDateTime>`. Ceci est un inconvénient majeur pour une fonctionnalité par défaut.

Seam remplace cette fonctionnalité et par défaut, toutes les dates et les heures sont du fuseau horaire de Seam. De plus, Seam fourni une balise `<s:convertDateTime>` qui réalise toujours une conversation dans le fuseau horaire de Seam.

Seam fourni aussi un convertisseur de date par défaut convertissant une valeur de chaine de caractères vers une date. Ceci vous préserve d'avoir à indiquer un convertisseur pour les champs de saisies qui sont simplement capturé comme des dates. Le patron est sélectionné en accord avec la langue de l'utilisateur et son fuseau horaire est sélectionné comme décrit ci-dessous.

16.5. Les thèmes

Les applications de Seam sont aussi très facilement personnalisable. L'API des thèmes est vraiment similaire à l'API de localisation, mais bien sur ces deux concernes de sujets orthogonaux et quelques application ont à la fois la localisation et les thèmes.

En permier, configurez le groupe de thème disponibles:

```
<theme:theme-selector cookie-enabled="true">
   <theme:available-themes>
      <value>default</value>
      <value>accessible</value>
      <value>printable</value>
   </theme:available-themes>
</theme:theme-selector>
```

Notez que le premier thème listé est le thème par défaut.

Les thèmes sont définies dans un fichier de propriétés avec le même nom que le thème. par exemple, le thème `default` est définie comme un groupe d'entréees dans `default.properties`. Par exemple, `default.properties` devrait définir:

```
css ../screen.css
template /template.xhtml
```

Habituellement, un fichier de ressource de thème aura des chemins vers des fichiers de styles CSS ou des images ou des noms de pattron de facelets (à la différence des fichiers de resource de la localisation qui sont habituellement des fichiers textes).

Maintenant, nous pouvons utiliser ces entrées dans nos pages JSP ou facelets. par exemple, pour personnaliser la feuille de style dans une page facelets:

```
<link href="#{theme.css}" rel="stylesheet" type="text/css" />
```

Ou, quand la définition de la page réside dans un sous-dossier:

```
<link href="#{facesContext.externalContext.requestContextPath}#{theme.css}"
   rel="stylesheet" type="text/css" />
```

Plus impressionnant, les facelets nous permette de personnaisée le pattron utilisé avec un `<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}">
```

Toute comme le sélectioneur de langue, il y a un sélectionneur de thème livré qui permet à l'utilisateur de librement basculer entre les thèmes:

```
<h:selectOneMenu value="#{themeSelector.theme}">
   <f:selectItems value="#{themeSelector.themes}"/>
</h:selectOneMenu>
<h:commandButton action="#{themeSelector.select}" value="Select Theme"/>
```

16.6. La préservation des préférences de langue et de thème via des cookies

Le sélectionneur de langue, le sélectionneur de thème et le sélectionneur de fuseau horaire supportent tous la persistance de la rpéférence vers un cookie. En simplement définissant la propriété `cookie-enabled` dans `components.xml`:

```
<theme:theme-selector cookie-enabled="true">
   <theme:available-themes>
      <value>default</value>
      <value>accessible</value>
      <value>printable</value>
   </theme:available-themes>
```

```
</theme:theme-selector>

<international:locale-selector cookie-enabled="true"/>
```

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.

17.1. Basic fomatting

Here is a simple example:

```
It's easy to make *emphasis*, |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 <i>emphasis</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.
```

(Note that a simple newline is ignored, you need an additional blank line to wrap text into a new 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>

<q>Nyeah nyeah-nee
<i>nyeah</i> nyeah!</q>

<p>
But what do you think he means by <q>nyeah-nee</q>?
</p>
```

17.2. 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>
```

Note that inline monospace formatting always escapes (most monospace formatted text is in fact code or tags with many special characters). So you can, for example, write:

```
This is a |<tag attribute="value"/>| example.
```

without escaping any of the characters inside the monospace bars. The downside is that you can't format inline monospace text in any other way (italics, underscore, and so on).

17.3. Links

A link may be created using the following syntax:

```
Go to the Seam website at [=>http://jboss.com/products/seam].
```

Or, if you want to specify the text of the link:

```
Go to [the Seam website=>http://jboss.com/products/seam].
```

For advanced users, it is even possible to customize the Seam Text parser to understand wikiword links written using this syntax.

17.4. 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!

17.5. Using the SeamTextParser

The `<s:formattedText/>` JSF component internally uses the `org.jboss.seam.text.SeamTextParser`. You can use that class directly and implement your own text parsing, rendering, or HTML sanitation procedure. This is especially useful if you have a custom frontend for entering rich text, such as a Javascript-based HTML editor, and you want to validate user input to protect your website against Cross-Site Scripting (XSS) attacks. Another usecase are custom wiki text parsing and rendering engines.

The following example defines a custom text parser that overrides the default HTML sanitizer:

```
public class MyTextParser extends SeamTextParser {

   public MyTextParser(String myText) {
      super(new SeamTextLexer(new StringReader(myText)));

      setSanitizer(
         new DefaultSanitizer() {
```

```
            @Override
            public void validateHtmlElement(Token element) throws SemanticException {
                // TODO: I want to validate HTML elements myself!
            }
        }
    );
}

// Customizes rendering of Seam text links such as [Some Text=>http://example.com]
@Override
protected String linkTag(String descriptionText, String linkText) {
    return "<a href=\"" + linkText + "\">My Custom Link: " + descriptionText + "</a>";
}

// Renders a <p> or equivalent tag
@Override
protected String paragraphOpenTag() {
    return "<p class=\"myCustomStyle\">";
}

public void parse() throws ANTLRException {
    startRule();
}

}
```

The `linkTag()` and `paragraphOpenTag()` methods are just some of many you can override to customize rendered output. These methods generally return `String`. See the Javadoc for more details.

Also consult the Javadoc of `org.jboss.seam.text.SeamTextParser.DefaultSanitizer` for more information on what HTML elements, attributes, and attribute values or filtered by default.

Génération iText PDF

Seam inclus maintenant un groupe de composant pour la génération de documents en utilisant iText. le premier but du support de document iText de Seam est pour la génération de documents PDF, mais Seam offre aussi un support basique pour la génération de document RTF.

18.1. Utilisation du support PDF

Le support de iText est fournie par `jboss-seam-pdf.jar`. Ce JAR contient les contrôles iText JSF, qui sont utilisés pour construire les vues qui peuvent être rendues vers le PDF et le composant DocumentStore qui servent pour rendre les documents à l'utilisateur. Pour inclure le support PDF dans votre application, il faut inclure`jboss-seam-pdf.jar` dans votre dossier `WEB-INF/lib` avec le fichier iText JAR. Il n'y a pas plus comme configuration nécéssaire pour le support de iText de Seam.

Le module de Seam iText requière l'utilisation de Facelets comme technologie d'affichage. Les versions futures de la bibliothèque pourrons aussi supporter l'utilisation de JSP. En plus, il faut utiliser le paquet seam-ui.

Le projet `examples/itext` contient un exemple du support de PDF en action. Il démontre comment est fait le déploiement propre des paquets et il contient de nombreux exemples qui montre les fonctionnalités clef de la génération de PDF actuellement supportées.

18.1.1. La création d'un document

`<p:document>`	*Description* Les documents sont générés par les documents facelets en utilisant les tags dans l'espace de nom `http://jboss.com/products/seam/pdf`. Les documents devraient toujours avoir le tag `document` comme racine du document. Le tag `document` prépare Seam pour générer un document dans le DocumentStore et à le rendre en HTML en redirigeant le contenu stocké. *Attribues* • `type` — Le type de document qui doit être produit. Les valeurs valides sont les modes `PDF`, `RTF` et `HTML` . Seam prends par défaut une génération PDF et beaucoup de fonctionnalités ne fonctionnent correctement qu'en génération des documents PDF. • `pageSize` — La taille de page à générer. Les valeurs utilisées de manière classiques devraient être `LETTER` et `A4`. Une pleine liste de tailles de pages supportées peut être trouvée dans la classe `com.lowagie.text.PageSize`. Autrement, pageSize peut fournir

une largeur et une hauteur de page directement. La valeur "612 792", par exemple, est équivalent à une taille de page de type LETTER.

- `orientation` — L'orientation de la page. Les valeurs valides sont `portrait` et `landscape`. En mode paysage, la les valeurs de largeur et la hauteur de la page sont inversées.
- `margins` — Les valeurs de marges haute, basse , droite et gauche.
- `marginMirroring` — Indique si les valeurs des marges doivent être inversées sur la page suivante.
- `disposition` — Avec la génération de PDF dans un navigateur web ceci détermine le `Content-Disposition` HTTP du docment. Les valeurs valides sont `inline`, qui indique que le document devrait ête affiché dans la fenêtre du navigateur si c'est possible, et `attachment`, qui indique le document devrait être traité comme téléchargeable. La valeur par défaut est `inline`.
- `fileName` — Pour les pièces jointes, cette valeur surcharge le nom du fichier téléchargé.

Metadata Attributes

- `title`
- `subject`
- `keywords`
- `author`
- `creator`

Usage

```
<p:document xmlns:p="http://jboss.com/products/seam/pdf"
>
  The document goes here.

</p:document
>
```

18.1.2. Les éléments d textes basiques

Les documents courrants ont besoin de contenir bien plus que seulement du texte; cependant, les composants UI standards sont prévue pour la génération HTML et pas très utile pour la génération

de contenu PDF. Ainis, Seam fourni des composants UI spéciaux pour la génération de parfait contenu PDF. Les balises comme `<p:image>` et `<p:paragraph>` sont les fondations de bases de simples documents. Les balises comme `<p:font>` fournissent des informations sur le style de tous les contenues autour d'elles.

`<p:paragraph>`	*Description*
	
	La plus part des textes doivent pouvoir être divisés en paragraphes donc les fragments de texte peuvent être enchainnées, formatés et disposant d'un style en groupes logiques.
	
	Attribues
	
	• `firstLineIndent`
	• `extraParagraphSpace`
	• `leading`
	• `multipliedLeading`
	• `spacingBefore` — Un espace doit être inséré avant l'élément.
	• `spacingAfter` — Un espace doit être inséré après l'élément.
	• `indentationLeft`
	• `indentationRight`
	• `keepTogether`
	
	Usage
	
	`<p:paragraph alignment="justify">`
	`  This is a simple document.  It isn't very fancy.`
	`</p:paragraph`
	`>`

`<p:text>`	*Description* La balise `text` permet aux fragments de textes d'être produit depuis des donnnées de l'application en utilisant les méchanisme de conversation JSF classique. De manière similaire à la balise `outputText` utilisé pour rendre les documents en HTML. *Attribues*

- `value` — La valeur a affiché. Ceci sera typiquement une expression liée à une valeur.

Usage

```
<p:paragraph>
   The item costs <p:text value="#{product.price}">
      <f:convertNumber type="currency" currencySymbol="$"/>
   </p:text>
</p:paragraph
>
```

`<p:html>`

Description

La balise `html` rends le contenu Html en PDF.

Attribues

- `value` — Le texte à afficher.

Usage

```
<p:html value="This is HTML with <b
>some markup</b
>." />
<p:html>
   <h1
>This is more complex HTML</h1>
   <ul>
      <li
>one</li>
      <li
>two</li>
      <li
>three</li>
   </ul>
</p:html>

<p:html>
      <s:formattedText value="*This* is |Seam Text| as HTML.  It's very^cool^." />
```

```
</p:html
>
```

`<p:font>`

Description

La balise de police de la police par défaut à utiliser pour tous les textes qu'elle englobe.

Attribues

- `name` — Le nom de la police, par exemple: `COURIER`, `HELVETICA`, `TIMES-ROMAN`, `SYMBOL` ou `ZAPFDINGBATS`.
- `size` — La taille en point de la police.
- `style` — Le style de la police. Toute combinaison de : `NORMAL`, `BOLD`, `ITALIC`, `OBLIQUE`, `UNDERLINE`, `LINE-THROUGH`
- `color` — La couleur de la police. (voir *Section 18.1.7.1, « Les valeurs des couleurs »* pour les valeurs des couleurs)
- `encoding` — Le groupe d'encodage de caractères.

Usage

```
<p:font name="courier" style="bold" size="24">
   <p:paragraph
>My Title</p:paragraph>
</p:font
>
```

`<p:textcolumn>`

Description

`p:textcolumn` inserts a text column that can be used to control the flow of text. The most common case is to support right to left direction fonts.

Attribues

- `left` — The left bounds of the text column
- `right` — The right bounds of the text column
- `direction` — The run direction of the text in the column: `RTL`, `LTR`, `NO-BIDI`, `DEFAULT`

Usage

```
<p:textcolumn left="400" right="600" direction="rtl"
>
   <p:font name="/Library/Fonts/Arial Unicode.ttf"
         encoding="Identity-H"
         embedded="true"
>#{phrases.arabic}</p:font
>
</p:textcolumn
>
```

`<p:newPage>`	*Description* `p:newPage` insère un saut de page. *Usage* `<p:newPage />`
`<p:image>`	*Description* `p:image` insère une image dans le document. Les images peuvent être chargées depuis le classpath ou depuis le contexte de l'applicationw web en utilisant l'attribut `value`. Les resources peuvent aussi être générée dynamiquement par le code de l'application. L'attribut `imageData` peut spécifier une expression en liaison avec une valeur cette valeur est un objet `java.awt.Image`. *Attribues* • `value` — Un nom de ressource ou une relation avec une expression d'une méthode pour l'imagé générée de l'application. • `rotation` — La rotation de l'image en degrée. • `height` — La hauteur de l'image . • `width` — La largeur de l'image. • `alignment`— L'alignement de l'image (voir *Section 18.1.7.2, « Les valeurs des aignments »* pour les valeurs possibles) • `alt` — Texte alternatif pour la représentation de l'image.

	• `indentationLeft`
	• `indentationRight`
	• `spacingBefore` — Un espace doit être inséré avant l'élément.
	• `spacingAfter` — Un espace doit être inséré après l'élément.
	• `widthPercentage`
	• `initialRotation`
	• `dpi`
	• `scalePercent` — La facteur de mise à l'échelle (en pourcentage) à utiliser pour l'image. Ceci peut être exprimée comme un seul pourcentage ou par deux valeurs représentant des pourcentages différents sur l'échelle de x et de y.
	• `scaleToFit` — Indique la taille en X et la taille Y sera calculé pour mettre à l'échelle l'image. L'image sera mise à l'échelle pour faire correspondre à ces dimnesions de manière aussi proche que possible pour préserver le ratio XY de l'image.
	• `wrap`
	• `underlying`
	Usage
	`<p:image value="/jboss.jpg" />`
	`<p:image value="#{images.chart}" />`

`<p:anchor>`	*Description* `p:anchor` défini des liens cliquables pour un document. Il supporte les attributs suivants: *Attribues* • `name` — Le nom du point d'ancrage dans le document de destination. • `reference` — La destination du lien référé par. Les liens vers des autres points dans le document devraient commencer par un "#". Par exemple, "#link1" pour indiquer une position d'ancrage avec un `name`

	à `link1`. Les liens peuvent aussi avoir un URL complète pour indiquer une ressource à l'extérieure du document. *Usage* `<p:listItem ><p:anchor reference="#reason1" >Reason 1</p:anchor ></p:listItem > ... <p:paragraph> <p:anchor name="reason1" >It's the quickest way to get "rich"</p:anchor > ... </p:paragraph >`

18.1.3. Entêtes et pieds de page

`<p:header>`
 	
`<p:footer>`
	Description
	
	Les composants `p:header` et `p:footer` fournissent la capacité de placer des textes en entêtes et en pieds de pages sur chaque page du document généré. Les déclarations d'entêtes et de pieds de pages devrait apparaitrent au début du document.
	
	Attribues
	
	• `alignment` — L'alignement de la section de la boite d'entête ou de pied de page. (voir *Section 18.1.7.2, « Les valeurs des aignments »* pour les valeurs d'alignements)
	
	• `backgroundColor` — La couleur d'arrière plan de la boite d'entête ou de pied de page. (voir *Section 18.1.7.1, « Les valeurs des couleurs »* pour les valeurs des couleurs)
	
	• `borderColor` — La couleur de la bordure de la boit d'entête oude pied de page. Les traits de bordures peuvent être définis individuellement en utilisant `borderColorLeft`, `borderColorRight`, `borderColorTop` et `borderColorBottom`.(voir *Section 18.1.7.1, « Les valeurs des couleurs »* pour les valeurs des couleurs)

	• `borderWidth` — La largeur de la bordure. Les traits de bordures peuvent être définie individuellement en utilisant `borderWidthLeft`, `borderWidthRight`, `borderWidthTop` et `borderWidthBottom`.
	
	Usage
	
	`<f:facet name="header">`
	`  <p:font size="12">`
	`  <p:footer borderWidthTop="1" borderColorTop="blue"`
	`  borderWidthBottom="0" alignment="center">`
	`  Why Seam? [<p:pageNumber />]`
	`  </p:footer>`
	`  </p:font>`
	`</f:facet`
	`>`

`<p:pageNumber>`	*Description*
	
	Le numéro de page courrant peut être placé dans l'entête ou le pied de page en utilisant la balise `p:pageNumber`. La balise de numéro de page peut seulement être utilisé dans le contexte d'une entête ou d'un pied de page et ne peut être utilisé qu'une fois.
	
	Usage
	
	`<p:footer borderWidthTop="1" borderColorTop="blue"`
	`  borderWidthBottom="0" alignment="center">`
	`  Why Seam? [<p:pageNumber />]`
	`</p:footer>`

18.1.4. Les chapitres et les sections

`<p:chapter>`
 	
`<p:section>`
	Description
	
	Si le document généré suit la structure d'un livre ou d'un article, les balises `p:chapter` et `p:section` peuvent être utilisés pour fournir la structure nécéssaire. Les sections peuvent seulement être utilisé dans des chapitres mais peuvent être englobé de manière arbitraire quelquesoit la profondeur de leur encapsulation. La plus part des visualisateurs de PDF fournissent une navigation facilitée entre les chapitres et les sections dans un document.

Note

You cannot include a chapter into another chapter, this can be done only with section(s).

Attribues

- `alignment` — L'alignement de la section de la boite d'entête ou de pied de page. (voir *Section 18.1.7.2, « Les valeurs des aignments »* pour les valeurs d'alignements)
- `number` — The chapter/section number. Every chapter/section should be assigned a number.
- `numberDepth` — The depth of numbering for chapter/section. All sections are numbered relative to their surrounding chapter/sections. The fourth section 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.

Note

Chapter(s) can have a number or without it by setting numberDepth to 0.

Usage

```
<p:document xmlns:p="http://jboss.com/products/seam/pdf"
        title="Hello">

  <p:chapter number="1">
    <p:title
><p:paragraph
>Hello</p:paragraph
></p:title>
    <p:paragraph
>Hello #{user.name}!</p:paragraph>
  </p:chapter>

  <p:chapter number="2">
```

```
      <p:title
><p:paragraph
>Goodbye</p:paragraph
></p:title>
      <p:paragraph
>Goodbye #{user.name}.</p:paragraph>
   </p:chapter>

</p:document
>
```

`<p:header>`	*Description* Tout chapitre ou section devrait contenur un `p:title`. Le titre sera affiché à côté du numéro de chapitre ou de section. Le corps du titre peut contenir un texte but ou peut être un `p:paragraph`.

18.1.5. Les listes

Les structures de listes peuvent être affiché en utilisant les balises `p:list` et `p:listItem`. Les listes peuvent contenir des sous-listes englobées arbitrairement. Les éléments de la liste ne devraient pas être utilisés à l'extérieur de la liste. Le document suivant utilise la balise `ui:repeat` pour afficher une liste des valeurs extraites depuis un composant de Seam.

```
<p:document xmlns:p="http://jboss.com/products/seam/pdf"
            xmlns:ui="http://java.sun.com/jsf/facelets"
            title="Hello">
   <p:list style="numbered">
      <ui:repeat value="#{documents}" var="doc">
         <p:listItem
>#{doc.name}</p:listItem>
      </ui:repeat>
   </p:list>
</p:document
>
```

`<p:list>`	*Attribues* • `style` — Les styles de listes de type ordonnées/numérotées. Une parmis: `NUMBERED`, `LETTERED`, `GREEK`, `ROMAN`, `ZAPFDINGBATS`, `ZAPFDINGBATS_NUMBER`. Si aucun style n'est données, les éléments de la listes sont précédés d'un rond.

- `listSymbol` — Pour les listes avec symboles, indique le type de symbole.
- `indent` — Le niveau d'indentation de la liste.
- `lowerCase` — Pour les styles de listes utilisant les lettres, indique si les lettres doivent être en minuscules.
- `charNumber` — Pour le style ZAPFDINGBATS, indique si le code du caractères précédant l'élément de la liste.
- `numberType` — Pour le style ZAPFDINGBATS_NUMBER, indique le style de numérotation.

Usage

```
<p:list style="numbered">
  <ui:repeat value="#{documents}" var="doc">
    <p:listItem
>#{doc.name}</p:listItem>
  </ui:repeat>
</p:list
>
```

`<p:listItem>`

Description

`p:listItem` dispose des attributs suivant:

Attribues

- `alignment` — L'alignement de l'élément de la liste. (Voir *Section 18.1.7.2, « Les valeurs des aignments »* pour les valeurs possibles)
- `indentationLeft` — La quantité d'indentation à gauche.
- `indentationRight` — La quantité d'indentation à droite.
- `listSymbol` — Remplace le symbole de la liste par défaut pour cet élement de la liste.

Usage

```
...
```

18.1.6. Les tableaux

Les structures d'un tableau peuvent être créés en uutilisant les balises `p:table` et `p:cell`. A l'inverse de beaucoup de structures de tableaux, il n'y a pas de déclaration explicite de ligne. Si un tableau a 3 colonnes, alors chacune des trois cellules devront automatiquement former une ligne. Les lignes d'entêtes et de pieds de pages peuvent être déclarées et seront répétées dans le cas ou la structure du tableau s'étallerait sur plusieurs pages.

`<p:table>`	*Description* `p:table` dispose des attributs suivants. *Attribues* • `columns` — Le nombre de colonnes (cellules) qui font une ligne de tableau. • `widths` — La largeur relative de chaque colonne. Il faudrait une valeur pour chacune des colonnes. Par exemple: widths="2 1 1" devrait indiquer qu'il y a 3 colonnes, et que la première colonne devrait être deux fois plus grande que la seconde et la troisième. • `headerRows` — Le nombre initial de ligne qui seront considérés comme des lignes d'entêtes ou des bas de tableau et devrait être répétés si le tableau s'étalle sur plusieurs pages. • `footerRows` — Le nombre de ligne qui sont à considérées comme des lignes de bas de tableaux. Cette valeur est soustraite de la valeur`headerRows`. Si le document a 2 lignes qui constituent un entête et une ligne qui constitut le bas du tableau, `headerRows` devrait être défini à 3 et `footerRows` devrait être définie à 1 • `widthPercentage` — Le pourcentage de la largeur de page que le tableau occupe. • `horizontalAlignment` — L'alignement horizontal du tableau. (Voir *Section 18.1.7.2, « Les valeurs des aignments »* pour les valeurs possibles) • `skipFirstHeader` • `runDirection` • `lockedWidth` • `splitRows` • `spacingBefore` — Un espace doit être inséré avant l'élément.

	• `spacingAfter` — Un espace doit être inséré après l'élément. • `extendLastRow` • `headersInEvent` • `splitLate` • `keepTogether` *Usage* `<p:table columns="3" headerRows="1"> <p:cell >name</p:cell> <p:cell >owner</p:cell> <p:cell >size</p:cell> <ui:repeat value="#{documents}" var="doc"> <p:cell >#{doc.name}</p:cell> <p:cell >#{doc.user.name}</p:cell> <p:cell >#{doc.size}</p:cell> </ui:repeat> </p:table >`
`<p:cell>`	*Description* `p:cell` dispose des attributs suivants. *Attribues* • `colspan` — Les cellules peuvent s'étendre sur plus d'une colonne en déclarant un `colspan` suppérieur à 1. Les tableaux ne n'ont pas la possibilité de s'étendre sur plusieurs lignes. • `horizontalAlignment` — L'alignement horizontal de la cellule. (voir *Section 18.1.7.2, « Les valeurs des aignments »* pour les valeurs possibles)

- `verticalAlignment` — L'alignement vertical de la cellule. (voir *Section 18.1.7.2, « Les valeurs des aignments »* pour les valeurs possibles)
- `padding` — L'encadrement d'un côté donné peut être spécifiés en utilisant `paddingLeft`, `paddingRight`, `paddingTop` et `paddingBottom`.
- `useBorderPadding`
- `leading`
- `multipliedLeading`
- `indent`
- `verticalAlignment`
- `extraParagraphSpace`
- `fixedHeight`
- `noWrap`
- `minimumHeight`
- `followingIndent`
- `rightIndent`
- `spaceCharRatio`
- `runDirection`
- `arabicOptions`
- `useAscender`
- `grayFill`
- `rotation`

Usage

```
<p:cell
>...</p:cell
>
```

18.1.7. Les constantes du document

Cette section documente quelques unes des constantes partagées par de multiples balises.

18.1.7.1. Les valeurs des couleurs

Plusieurs façons de spécifier les coleurs sont proposées. Un nombre limité de couleurs sont disponibles par leur nom. Il y a : `white`, `gray`, `lightgray`, `darkgray`, `black`, `red`, `pink`, `yellow`, `green`, `magenta`, `cyan` et `blue`. Les couleurs peuvent être" spécifiées comme une valeur intière, tout comme défini par `java.awt.Color`. Enfin, une valeur de couleur peut être spécifiée par `rgb(r,g,b)` ou `rgb(r,g,b,a)` avec les valeurs de rouge, vert, blue et alpha indiqué par un entier entre 0 et 255 ou comme un pourcentage avec un nombre à virgule suivi du signe '%'.

18.1.7.2. Les valeurs des aignments

Quand les valeurs des alignements sont utilisées, le PDF de Seam permet d'avoir les valeurs d'alignement horizontales suivants: `left`, `right`, `center`, `justify` et `justifyall`. Les valeurs d'alignement verticales sont `top`, `middle`, `bottom`, et `baseline`.

18.2. Diagrammes

Le support des diagrammes est aussi fourni avec `jboss-seam-pdf.jar`. Les diagrammes peuvent être utilisé dans des documents PDF pi peuvent être utilisés comme des images dans une page HTML. Faire des diagrammes nécéssite que la bibliothèque JFreeChart (`jfreechart.jar` et`jcommon.jar`) soit ajouté au dossier `WEB-INF/lib`. Quatres types de diagrammes sont actuellement diponibles: diagramme en camenbert, diagramme en baton et diagramme avec des courbes. Avec beaucoup de variantes ou de controles selon les besoin, il est possible de contruire des diagrammes en utilisant du code Java.

`<p:chart>`	*Description* L'affichage d'une diagramme créé en Java par un composant de Seam. *Attribues* • `chart` — L'objet diagramme à afficher. • `height` — La hauteur du diagramme. • `width` — La largeur du diagramme. *Usage* `<p:chart chart="#{mycomponent.chart}" width="500" height="500" />`
`<p:barchart>`	*Description*

L'affichage d'une diagramme en baton.

Attribues

- `chart` — L'objet diagramme à afficher, si la création du diagramme via la programmation est utilisée.
- `dataset` — Le groupe de données à afficher, si le groupe de données via la programmation est utilisé.
- `borderVisible` — Controle si oui ou non la bordure doit être affichée autour de tout le diagramme.
- `borderPaint` — La couleur de la bordure, si elle est visible;
- `borderBackgroundPaint` — La couleur d'arrière plan par défaut du diagramme.
- `borderStroke` —
- `domainAxisLabel` — Le label du texte pour l'axe du domaine.
- `domainLabelPosition` — L'angle pour les labels de categorie de l'axe du domaine. les valeurs valident sont `STANDARD`, `UP_45`, `UP_90`, `DOWN_45` et `DOWN_90`. Autrement, la valeur peut être une valeur dangle postive ou négative en radians.
- `domainAxisPaint` — La couleur du label de l'axe du domaine.
- `domainGridlinesVisible`— Les contrôles que les lignes d'arrière plan pour les axes du domaines soient visibles ou non dans le diagramme.
- `domainGridlinePaint`— La couleur des lignes d'arrière plan, si visible.
- `domainGridlineStroke` — Le style de ligne pour les lignes d'arrière plan du domaine, si visible.
- `height` — La hauteur du diagramme.
- `width` — La largeur du diagramme.
- `is3D` — Une valeur booléenne indiquant qe le diagramme devrait être rendu en 3D au lieu d'être en 2D.
- `legend` — Une valeur booléenne indiquant s'il faut ou non que le diagramme doit inclure une legende.

- `legendItemPaint`— La couleur par défaut pour les labels de texte dans la légende.
- `legendItemBackgoundPaint`— La couleur d'arrière plan pour la légende, si différente de la couleur d'arrière plan du diagramme.
- `legendOutlinePaint`— La couleur de la bordure autour de la légende.
- `orientation` — L'orientation du point , soit `vertical` (par défaut) ou `horizontal`.
- `plotBackgroundPaint`— La couleur de l'arrière plan du point.
- `plotBackgroundAlpha`— Le niveau alpha (transparence) de l'arrière plan du point. Il devrait être entre 0 (complètement transparant) et 1 (complètement opaque).
- `plotForegroundAlpha`— Le niveau alpha (transparence) du point. Il devrait être entre 0 (complètement transparant) et 1 (complètement opaque).
- `plotOutlinePaint`— La couleur du dégradé des lignes de grilles d'arrière plan, si visible.
- `plotOutlineStroke` — Le style de trait pour le dégradé des lignes de grilles d'arrière plan, si visible.
- `rangeAxisLabel` — Le label du texte de l'axe vertical .
- `rangeAxisPaint` — La couleur du label de l'axe vertical.
- `rangeGridlinesVisible`— Controle si les lignes de grilles darrière plan sont visibles ou non dans le diagramme.
- `rangeGridlinePaint`— La couleur des lignes de grille d'arrière plan verticales, si visibles.
- `rangeGridlineStroke` — La style de lignes pour les lignes d'arrières plan verticales, si visible.
- `title` — Le texte du titre du diagramme.
- `titlePaint`— La couleur du texte du titre du diagramme.
- `titleBackgroundPaint`— La couleur d'arrière plan autour du titre du diagramme.
- `width` — La largeur du diagramme.

	Usage `<p:barchart title="Bar Chart" legend="true" width="500" height="500"> <p:series key="Last Year"> <p:data columnKey="Joe" value="100" /> <p:data columnKey="Bob" value="120" /> </p:series > <p:series key="This Year"> <p:data columnKey="Joe" value="125" /> <p:data columnKey="Bob" value="115" /> </p:series> </p:barchart >`
`<p:linechart>`	*Description* Affichage d'une diagramme avec des courbes. *Attribues* • `chart` — L'objet diagramme à afficher, si la création du diagramme via la programmation est utilisée. • `dataset` — Le groupe de données à afficher, si le groupe de données via la programmation est utilisé. • `borderVisible` — Controle si oui ou non la bordure doit être affichée autour de tout le diagramme. • `borderPaint` — La couleur de la bordure, si elle est visible; • `borderBackgroundPaint` — La couleur d'arrière plan par défaut du diagramme. • `borderStroke` — • `domainAxisLabel` — Le label du texte pour l'axe du domaine. • `domainLabelPosition` — L'angle pour les labels de categorie de l'axe du domaine. les valeurs valident sont `STANDARD`, `UP_45`, `UP_90`, `DOWN_45` et `DOWN_90`. Autrement, la valeur peut être une valeur dangle postive ou négative en radians. • `domainAxisPaint` — La couleur du label de l'axe du domaine.

- `domainGridlinesVisible`— Les contrôles que les lignes d'arrière plan pour les axes du domaines soient visibles ou non dans le diagramme.
- `domainGridlinePaint`— La couleur des lignes d'arrière plan, si visible.
- `domainGridlineStroke` — Le style de ligne pour les lignes d'arrière plan du domaine, si visible.
- `height` — La hauteur du diagramme.
- `width` — La largeur du diagramme.
- `is3D` — Une valeur booléenne indiquant qe le diagramme devrait être rendu en 3D au lieu d'être en 2D.
- `legend` — Une valeur booléenne indiquant s'il faut ou non que le diagramme doit inclure une legende.
- `legendItemPaint` — La couleur par défaut pour les labels de texte dans la légende.
- `legendItemBackgoundPaint` — La couleur d'arrière plan pour la légende, si différent de la couleur d'arrière plan du diagramme.
- `legendOutlinePaint` — La couleur de la bordure autour de la légende.
- `orientation` — L'orientation du point , soit `vertical` (par défaut) ou `horizontal`.
- `plotBackgroundPaint` — La couleur de l'arrière plan du point.
- `plotBackgroundAlpha` — Le niveau alpha (transparence) pour l'arrière plan du point. Il devrait être un nombre entre 0 (complètement transparant) et 1 (complètement opaque).
- `plotForegroundAlpha` — Le niveau alpha (transparent) du point. Il devrait être un nombre entre 0 (complètement transparant) et 1 (complètement opaque).
- `plotOutlinePaint` — La couleur des lignes de grilles d'arrièreplans verticales, si visible.
- `plotOutlineStroke` — Le style de trait pour le dégradé des lignes de grilles d'arrière plan, si visible.
- `rangeAxisLabel` — Le label du texte de l'axe vertical .

- `rangeAxisPaint` — La couleur du label de l'axe vertical.
- `rangeGridlinesVisible` — Controle si les lignes d'arrière plans pour l'axe vertical doivent être visibles sur le diagramme.
- `rangeGridlinePaint` — La couleur des lignes d'arrière plan, si visible.
- `rangeGridlineStroke` — La style de lignes pour les lignes d'arrières plan verticales, si visible.
- `title` — Le texte du titre du diagramme.
- `titlePaint` — La couleur du texte du titre du diagramme.
- `titleBackgroundPaint` — La couleur d'arrière plan aaout du titre du diagramme.
- `width` — La largeur du diagramme.

Usage

```
<p:linechart title="Line Chart"
        width="500" height="500">
    <p:series key="Prices">
        <p:data columnKey="2003" value="7.36" />
        <p:data columnKey="2004" value="11.50" />
        <p:data columnKey="2005" value="34.625" />
        <p:data columnKey="2006" value="76.30" />
        <p:data columnKey="2007" value="85.05" />
    </p:series>
</p:linechart
>
```

`<p:piechart>`

Description

Affichage d'un diagramme en cammenbert.

Attribues

- `title` — Le texte du titre du diagramme.
- `chart` — L'objet diagramme à afficher, si la création du diagramme via la programmation est utilisée.

- `dataset` — Le groupe de données à afficher, si le groupe de données via la programmation est utilisé.
- `label` — Le texte du label par défaut pour les sections de camembert.
- `legend` — Une valeur booléenne indiquant si le diagramme devrait inclure une legende ou pas. Par défaut la valeur est à vrai.
- `is3D` —Une valeur booléenne indiquant que le diagramme devrait êtrere rendue en 3D au lieu d'en 2D.
- `labelLinkMargin` — La marge de lien pour les labels.
- `labelLinkPaint` — La couleur utilisé pour les lignes de liens des labels.
- `labelLinkStroke` — le type de bordure utilisé pour les lignes liant les labels.
- `labelLinksVisible` — Un drapeau qui controle si les liens vers les labels sont déssinés.
- `labelOutlinePaint` — La couleur utilisé pour déssiner les bordure des labels de sections.
- `labelOutlineStroke` — Le type de bordure utilisé pour dessiner les bordures des labels des sections.
- `labelShadowPaint` — La couleur utilisée pour dessiner l'ombrée sur les labels des sections.
- `labelPaint` — La couleur utilisé pour dessiner les labels des sections.
- `labelGap` — La distance entre les labels et les points comme un pourcentage de la largeur du point.
- `labelBackgroundPaint` — La couleur utilisée pour dessiner l'arrière plan des labels de sections. Si null, l'arrière plan n'est pas rempli.
- `startAngle` — L'angle de départ de la première section.
- `circular` — Une valeur boolléenne indicant que le diagramme devrait être dessiné comme un cercle. Si faux, le driagramme est déssiné comme une elipse. Par défaut à vrai.
- `direction` — La direction dont la section du camembert est dessinée. Soit: `clockwise` ou `anticlockwise`. Par défaut a `clockwise`.

	• `sectionOutlinePaint` — La couleur de bordure pour toutes les sections. • `sectionOutlineStroke` — Le type de bordure pour toutes les sections • `sectionOutlinesVisible` — Indique si une bordure est dessinée pour chaque section du point. • `baseSectionOutlinePaint` — La bordure de la section de base à dessiner. • `baseSectionPaint` — Le dessin de la section de base. • `baseSectionOutlineStroke` — Le style de trait pour la bordure extérieure . *Usage* `<p:piechart title="Pie Chart" circular="false" direction="anticlockwise" startAngle="30" labelGap="0.1" labelLinkPaint="red" > <p:series key="Prices" > <p:data key="2003" columnKey="2003" value="7.36" /> <p:data key="2004" columnKey="2004" value="11.50" /> <p:data key="2005" columnKey="2005" value="34.625" /> <p:data key="2006" columnKey="2006" value="76.30" /> <p:data key="2007" columnKey="2007" value="85.05" /> </p:series > </p:piechart >`
`<p:series>`	*Description* Les données de catégories peuvent être divisées en séries. La balise de la série est utilisée pour catégoriser un groupe de données et lui appliquer un style sur toute la série. *Attribues* • `key` — Le nom de la série. • `seriesPaint` — La couleur de chaque élément de la série

	• `seriesOutlinePaint` — La couleur pour chaque élément de la série. • `seriesOutlineStroke` — Le style de bordure à utiliser pour chaque élément de la série. • `seriesVisible` — Un booléen indiquant si la série doit ête affichée. • `seriesVisibleInLegend` — Un booléen indiquant si la serie doit être listé dans la légende. *Usage* `<p:series key="data1"> <ui:repeat value="#{data.pieData1}" var="item"> <p:data columnKey="#{item.name}" value="#{item.value}" /> </ui:repeat> </p:series >`
`<p:data>`	*Description* La balise data décrit chaque point à afficher pour le graphe. *Attribues* • `key` — Le nom d'élément donnée. • `series` — Le nom de la série, quand elle n'est pas embarquée dans un `<p:series>`. • `value` — La donnée en valeur numéique. • `explodedPercent` — Pour les diagramme en camembert, indique comme éclater les parts du camembert. • `sectionOutlinePaint` — Pour les diagrammes en baton, la couleur de la bordure de la section. • `sectionOutlineStroke` — Pour les diagrammes en baton, le type de bordure pour la section. • `sectionPaint` — Pour les diagramme en baton, la couleur de la section. *Usage*

```
<p:data key="foo" value="20" sectionPaint="#111111"
        explodedPercent=".2" />
<p:data key="bar" value="30" sectionPaint="#333333" />
<p:data key="baz" value="40" sectionPaint="#555555"
        sectionOutlineStroke="my-dot-style" />
```

`<p:color>`

Description

Le composant couleur déclare une couleur ou un dégradé qui peut être utilisé quand on dessine des formes géométriques pleines.

Attribues

- `color` — La valeur de la couleur. Pour un dégradé, ceci ets la couleur de départ. *Section 18.1.7.1, « Les valeurs des couleurs »*
- `color2` — Pour le dégradé, ceci ets la couleur de fin.
- `point` — La coordonnée où le dégrradé de couleur commence.
- `point2` — La coordonnées où le dégradé de couleur fini.

Usage

```
<p:color id="foo" color="#0ff00f"/>
<p:color id="bar" color="#ff00ff" color2="#00ff00"
                point="50 50" point2="300 300"/>
```

`<p:stroke>`

Description

Description d'un trait pour dessiner des lignes dans un diagramme.

Attribues

- `width` — La largeur du trait.
- `cap` — Le type de ligne. Les valeurs valident sont `butt`, `round` et `square`
- `join` — Le type de ligne de liaison. Les valeurs valident sont `miter`, `round` et `bevel`
- `miterLimit` — Pour une liaison en angle, cette valeur est la taille limite de la liaison.

	• `dash` — La valeur des tirets défini pour le patron de tirets à utiliser pour dessiner la ligne. Des entiers séparés par des espaces indique la longeur de chaque segments d'espacement, puis de tiret. • `dashPhase` — The dash phase indicates the offset into the dash pattern that the line should be drawn with. *Usage* `<p:stroke id="dot2" width="2" cap="round" join="bevel" dash="2 3" />`

18.3. Les codes Bar.

Seam peut utiliser iText pour générer des codes barres dans une grande variété de formats. Ces codes barres peuvent être embarqués dans un document PDF ou affichés comme une image sur une page Web. Notez que quand on utilise comme une image en HTML, les codes barres ne peuvent pas afficher de texte dans le code barre.

`<p:barCode>`	*Description* Affichage d'une image barcode. *Attribues* • `type` — Un type de code barre supporté par iText. Les valeurs valident sont : `EAN13`, `EAN8`, `UPCA`, `UPCE`, `SUPP2`, `SUPP5`, `POSTNET`, `PLANET`, `CODE128`, `CODE128_UCC`, `CODE128_RAW` et `CODABAR`. • `code` — La valeur a encodé comme un code barres. • `xpos` — Pour les PDFs, la position absolue en y du barre code dans la page . • `ypos` — Pour les PDFs, la position absolue en y du code barre dans la page. • `rotDegrees` — Pour les PDFs, le facteur de rotation du code barre en degrée. • `barHeight` — La hauteur du code barre dans le barCode • `minBarWidth` — La largeur minimale de la barre. • `barMultiplier` — Le facteur multiplicateur de la barre pour les barres larges ou la ditance entre les barres pour les codes `POSTNET` et`PLANET`.

	• `barColor` — La couleur pour déssiner les barres. • `textColor` — La couleur pour tout texte sur le code barres. • `textSize` — La taille du texte du code barres s'il y a du texte. • `altText` — Le texte `alt` pour les liens des images en HTML. *Usage* `<p:barCode type="code128" barHeight="80" textSize="20" code="(10)45566(17)040301" codeType="code128_ucc" altText="My BarCode" />`

18.4. Remplissage de formulaires

Si vous avez un PDF prégénéré et complexe avec des champs de formunaire, vous pouvez facillement remplir les valeurs depuis votre applicatio et le présenter aux utilisateurs.

`<p:form>`	*Description* Définissez un modèle de formulaire à remplir *Attribues* • `URL` — Une URL indiquant où le fichier PDF est à utiliser comme modèle. Si la valeur n'a pas de partie avec un protocole(://), le fichier est lu localement. • `filename` — Le nom de fichier à utiliser pour générer le fichier PDDF • `exportKey` — Placez le fichier PDF à générer dans un objet DocumentData sous la clé spécifiée dans le contexte evenementiel. Si défini, aucune redirection n'intervient.

`<p:field>`	*Description* Connectez le nom d'un champs à sa valeur *Attribues* • `name` — Le nom du champs.

	• `value` — La valeur du champs • `readOnly` — Le champ est-il en lecture seule? Par défaut à vrai.

```
<p:form
      xmlns:p="http://jboss.com/products/seam/pdf"
      URL="http://localhost/Concept/form.pdf">
   <p:field name="person.name" value="Me, myself and I"/>
</p:form>
```

18.5. Le rendu des composants Swing/AWT

Seam now provides experimental support for rendering Swing components into a PDF image. Some Swing look and feels supports, notably ones that use native widgets, will not render correctly.

`<p:swing>`	*Description* Rendre un composant Swing dans un document PDF. *Attribues* • `width` — La largeur du document à rendre. • `height` — La hauteur du document à rendre. • `component` — Une expression dont la valeur est un composant Swing ou AWT. *Usage* `<p:swing width="310" height="120" component="#{aButton}" />`

18.6. La configuration de iText

La génération de document fonctionne tel quelle sans aucune configurationns additionnelles nécéssaire. Cependant, il y a quelques éléments de configuration qui sont écéssaire pour des applications plus importantes.

L'implémentation par défaut délivre des documents PDF depuis une URL générique, `/seam-doc.seam`. Beaucoup de navigateurs (et d'utilisateurs) préfèrerons voir les URLs qui contienne un véritable nom de PDF comme `/myDocument.pdf`. Cette capacité nécéssite un peu de configuration. Pour délivrer des fichiers PDF , toutes les resources `*.pdf` devrait être liées au DocumentStoreServlet:

```
<servlet>
   <servlet-name
>Document Store Servlet</servlet-name>
   <servlet-class
>org.jboss.seam.document.DocumentStoreServlet</servlet-class>
</servlet>

<servlet-mapping>
   <servlet-name
>Document Store Servlet</servlet-name>
   <url-pattern
>*.pdf</url-pattern>
</servlet-mapping
>
```

L'option `use-extensions` sur le document stocke le composant complète la fonctionnalité en introduisant le document stocké en générant les URLs avec l'extension sur le nom de fichier correcte pour le document ayant été généré.

```
<components xmlns="http://jboss.com/products/seam/components"
   xmlns:document="http://jboss.com/products/seam/document"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
     http://jboss.com/products/seam/document http://jboss.com/products/seam/document-2.2.xsd
       http://jboss.com/products/seam/components http://jboss.com/products/seam/components-
2.2.xsd">
   <document:document-store use-extensions="true"/>
</components
>
```

Le porte-document stocke les documents dans l'étendue de conversation et les document vont expirés quand la conversation se terminera. A ce moment là, les références vers le document seront invalidées. Vous pouvez spécifier uen vue par défaut à afficher quand un document n'existe pas en utilisant la propriété `error-page` du `documentStore`.

```
<document:document-store use-extensions="true" error-page="/documentMissing.seam" />
```

18.7. Pour plus de documentation

Pour plus d'informations sur iText, voir:

- *Site Web de iText* [http://www.lowagie.com/iText/]
- *iText en Action* [http://www.manning.com/lowagie/]

The Microsoft® Excel® spreadsheet application

Seam support aussi la génération de feuilles de calculs the Microsoft® Excel® spreadsheet application au travers de l'excelente bibliothèqe *JExcelAPI* [http://jexcelapi.sourceforge.net/]. Le document généré est compatible avec the Microsoft® Excel® spreadsheet application versions 95, 97, 2000, XP et 2003. Actuellement un sous-groupe limité de fonctionnalité de la bibliothèque sont exploité mais le but ultime est d'être capable de proposer tout ce que la bibliothèque est capable de faire. Merci de vous référer à la documentation JExcelAPI pour plus d'informations sur les possibilités et les limitations.

19.1. Le support d'The Microsoft® Excel® spreadsheet application

The Microsoft® Excel® spreadsheet application `jboss-seam-excel.jar`. Ce JAR contient les controles JSF pour the Microsoft® Excel® spreadsheet application qui sont utilisé pour construire les vues qui peuvent rendre le document et le composant DocumentStore, qui donne le document rendu à l'utilisateur. Pour inclure le support d'the Microsoft® Excel® spreadsheet application dans votre application, includez `jboss-seam-excel.jar` dans votre dossier `WEB-INF/lib` avec le fichier JAR `jxl.jar`.Cependant, vous allez devoir configurer le servlet DocumentStore dans votre web.xml

Le module de Seam The Microsoft® Excel® spreadsheet application nécéssite l'utilisation des Facelets comme technologie des vues. De plus, il nécessite l'utilisation du package seam-ui.

Le projet `examples/excel` contient un exemple opérationnel du support d'the Microsoft® Excel® spreadsheet application. Il démontre un empaquetage de déploiement propre, et il montre les fonctionnalités disponibles.

La personnalisation du module pour le support d'autres type d'API pour les feuilles de calcul d'the Microsoft® Excel® spreadsheet application ont été fabriquée de manière simple. Implémentez l'interface `ExcelWorkbook` et enregistrez le dans components.xml.

```
<excel:excelFactory>
  <property name="implementations">
    <key
>myExcelExporter</key>
    <value
>my.excel.exporter.ExcelExport</value>
  </property>
</excel:excelFactory
>
```

et enregistrez l'espace de nom d'excel avec la balise du composant.

```
xmlns:excel="http://jboss.com/products/seam/excel"
```

Ensuite, définisez le type UIWorkbook à `myExcelExporter` et votre propre exportateur sera utilisé. par défaut c'est "jxl", mais le support de CSV a été aussi ajouté, en utilisant le type "csv".

Voir *Section 18.6, « La configuration de iText »* pour l'information de comment configurer le servlet de document pour distribuer le document avec une extension .xls.

Si vous avez des problèmes pour accéder au fichier généré sous IE (particulièrement avec https), soyez sur de ne pas avoir de restricutions trop strictes dans le navigateur (voir *http://www.nwnetworks.com/iezones.htm*), les contraintes de sécurité trop stricte dans web.xml ou une combinaisons des deux.

19.2. La création d'une simple classeur de travail

Lu'ilisation classique du support de feuille de calcul, utilise un classique `<h:dataTable>` et vous pouvez le lier avec un `List`, un `Set`, un `Map`, un `Array` ou un `DataModel`.

```
<e:workbook xmlns:e="http://jboss.com/products/seam/excel">
  <e:worksheet>
    <e:cell column="0" row="0" value="Hello world!"/>
  </e:worksheet>
</e:workbook>
```

Ce n'est pas très difficile, allons voir les clas les plus courants:

```
<e:workbook xmlns:e="http://jboss.com/products/seam/excel">
  <e:worksheet value="#{data}" var="item">
    <e:column>
      <e:cell value="#{item.value}"/>
    </e:column>
  </e:worksheet>
</e:workbook>
```

En premier, nous allons avoir un élément de haut-niveau workbook qui sert de containeur et qui n'a pas d'attributs. L'élément fil worksheet a deux attributs; value="#{data}" c'est la liaison EL avec la donnée et var="item" est le nom de l'élément courant. Englobé dans le worksheet, il y a une seule colonne et à l'intérieur vous pouvez voir une cellule qui à la liaison finale avec la données avec l'élément itirré courrant

Cela vous permet de commencer à remplir de données vos feuilles de calculs!

19.3. Les Workbooks

Les workboocks sont les parents de haut-niveaux des worksheets et des liens des feuilles de styles.

`<e:workbook>`	*Les attributes* • `type` — Défini quel module d'exportation à utiliser. La valeur est une chaine de caractères et peut aussi bien être "jxl" ou "csv". Par défaut c'est "jxl". • `templateURI` — Un modèle qui peut être utilisé comme une base de classeurs. La valeur est une chaine de caractères (URI). • `arrayGrowSize` — La quantité de mémoire pour augmenter la quantité de mémoire allouée pour stoquer les données du classeur. Pour les processus ouvrant de petits classeurs dans un WAS itl peut être nécéssaire de réduire la taille par défaut. La valeur par défaut est de 1 mégaoctet. La valeur est un nombre (bytes). • `autoFilterDisabled` — Le filtrage automatique est-il désactiver? La valeur est un booléen. • `cellValidationDisabled` — La validation de la cellule devrait-elle être ignorée? La valeur est un booléen. • `characterSet` — L'encodage de caractère. Seulement utilisé quand la feuille de clacul est lue, et n'a pas d'effets quand le classeur est écrit. La valeur est une chaine de caractères (character set encoding). • `drawingsDisabled` — Le dessin devrait-il être désactivé ? La valeur est un booléen. • `excelDisplayLanguage` — Le langage dans lequel le fichier généré sera affiché. Cette valeur est une chaine de caractères (deux caractères du code du pays en ISO 3166).

- `excelRegionalSettings` — Le réglage régionnal pour le fichier excel généré. La valeur est une chaine de caractères (deux caractères du code du pays en ISO 3166).
- `formulaAdjust` — Les formules devrait-elles être ajustée? La valeur est un booléen.
- `gcDisabled` — Le ramasse miètes devrait-il être désactivé? La valeur est un booléen.
- `ignoreBlanks` — Les espaces devrait-ils être ignorés? La valeur est un booléen
- `initialFileSize` — La quantité de mémoire initial allouée pour stoquer les données du classeur pendant la lecture du classeur. Pour des processus de lecture de plusieurs petits classuers dans un WAS il peut être nécéssaire de réduire la taille par défaut. La valeur par défaut est de 5 mega-octets. La valeur est un nombre (bytes).
- `locale` — La locale utilisé par JExcelApi pour générer le classeur. Le réglage de cette valeur n'a aucun effet sur la langue ou la régio du fichier excel généré. La valeur est une chaine de caractères.
- `mergedCellCheckingDisabled` — La fussion de la validation de cellule doit-elle être désactivé? La valeur est un booléen.
- `namesDisabled` — La liaison des noms doit-elle être désactivé. la valeur est un booléen.
- `propertySets` — La définition de propriétés devrait elle être défini (comme les macros) pour être copiée avec dans le classuer ? En laisant cette fonctionné active entrainera le processus JXL à utiliser beaucoup plus de mémoire. La valeur est un booléen.
- `rationalization` — Le formatage des cellules devrait-il être rationnalisé avant l'écriture de la feuille ? La valeur est un booléen. Par défaut à true.
- `supressWarnings` — Les alertes devrait-elles être supprimée? Du aux modifications dans la journalisation des logs dans la version 2.4, il va maintenant définir cette fonctionnalité au travers de la JVM (selon le type d'outil de log utilisé). La valeur est un booléen.
- `temporaryFileDuringWriteDirectory` — Utilisé en conjonction avec le réglage `useTemporaryFileDuringWrite` pour définir le dossier cible pour les fichiers temporaires. Cette valeur peut être NULL, dans ce cas le dossier temporaire par défaut normal pour

le système est utilisé en lieu et place. La valeur est une chaine (le dossier vers lequel les fichiers temporaires devraient être écrit).

- `useTemporaryFileDuringWrite` — Un fichier temporaire devrait être utilisé pendant la génération de ce classuer. Si non défini, le classeur sera placé dans la mémoire. La définition de ce drapeau entraine un choix dans la gestion entre l'utilisation de la mémoire et la performance. La valeur est un booléen.
- `workbookProtected` — Le classeur devrait-il être protégé? La valeur est un booléen.
- `filename` — Le nom de fichier utilisé pendant le téléchargement. La valeur est une chaine de caractère. Merci de noez que si vous lié le DocumentServlet à un patron, l'extension du fichier doit aussi correspondre.
- `exportKey` — Une clef qui stocke les données résultats dans un objet DocumentData dans l'étendue évènement. Si utilisé, il n'y a pas de redirection.

Les élements enfants

- `<e:link/>` — Zéro ou plusieurs liens vers des feuilles de styles (voir *Section 19.14.1, « Les liens vers les feuilles de styles »*).
- `<e:worksheet/>` — Zéro ou plusieurs feuilles de calculs (voir *Section 19.4, « Les feuilles de calculs »*).

Les Facets

- `aucun`

```
<e:workbook>
  <e:worksheet>
    <e:cell value="Hello World" row="0" column="0"/>
  </e:worksheet>
<e:workbook>
```

définie un classeur avec une feuille de calcul et format A1

19.4. Les feuilles de calculs

Les feuilles de calculs sont les enfant des classeurs et les parents des colonnes et des commandes des feuilles de calculs. Ils contiennents aussi les cellules placé explicitement, les formules, les images et les liens hypertextes. Ils sont les pages qui font le classeurs.

`<e:worksheet>`	• `value` — Une expression EL pour les données en réserve. La valeur est une chaine de caractères. La cible de cette expression est de trouver un Iterable. Noez que si la cible est une Map, l'itération se fait sur le Map.Entry entrySet(), et donc vous devez utilize un .key ou un .value pour cibler vos références. • `var` — Le nom de la variable de la ligne courrant itérée qui peut êtree plus tard référencée dans la cellule comme des attributs de valeur de cellule. La valeur est une chaine de caractères. • `name` — Le nom du classeur. La valeur est une cahine de caractères. Par défaut à Sheet# quand # est l'index de la feuille de calcul. Si le nom donné de classeur existe, la feuille est sélectionné. Ceci peut être utilisé pour fusionner des groupes de données dans une seule feuille de calcul, en définissant juste le même nom pour eux (en utilisant`startRow` et `startCol` pour être sur que cela ne va pas occuper le même emplacement). • `startRow` — Définie la ligne de départ pour les données. La valeur est un nombre. Utilisé pour positioner les données dans les différents endroits que le coin supérieur gauche (particulièrement utile s'il y a des groupes de données multiples pour une seule feuille de calcul). Par défaut à 0. • `startColumn` — Définie la colonne de démarrage pour les données. La valeur est un nombre. Utilisé pour placer les données dans un autre endroit que le coij suppérieur gauche (particulièrement utile si on a plusieurs groupes de données pour une seule feuille de calcul). Par défaut à 0. • `automaticFormulaCalculation` — Faut-il automatiquementy calculé les formules? La valeur est un booléen. • `bottomMargin` — La marge du bas. La valeur est un nombre (en pouces). • `copies` — Le nombre de copies. la valeur est un nombre. • `defaultColumnWidth` — La largeur de la colonne par défaut. La valeur est un nombre (nombre de caractères * 256).

- `defaultRowHeight` — La hauteur par défaut de la ligne. La valeur est un nombre (1/20ième d'un point).
- `displayZeroValues` — Faut-il afficher les valeur zéro? La valeur est un booléen.
- `fitHeight` — Le nombre de pages verticales qui imprimera la feuille. La valeur est un nombre.
- `fitToPages` — Faut-il en imprimer en faisant du sur-mesures pour les pages? La valeur est un booléen.
- `fitWidth` — La nombre de page de largeur que cette feuille va utilisé pour s'imprimer. La valeur est un nombre.
- `footerMargin` — La marge pour tout pied-de-page. La valeur est un nombre (en pouces).
- `headerMargin` — La marge pour tous les entêtes de pages. La valeur est un nombre (en pouces).
- `hidden` — La feuille de calcul doit-elle être cachée? La valeur est un booléen.
- `horizontalCentre` — La feuille de calcul doit-elle être centrée horizontallement? La valeur est un booléen.
- `horizontalFreeze` — La ligne à partir de laquelle le volet est bloquée verticallement. La valeur est un nombre.
- `horizontalPrintResolution` — La résolution d'impression horizontale. La valeur est un nombre.
- `leftMargin` — La marge de gauche. La valeur est un nombre (en pouces).
- `normalMagnification` — Le facteur de zoom normal (sans zoom ou sans facteur d'echelle). La valeur est un nombre (en pourcentage).
- `orientation` — Le sens d'orientation du papier pour l'impression de la feuille. La valeur est une chaine de caractères qui peut être aussi bien"landscape" que "portrait".
- `pageBreakPreviewMagnification` — Le facteur de zoom en prévisualisation du saut de page (aucun zoom ou de facteurs d'échelle). La valeur est un nombre (en pourcentage).
- `pageBreakPreviewMode` — Faut-il montrer les page en mode prévisualisation? La valeur est un booléen.

- `pageStart` — Le numéro de page à partir duquel commencer à imprimer. La valeur est un nombre.
- `paperSize` — La taille du papier à utiliser pour l'impression de cette feuille. La valeur est une chaine de caractères qui peut être soit "a4", "a3", "letter", "legal" etc (voir *jxl.format.PaperSize* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/PaperSize.html]).
- `password` — Le mot de passe pour cette feuille. La valeur est une chaine de caractères.
- `passwordHash` — Le type de hash du mot de passe - utilisation avec la copie de feuilles. La valeur est une chaine de caractères.
- `printGridLines` — Faut-il imprimer les lignes de la grille? La valeur est un booléen.
- `printHeaders` — Faut-il imprimer les entêtes? La valeur est un booléen.
- `sheetProtected` — Faut-il que la feuille soit protégé (en lecture seule)? La valeur est un booléen.
- `recalculateFormulasBeforeSave` — Faut-il que les formules soit recalculées quand la feuille est sauvegardée? La valeur est un booléen. Par fdéfaut la valeur est false.
- `rightMargin` — La marge de droit. La valeur est un nombre (en pouces).
- `scaleFactor` — Le facteur de zoom pour cette feuille à utiliser pour l'impression. la valeur est un nombre (en pourcentage).
- `selected` — Faut-il que la feuille soit sélectionné quand le classeur s'ouvre? La valeur est un booléen.
- `showGridLines` — Faut-il montrer la grille ? La valeur est un booléen.
- `topMargin` — La marge du haut. La valeur est un nombre (en pouces).
- `verticalCentre` — Centrer verticallement? La valeur est un booléen.
- `verticalFreeze` — La ligne à partir de laquelle le volet est bloqué verticallement. La valeur est un nombre.

- `verticalPrintResolution` — La résolution d'impression verticale. La valeur est un nombre.
- `zoomFactor` — Le facteur de zoom. Ne pas confondre avec le facteur de zoom (qui est relatif à la vue écran) et le facteur d'échelle (qui se réfère au facteur d'échelle pour l'impression). La valeur est un nombre (en pourcentage).

Les élements enfants

- `<e:printArea/>` — Zéro ou plusieurs définitions des zones d'impressions (voir *Section 19.11, « Les zones d'impressions et les titres »*).
- `<e:printTitle/>` — Zéro ou plusieurs définitions de titres pour l'impression (voir *Section 19.11, « Les zones d'impressions et les titres »*).
- `<e:headerFooter/>` — Zéroou plusieurs définitions d'entêtes/pieds de pages (voir *Section 19.10, « Les entêtes et les pieds de page »*).
- Zéro ou plusieurs commandes de feuilles de calculs (voir *Section 19.12, « Les commandes du classeur de calcul »*).

Les Facets

- `header`— Contenu qui sera placé dans le haut de bloc de données, au dessus des entêtes de colonnes (s'il y en a).
- `footer`— Contenu qui sera placé en bas du bloc de données, au dessous du bas des colonnes (s'il y en a).

```
<e:workbook>
  <e:worksheet name="foo" startColumn="1" startRow="1">
      <e:column value="#{personList}" var="person">
        <f:facet name="header">
          <e:cell value="Last name"/>
        </f:facet>
       <e:cell value="#{person.lastName}"/>
    </e:column>
  </e:worksheet>
<e:workbook>
```

défini un classeur de calcul avec le nom "foo", commençant en B2.

19.5. Les colonnes

Les colonnes sont les enfants des feuilles de calculs et les parents des cellules, des images, formules et hyperliens. Ils sont la structure qui controle l'itération dans les données de la feuille de calcul. Voir *Section 19.14.5, « Les réglages de la colonne »* pour le formattage.

`<e:column>`	*Les attributes* • `aucun` *Les élements enfants* • `<e:cell/>` — Zéro ou plusieurs cellules (voir *Section 19.6, « Les cellules »*). • `<e:formula/>` — Zéro ou plusieurs formules (voir *Section 19.7, « Les formules »*). • `<e:image/>` — Zéro ou plusieurs images (voir *Section 19.8, « Les images »*). • `<e:hyperLink/>` — Zéro ou plusieurs hyperliens (voir *Section 19.9, « Les hyperliens »*). *Les Facets* • `header` — Ce facet peut/pourra contenenir une `<e:cell>` , une `<e:formula>` , une `<e:image>` ou un `<e:hyperLink>` qui sera utilisé comme entête pour la colonne. • `footer` — Ce facet peux/pourra contenir une `<e:cell>` , une `<e:formula>` , une `<e:image>` ou un `<e:hyperLink>` qui sera utilisé pour le pied de la colonne.

```
<e:workbook>
    <e:worksheet value="#{personList}" var="person">
      <e:column>
        <f:facet name="header">
          <e:cell value="Last name"/>
```

```
                </f:facet>
                <e:cell value="#{person.lastName}"/>
            </e:column>
        </e:worksheet>
    <e:workbook>
```

définie une colonne avec un entétête et une sortie itérable

19.6. Les cellules

Les cellules sont regroupé dans des colonnes (pour l'itération) ou dans des classeurs (pour un positionnement directe en utilisant les attributs `column` et `row`) et sont responsable pour l'affichage de valeurs (habituellement au travers d'expressions EL en liaison avec l'attribut `var`-de la base de données. Voir *???*

`<e:cell>`	*Les attributes* • `column` — La colonne où la position de la cellule. Par défaut c'est un compteur interne. La valeur est un nombre. Notez que la valeur est basé sur un index zéro. • `row` — La ligne où est localisé la cellule. Par défaut c'est un compteur interne. La valeur est un nombre. Notez que la valeur est basé sur un index zéro. • `value` — La valeur à afficher. Habituellement une expression EL référençant un attribut var lié à la base de données. La valeur est une chaine de caractères. • `comment` — Un commentaire à ajouter à la cellule. La valeur est une chaine de caractères. • `commentHeight` — La hauteur du commentaire. La valeur est un nombre (en pixel). • `commentWidth` — La largeur du commentaire. la valeur est un nombre (en pixels). *Les élements enfants* • Zéro ou plusieurs conditions de validation (voir *Section 19.6.1, « La validation »*). *Les Facets*

	• aucun

```
        <e:workbook>
          <e:worksheet
>
            <e:column value="#{personList}" var="person">
              <f:facet name="header">
                  <e:cell value="Last name"/>
              </f:facet>
              <e:cell value="#{person.lastName}"/>
            </e:column>
          </e:worksheet>
        </e:workbook
>
```

définie une colonne avec un entétête et une sortie itérable

19.6.1. La validation

Les validations sont incluses dans les cellules ou les formules. Elles ajoutent des contraintes pour les données des cellules.

`<e:numericValidation>`	*Les attributes*
	• `value` — La limite (ou la limite basse quand c'est applicable) de la validation. La valeur est un nombre.
	• `value2` — La limite haute (quand c'est applicable) de la validation. La valeur est un nombre.
	• `condition` — La condition de validation. La valeur est une chaine de caractères.
	  • "equal" - requise la valeur de la cellule à faire correspondre à une définie dans l'attribut valeur
	  • "greater_equal" - requises la valeur de la cellule devant être plus grande ou égale à al valeur définie dans l'attribut valeur
	  • "less_equal" - requies que la valeur de la cellule soit inférieur ou égale à la valeur définie dans l'attribut valeur

	• "less_than" - requière que la valeur de la cellule soit inférieure à la valeur définie dans la valeur attribue • "not_equal" - requière que la valeur de la cellule ne corresponde pas à celle définie dans l'attribut valeur. • "between" - requière que la valeur de la cellule soit entre les valeur définies dans les attributs valeur- et valeur2 • "not_between" - requière que la valeur de la cellule ne soit pas entre les valeur définies dans les attributs valeur- et valeur2 *Les élements enfants* • `aucun` *Les Facets* • `aucun`

```
        <e:workbook>
          <e:worksheet>
            <e:column value="#{personList}" var="person"
>
              <e:cell value="#{person.age">
                <e:numericValidation condition="between" value="4"
                  value2="18"/>
              </e:cell>
            </e:column>
          </e:worksheet>
        </e:workbook
>
```

ajoute une validation numérique à une cellule en spécifiant que la valeur doit être entre 4 et 18.

`<e:rangeValidation>`	*Les attributes* • `startColumn` — La colonne de départ pour un écart de valeur serva,t à valider. La valeur est un nombre.

	• `startRow` — La ligne de départ d'un écart de valeur servant à valider. la valeur est un nombre. • `endColumn` — La colonne de fin d'un écart de valeur servant à valider. La valeur est un nombre. • `endRow` — La ligne de fin d'un écart de valeur servant à valider. La valeur est un nombre. *Les élements enfants* • `aucun` *Les Facets* • `aucun`

```
        <e:workbook>
          <e:worksheet>
            <e:column value="#{personList}" var="person"
>
                  <e:cell value="#{person.position">
                 <e:rangeValidation startColumn="0" startRow="0"
                   endColumn="0" endRow="10"/>
               </e:cell>
             </e:column>
          </e:worksheet>
        </e:workbook
>
```

ajoute la validation à une cellule spécifiant que la valeur doit être comprise dans les valeurs spécifiées dans la zone A1:A10.

`<e:listValidation>`	*Les attributes* • `aucun` *Les élements enfants* • `Zéro ou une liste d'éléments de validations.`

	Les Facets • `aucun`

e:listValidation est juste un container pour conserver les multiples balises e:listValidationItem.

`<e:listValidationItem>`	*Les attributes* • `value` — Une valeur servant à valider. *Les élements enfants* • `aucun` *Les Facets* • `aucun`

```
        <e:workbook>
          <e:worksheet>
            <e:column value="#{personList}" var="person"
>
              <e:cell value="#{person.position">
                <e:listValidation>
                  <e:listValidationItem value="manager"/>
                  <e:listValidationItem value="employee"/>
                </e:listValidation>
              </e:cell>
            </e:column>
          </e:worksheet>
        </e:workbook
>
```

ajoute une validation à la cellule spécifiant que la valeur doit être "manager" ou "employee".

19.6.2. Masque de formatage

Les masques de formatage sont spécifiés dans l'attribut masque des cellules ou des formules. Il ya deux types de masques de formatage, un pour les nombres, l'autree pour les dates

19.6.2.1. Les masques pour les nombres

Quand on rencontre un masque de format, en premier il est vérifié s'il est de la forme interrne, par exemple "format1", "accounting_float" et ensuite s'il (voir *jxl.write.NumberFormats* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/write/NumberFormats.html]).

Si le masque n'est pas dans la liste, il est traité comme un masque personnalisé (voir *java.text.DecimalFormat* [http://java.sun.com/javase/6/docs/api/java/text/DecimalFormat.html]). par exemple "0.00" et automatiquement converti dans le correspondant le plus proche.

19.6.2.2. Les masques de date

When encountering a format mask, first it is checked if it is in internal form, e.g "format1", "format2" and so on (see *jxl.write.DateFormats* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/write/DateFormats.html]).

si le masque n'est pas dans la liste, il estr traité comme un masque personnalisé (voir *java.text.DateFormat* [http://java.sun.com/javase/6/docs/api/java/text/DateFormat.html])., e.g "dd.MM.yyyy" et automatiquement converti dans le correspondant le plus proche.

19.7. Les formules

Les formules sont regroupé dans les colonnes (pour l'itération) ou dans les feuilles de calcul (pour le placement direct en utilisant les attributs `column` et `row`) et ajoute les opérations de calcul ou les fonctions à un groupe de cellules. Ils sont surtout des cellules, voir *Section 19.6, « Les cellules »* pour les attributs disponibles, Notez qu'ils peuvent appliquer des modèles et ont leur propre définition depolices comme des cellules classiques.

La formule de la cellule est placé dans l'attribut `value` comme une notation the Microsoft® Excel® spreadsheet application classique. Notez que quand on fait des formules croisant des feuilles, le classeur doit exister avant de lui référencer une formule. La valeur est une chaine de caractères.

```
        <e:workbook>
          <e:worksheet name="fooSheet">
            <e:cell column="0" row="0" value="1"/>
          </e:worksheet>
          <e:worksheet name="barSheet">
            <e:cell column="0" row="0" value="2"/>
            <e:formula column="0" row="1"
              value="fooSheet!A1+barSheet1!A1">
              <e:font fontSize="12"/>
            </e:formula>
          </e:worksheet>
        </e:workbook
>
```

définie une formule dans B2 sommant les cellules A1 dans les feuilles de calculs FooSheet et BarSheet

19.8. Les images

Les images sont regroupées dans les colonnes (pour l'itération) ou dans les feuilles de calculs (pour un placement direct en utilisant les attributs `startColumn/startRow` et `rowSpan/columnSpan`). L'étendue est optionnelet si il omit, l'image sera insérée sans être retaillée.

`<e:image>`	*Les attributes* • `startColumn` — La colonne de départ de l'image. Par défaut c'est un compteur intern. La valeur est un nombre. Notez que la valeur est basée sur un index zéro. • `startRow` — La ligne de départ de l'image. Par défaut c'est un compteur interne. La valeur est un nombre. Notez que la valeur est basée sur un index zéro. • `columnSpan` — L'étendue en colonne de l'image. Par défaut est le résultat de la largeur par défaut de l'image. La valeur est un nombre flottant. • `rowSpan` — L'étendue en ligne de l'image. Par défaut est un le résultat de la hauteur par défaut de l'image. La valeur est un nombre flottant. • `URI` — L'URI de l'image. La valeur est une chaine de caractères. *Les élements enfants* • `aucun` *Les Facets* • `aucun`

```
<e:workbook>
  <e:worksheet>
    <e:image startRow="0" startColumn="0" rowSpan="4"
      columnSpan="4" URI="http://foo.org/logo.jpg"/>
  </e:worksheet>
```

```
        </e:workbook
>
```

définie une image dans A1:E5 basée sur les données transmisses

19.9. Les hyperliens

Les hyperliens sont regroupé dans les colonnes (pour l'itération) ou dans les classeurs (pour un placement direct en utilisant les attributs `startColumn/startRow` et `endColumn/endRow`). Ils ajoutent des liens de navigations aux URIs

`<e:hyperlink>`	*Les attributes* • `startColumn` — La colonne de départ de l'hyperlien. Par défaut est un compteur interne. La valeur est un nombre. Notez que la valeur est un index basée zéro. • `startRow` — La ligne de départ de l'hyperlien. Par défaut c'est un compteur interne. La valeur est un nombre. Notez que la valeur est basée sur un index zéro. • `endColumn` — La colonne de fin de l'hyperlien. Par défaut c'est un compteur interne. La valeur est un nombre. Notez que la valeur est basée sur un index zéro. • `endRow` — La ligne de fin de l'hyperlien. Par défaut c'est un compteur interne. La valeur est un nombre. Notez que la valeur est basée sur un index zéro. • `URL` — L'URL du lien. La valeur est une chaine de caractères. • `description` — La description du lien. La valeur est une chaine de caractères. *Les élements enfants* • `aucun` *Les Facets* • `aucun`

```
        <e:workbook>
          <e:worksheet>
            <e:hyperLink startRow="0" startColumn="0" endRow="4"
              endColumn="4" URL="http://seamframework.org"
              description="The Seam Framework"/>
          </e:worksheet>
        </e:workbook
>
```

défini un hyperlien et sa description qui point vers SFWK dans la zone A1:E5

19.10. Les entêtes et les pieds de page

Les entêtes et les pieds-de-pages sont des enfants des classeurs et contients les facets qui à leur tour contiennent les commandes qui sont analysées.

`<e:header>`	*Les attributes* • `aucun` *Les élements enfants* • `aucun` *Les Facets* • `left` — Le contenu de la partie gauche de l'entête/du pied de page. • `center` — Le contenu de la partie centrale de l'entête/du pied de page. • `right` — Le contenu de la partie droite de l'entête/du pied de page.

`<e:footer>`	*Les attributes* • `aucun` *Les élements enfants* • `aucun` *Les Facets*

	• `left` — Le contenu de la partie gauche de l'entête/du pied de page. • `center` — Le contenu de la partie centrale de l'entête/du pied de page. • `right` — Le contenu de la partie droite de l'entête/du pied de page.

Le contenu de la facets est dans une chaine de caractères qui peut contenir plusieurs commandes délimitées par # comme ci-dessous:

#date#	Insère la date courante
#page_number#	Insère le numéro de la page courante
#time#	Insère l'heure actuelle
#total_pages#	Insère le nombre total de page
#worksheet_name#	Insère le nom de la feuille de calcul
#workbook_name#	Insère le nom du classeur
#bold#	Activer la police en gras, ustilisez un autre #bold# pour le désactiver.
#italics#	Activer la police en italique, ustilisez un autre #italic# pour le désactiver.
#underline#	Basculer le soulignement, utilisez un autre #underline# pour désactiver
#double_underline#	Basculez le double soulignement, utilisez un autre #double_underline# pour désactiver
#outline#	Basculez la police surlignée, utilisez un autre #outline# pour désactiver
#shadow#	Basculez la police ombrée, utilisez un autre #shadow# pour désactiver
#strikethrough#	Basculez la police barrée, utilisez un autre #strikethrough# pour désactiver
#subscript#	Basculez la police en indice, utilisez un autre #subscript# pour désactiver
#superscript#	Basculez la police en exposant, utilisez un autre #superscript# pour désactiver
#font_name#	Définir le nom de la police, utilisez quelquechose comme #font_name=Verdana"
#font_size#	Défini une taille de police, utilisez quelquechose comme #font_size=12#

```
        <e:workbook>
          <e:worksheet
>
            <e:header>
```

```
            <f:facet name="left">
              This document was made on #date# and has #total_pages# pages
            </f:facet>
            <f:facet name="right">
              #time#
            </f:facet>
          </e:header>
        <e:worksheet>
      </e:workbook>
```

19.11. Les zones d'impressions et les titres

Les zones d'impression et les titres sont les enfants des feuilles de calculs et des modèles de classeur et fournissent... zones d'impressions et titres.

`<e:printArea>`	*Les attributes* • `firstColumn` — La colonne dans le coin supérieur gauche de la zone. Le paramètre est un nombre. Notez que la valeur est un index basée sur zéro. • `firstRow` — La ligne dans le coin supérieur gauche de la zone. Le paramètre est un nombre. Notez que la valeur est un index basée sur zéro. • `lastColumn` — La colonne dans le coin inférieur droit de la zone. Le paramètre est un nombre. Notez que la valeur est basé sur un index zéro. • `lastRow` — La ligne du coin inférieur droit de la zone. Le paramètre est un nombre. Notez que la valeur est basé sur un index zéro. *Les élements enfants* • `aucun` *Les Facets* • `aucun`

```
      <e:workbook>
```

```
          <e:worksheet
>
            <e:printTitles firstRow="0" firstColumn="0"
              lastRow="0" lastColumn="9"/>
            <e:printArea firstRow="1" firstColumn="0"
              lastRow="9" lastColumn="9"/>
          </e:worksheet>
        </e:workbook>
```

définie un titre d'impression entre A1:A10 et une zone d'impression entre B2:J10.

19.12. Les commandes du classeur de calcul

Les commandes de la feuille de calculs sont les enfant des classeurs et habituellement exécuté seulement une fois.

19.12.1. Le regroupement

Fournir le regroupement de colonnes et de lignes.

`<e:groupRows>`	*Les attributes* • `startRow` — la ligne qui indique le commencent du regroupement La valeur est un nombre. Notez que la valeur est un index basé sur zéro. • `endRow` — La ligne qui indique la fin du regroupement. La valeur est un nombre. Notez que la valeur est un basé sur un index zéro. • `collapse` — Faut il que le regroupement soit replié initiallement? La valeur est un booléen. *Les élements enfants* • `aucun` *Les Facets* • `aucun`
`<e:groupColumns>`	*Les attributes* • `startColumn` — La colonne de départ du regroupement. La valeur est un nombre. Notez que la valeur est un index basé sur zéro.

	• `endColumn` — La colonne de fin du regroupement. La valeur est un nombre. Notez que la valeur est un index basé sur zéro. • `collapse` — Faut il que le regroupement soit replié initiallement? La valeur est un booléen. *Les élements enfants* • `aucun` *Les Facets* • `aucun`

```
<e:workbook>
  <e:worksheet
>
    <e:groupRows startRow="4" endRow="9" collapse="true"/>
    <e:groupColumns startColumn="0" endColumn="9" collapse="false"/>
  </e:worksheet>
</e:workbook>
```

les lignes groupées de 5 à 10 et les colonnes de 5 à 10 ainsi les lignes sont initiallement repliées (mais pas les colonnes).

19.12.2. Saut de page

Fournir des sauts de page

`<e:rowPageBreak>`	*Les attributes* • `row` — La ligne de l'endroit de saut. La valeur est un nombre. Notez que la valeur est un index basé sur zéro. *Les élements enfants* • `aucun` *Les Facets*

	• aucun

```
<e:workbook>
  <e:worksheet
>
    <e:rowPageBreak row="4"/>
  </e:worksheet>
</e:workbook
>
```

saut de page à la ligne 5.

19.12.3. La fusion

Fournir le regroupement de cellule

`<e:mergeCells>`	*Les attributes* • `startRow` — La ligne de départ pour la fusion. La valeur est un nombre. Notez que la valeur est un index basé sur zéro. • `startColumn` — La colonne de départ pour la fusion. La valeur est un nombre. Notez que la valeur est un index basé sur zéro. • `endRow` — La ligne de fin pour la fusion. La valeur est un nombre. Notez que la valeur est un index basé sur zéro. • `endColumn` — La colonne de fin de la fusion. La valeur est un nombre. Notez que la valeur est un index basé sur zéro. *Les élements enfants* • aucun *Les Facets* • aucun

```
        <e:workbook>
          <e:worksheet>
            <e:mergeCells startRow="0" startColumn="0" endRow="9" endColumn="9"/>
          </e:worksheet>
        </e:workbook
>
```

fusions des cellules dans la zone A1:J10

19.13. Exportateur de tableau de données

Si vou préférez exporter une table de données JSF existante au lieu d'écrire un document XHTML dédié, ce qui peut aussi être réalisé facilement en exécutant le composant `org.jboss.seam.excel.excelExporter.export`, en passant l'identifiant de la table de données comme un paramètre EL de Seam. Considerant que vous avez une table de données

```
<h:form id="theForm">
  <h:dataTable id="theDataTable" value="#{personList.personList}"
    var="person">
    ...
  </h:dataTable>
</h:form>
```

que vous voulez voir coomme un classeur Excel® Microsoft® . Placez un

```
<h:commandLink
  value="Export"
  action="#{excelExporter.export('theForm:theDataTable')}"
/>
```

dans le formulaire et c'est fait. VOus pouvez bien sûr exécuter l'exportateur avec un bouton s:link ou toute méthode préférée. Il y a aussi des plans pour un tag d'exportation qui peut être placé dans la balise table de données ainsi vous n'avez pas à indiquer la table de données par son ID.

Voir *Section 19.14, « Les polices et les calques »* pour le formatage.

19.14. Les polices et les calques

Le control de comment l'affichage est fait est une combinaison d'attributs de style CSS et d'attributs de tag. Les plus communs (polices, bordures, arrières plan, etc) sont CSS et quelques réglages généraux sont des attributs de tag.

Les attributs CSS en cascade depuis les parents vers les enfants et avec un tag surcharge les classes CSS référencés dans l'attribut `styleClass` et finallement surcharge les attributs CSS définis dans l'attribut `style`. Vous pouvez les placer dans presque partout mais en indiquant une largeur de colonne défini une cellule incluse dans la colonne est assez évident.

Si vous avez un masque de formatage ou des polices qui utiliser des caractères spéciaux, comme des espaces ou des points virgules, vous pouvez déspécialiser des caractères css avec le caractère " comme xls-format-mask:'$;$'

19.14.1. Les liens vers les feuilles de styles

Les feuilles de styles externes sont référencés dans la balise e:link. Ils sont placé comme enfant du classeur.

`<e:link>`	*Les attributes* • `URL` — L'URL vers la feuille de style *Les élements enfants* • `aucun` *Les Facets* • `aucun`

```
        <e:workbook>
          <e:link URL="/css/excel.css"/>
        </e:workbook
>
```

Référence une feuille de style qui peut ête trouvé dans /css/excel.css

19.14.2. Les polices

Ce groupe d'attributs XLS-CSS définit une police et ces attributs

xls-font-family	Le nom de la police. Soyez sur qu'elle est supportée par votre système.
xls-font-size	La taille de la font. Utilisez un nombre entier.
xls-font-color	La couleur de la police (voir *jxl.format.Colour* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Colour.html]).
xls-font-bold	Doit-elle être en gras? Les valeurs valides sont "true" et "false"
xls-font-italic	Doit-elle être en italique? Les valeurs valides sont "true" et "false"
xls-font-script-style	Le style de script de la police (voir *jxl.format.ScriptStyle* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/ScriptStyle.html]).
xls-font-underline-style	Le style de soulignement de la police (voir *jxl.format.UnderlineStyle* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/UnderlineStyle.html]).
xls-font-struck-out	Doit-elle être soulignée? Les valeurs valides sont "true" et "false"
xls-font	Un notation plus courte pour définir toutes ces valeurs. Indiquez le nom de la police à la fin et utilisez les graduations de la polices avec des espaces entre eux, par exemple 'Times New Roman'. Utilisez "italic", "bold" et "struckout". Example style="xls-font: red bold italic 22 Verdana"

19.14.3. Les bordures

Ce groupe d'attributs XLS-CSS déinissent les bordures de la cellule

xls-border-left-color	La couleur de la bordure pour la bordure gauche de la cellule (voir *jxl.format.Colour* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Colour.html]).
xls-border-left-line-style	Le style de la ligne de bordure de la bordure gauche de la cellule (voir *jxl.format.BorderLineStyle* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/BorderLineStyle.html]).
xls-border-left	Un raccourci pour définir le style de la ligne et la couleur de la bordure gauche de la cellule, par exemple style="xls-border-left: thick red"
xls-border-top-color	La couleur de la bordure pour la bordure suppérieure de la cellule (voir *jxl.format.Colour* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Colour.html]).
xls-border-top-line-style	

	Le style de la ligne de la bordure du bord suppérieur de la cellule (voir *jxl.format.BorderLineStyle* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/BorderLineStyle.html]).
xls-border-top	Un raccourci pour définir le style de la ligne et de la couleur de la bordure suppérieure de la cellule, par exemple style="xls-border-top: red thick"
xls-border-right-color	La couleur de la bordure de la bordure droite de la cellule (voir *jxl.format.Colour* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Colour.html]).
xls-border-right-line-style	Le style de la ligne de bordure de la bordure de droit de la cellule (voir *jxl.format.BorderLineStyle* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/BorderLineStyle.html]).
xls-border-right	Un raccourci pour définir le style de la ligne et la couleur pour la bordure droit de la cellule , par exemple style="xls-border-right: thick red"
xls-border-bottom-color	La couleur de la bordure de la bordure inférieure de la cellule (voir *jxl.format.Colour* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Colour.html]).
xls-border-bottom-line-style	Le style de ligne de bordure dans le bord inférieur de la cellule (voir *jxl.format.BorderLineStyle* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/BorderLineStyle.html]).
xls-border-bottom	Un raccourci pour définir les réglages du style de la ligne pour la bordure inférieure de la cellule , par exemple style="xls-border-bottom: thick red"
xls-border	Une version courte pour définir un style de ligne et de couleur pour toutes les bordures de la cellule, par exemple style="xls-border: thick red"

19.14.4. Arrière plan

Ce groupe d'attributs XLS-CSS définissent l'arrière plan de la cellule

xls-background-color	La couleur de l'arrière plan (voir *jxl.format.BorderLineStyle* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/BorderLineStyle.html]).
xls-background-pattern	Le patron de l'arrière plan (voir *jxl.format.Pattern* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Pattern.html]).
xls-background	Un raccourci pour définir la couleur d'arrière plan et le patron. Voir plus loiin les règles.

19.14.5. Les réglages de la colonne

Ce group d'attributs XLS-CSS définissent les largeurs de la colonne etc.

xls-column-width	La largeur de la colonne. Utilisez une valeur importante (~5000) pour commencer. Utiliser par e:column dans le mode xhtml.
xls-column-widths	La largeur de la colonne? Utilisez les valeurs importante (~5000) pour commencer. Utiliser par l'exportateur d'excel, à indiquer dans l'attribut de style de la table de données. Utilisez des valeurs numériques ou * pour laisser libre une colonne. Exemple style="xls-column-widths: 5000, 5000, *, 10000"
xls-column-autosize	Faut il avoir une taille automatique pour la colonne? Les valeurs valides sont "true" et "false".
xls-column-hidden	La colonne doit elle être cachée? Les valeurs valides sont "true" et "false".
xls-column-export	La colonne doit elle être montrée dans l'exportation? Les valeurs valides sont "true" et "false".Par défaut c'est "true".

19.14.6. Les réglages de la cellule

Ce groupe d'attributs XLS-CSS dé"finissent les propriétés de cellule

xls-alignment	L'allignement de la valeur de la cellule (voir *jxl.format.Alignment* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Alignment.html]).
xls-force-type	Le type forcé de la donnée de la cellule. La valeur est une chaine de caractères qui peut être une de celle-ci "general", "number", "text", "date", "formula" or "bool". Le type est automatiquement détecté donc c'est rare de devoir utiliser cet attribut.
xls-format-mask	Le masque de format de la cellule, voir *Section 19.6.2, « Masque de formatage »*
xls-indentation	L'indentation de la valeur de la cellule. La valeur est un nombre.
xls-locked	Faut-il que la cellule soit vérouillée. A utiliser avec le niveau de vérouillage du classeur. Les valeurs valides sont "true" et "false".
xls-orientation	L'orientation de la valeur de la cellule (voir *jxl.format.Orientation* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/Orientation.html]).
xls-vertical-alignment	L'alignement verticale de la valeur de cellule (voir *jxl.format.VerticalAlignment* [http://jexcelapi.sourceforge.net/resources/javadocs/current/docs/jxl/format/VerticalAlignment.html]).
xls-shrink-to-fit	Les valeurs de la cellules doivent être réduite pour correspondre? Les valeurs valides sont "true" et "false".
xls-wrap	La cellule doit elle s'étendre avec des nouvelles lignes ? Les valeurs valides sont "true" et "false".

19.14.7. L'exportateur de base de données

L'exportateur de la table de données utilise les même attributs xls-css comme document xhtml avec l'exception suivante que les largeurs de colonnes sont définies avec l'attribut`xls-column-widths` de la table de données (car UIColumn ne permet pas d'avoir les attributs style ou styleClass).

19.14.8. Les exemples de calques

A FAIRE

19.14.9. Les limitations

Dans la version actuelle, il y a quelques limitations connues par rapport au support de CSS

- Avec l'utilisation de documents .xhtml, les feuilles de styles doivent être référencés au travers d'une balise `<e:link>`
- Avec l'utilisation de l'exportateur de table de donné, le CSS doit être entré en mode d'attributs, les feuilles de styles externes ne sont pas supportées

19.15. Internationalisation

Il ya seulement deux clefs livrées utilisées, les deux pour des formatage de données invalides et les deux prennent une paramètree (la valeur invalide)

- `org.jboss.seam.excel.not_a_number` — Quand une valeur doit être un nombre et ne peut être traité en temps que tel
- `org.jboss.seam.excel.not_a_date` — Quand une valeur doit être une date ne peux être traité en tant que telle

19.16. Les liens et de la documentation additionnelle

Le coeur des fonctionnalitées de the Microsoft® Excel® spreadsheet application est basé sur l'excelente bibliothèque JExcelAPI qui peut être trouvé sur *http://jexcelapi.sourceforge.net/* [http://jexcelapi.sourceforge.net] et la plus part des fonctionnalités et les limitations possibles ont été héritées.

Si vous utiliser le forum ou les listes de diffusions, merci de vous rappelez qu'ils ne connaissent rien à propos de Seam et de l'utilisation de ses bibliothèques, tout problèmes doivent être indiqués dans le JBoss Seam JIRA dans le module "excel".

RSS support

It is now easy to integrate RSS feeds in Seam through the *YARFRAW* [http://yarfraw.sourceforge.net/] library. The RSS support is currently in the state of "tech preview" in the current release.

20.1. Installation

To enable RSS support, include the `jboss-seam-rss.jar` in your applications `WEB-INF/lib` directory. The RSS library also has some dependent libraries that should be placed in the same directory. See *Section 42.2.6, « Le support des RSS de Seam »* for a list of libraries to include.

The Seam RSS support requires the use of Facelets as the view technology.

20.2. Generating feeds

The `examples/rss` project contains an example of RSS support in action. It demonstrates proper deployment packaging, and it shows the exposed functionality.

A feed is a xhtml-page that consist of a feed and a list of nested entry items.

```
<r:feed
  xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:r="http://jboss.com/products/seam/rss"
  title="#{rss.feed.title}"
  uid="#{rss.feed.uid}"
  subtitle="#{rss.feed.subtitle}"
  updated="#{rss.feed.updated}"
  link="#{rss.feed.link}">
  <ui:repeat value="#{rss.feed.entries}" var="entry">
    <r:entry
      uid="#{entry.uid}"
      title="#{entry.title}"
      link="#{entry.link}"
      author="#{entry.author}"
      summary="#{entry.summary}"
      published="#{entry.published}"
      updated="#{entry.updated}"
        />
   </ui:repeat>
</r:feed>
```

20.3. Feeds

Feeds are the top-level entities that describe the properties of the information source. It contains zero or more nested entries.

`<r:feed>`	*Attributes* • `uid` — An optional unique feed id. The value is a string. • `title` — The title of the feed. The value is a string. • `subtitle` — The subtitle of the feed. The value is a string. • `updated` — When was the feed updated? The value is a date. • `link` — The link to the source of the information. The value is a string. • `feedFormat` — The feed format. The value is a string and defaults to ATOM1. Valid values are RSS10, RSS20, ATOM03 and ATOM10. *Child elemenents* • `Zero or more feed entries` *Facets* • `none`

20.4. Entries

Entries are the "headlines" in the feed.

`<r:feed>`	*Attributes* • `uid` — An optional unique entry id. The value is a string. • `title` — The title of the entry. The value is a string. • `link` — A link to the item. The value is a string. • `author` — The author of the story. The value is a string. • `summary` — The body of the story. The value is a string.

	• `textFormat` — The format of the body and title of the story. The value is a string and valid values are "text" and "html". Defaults to "html". • `published` — When was the story first published? The value is a date. • `updated` — When was the story updated? The value is a date. *Child elemenents* • `none` *Facets* • `none`

20.5. Links and further documentation

The core of the RSs functionality is based on the YARFRAW library which can be found on *http://yarfraw.sourceforge.net/* and most features and possible limitations are inherited from here.

For details on the ATOM 1.0 format, have a look at *the specs* [http://atompub.org/2005/07/11/draft-ietf-atompub-format-10.html]

For details on the RSS 2.0 format, have a look at *the specs* [http://cyber.law.harvard.edu/rss/rss.html]

Email

Seam now includes an optional components for templating and sending emails.

Email support is provided by `jboss-seam-mail.jar`. This JAR contains the mail JSF controls, which are used to construct emails, and the `mailSession` manager component.

The examples/mail project contains an example of the email support in action. It demonstrates proper packaging, and it contains a number of example that demonstrate the key features currently supported.

You can also test your mail's using Seam's integration testing environment. See *Section 37.3.4, « Integration Testing Seam Mail »*.

21.1. Creating a message

You don't need to learn a whole new templating language to use Seam Mail — an email is just facelet!

```
<m:message xmlns="http://www.w3.org/1999/xhtml"
   xmlns:m="http://jboss.com/products/seam/mail"
   xmlns:h="http://java.sun.com/jsf/html">

   <m:from name="Peter" address="peter@example.com" />
   <m:to name="#{person.firstname} #{person.lastname}">#{person.address}</m:to>
   <m:subject>Try out Seam!</m:subject>

   <m:body>
      <p><h:outputText value="Dear #{person.firstname}" />,</p>
      <p>You can try out Seam by visiting
      <a href="http://labs.jboss.com/jbossseam">http://labs.jboss.com/jbossseam</a>.</p>
      <p>Regards,</p>
      <p>Pete</p>
   </m:body>

</m:message>
```

The `<m:message>` tag wraps the whole message, and tells Seam to start rendering an email. Inside the `<m:message>` tag we use an `<m:from>` tag to set who the message is from, a `<m:to>` tag to specify a sender (notice how we use EL as we would in a normal facelet), and a `<m:subject>` tag.

The `<m:body>` tag wraps the body of the email. You can use regular HTML tags inside the body as well as JSF components.

So, now you have your email template, how do you go about sending it? Well, at the end of rendering the `m:message` the `mailSession` is called to send the email, so all you have to do is ask Seam to render the view:

```
@In(create=true)
private Renderer renderer;

public void send() {
   try {
      renderer.render("/simple.xhtml");
      facesMessages.add("Email sent successfully");
   }
   catch (Exception e) {
      facesMessages.add("Email sending failed: " + e.getMessage());
   }
}
```

If, for example, you entered an invalid email address, then an exception would be thrown, which is caught and then displayed to the user.

21.1.1. Attachments

Seam makes it easy to attach files to an email. It supports most of the standard java types used when working with files.

If you wanted to email the `jboss-seam-mail.jar`:

```
<m:attachment value="/WEB-INF/lib/jboss-seam-mail.jar"/>
```

Seam will load the file from the classpath, and attach it to the email. By default it would be attached as `jboss-seam-mail.jar`; if you wanted it to have another name you would just add the `fileName` attribute:

```
<m:attachment value="/WEB-INF/lib/jboss-seam-mail.jar" fileName="this-is-so-cool.jar"/>
```

You could also attach a `java.io.File`, a `java.net.URL`:

```
<m:attachment value="#{numbers}"/>
```

Or a `byte[]` or a `java.io.InputStream`:

```
<m:attachment value="#{person.photo}" contentType="image/png"/>
```

You'll notice that for a `byte[]` and a `java.io.InputStream` you need to specify the MIME type of the attachment (as that information is not carried as part of the file).

And it gets even better, you can attach a Seam generated PDF, or any standard JSF view, just by wrapping a `<m:attachment>` around the normal tags you would use:

```
<m:attachment fileName="tiny.pdf">
   <p:document>
      A very tiny PDF
   </p:document>
</m:attachment>
```

If you had a set of files you wanted to attach (for example a set of pictures loaded from a database) you can just use a `<ui:repeat>`:

```
<ui:repeat value="#{people}" var="person">
                        <m:attachment value="#{person.photo}" contentType="image/jpeg"
 fileName="#{person.firstname}_#{person.lastname}.jpg"/>
</ui:repeat>
```

And if you want to display an attached image inline:

```
<m:attachment
   value="#{person.photo}"
   contentType="image/jpeg"
   fileName="#{person.firstname}_#{person.lastname}.jpg"
   status="personPhoto"
   disposition="inline" />
<img src="cid:#{personPhoto.contentId}" />
```

You may be wondering what `cid:#{...}` does. Well, the IETF specified that by putting this as the src for your image, the attachments will be looked at when trying to locate the image (the `Content-ID`'s must match) — magic!

You must declare the attachment before trying to access the status object.

21.1.2. HTML/Text alternative part

Whilst most mail readers nowadays support HTML, some don't, so you can add a plain text alternative to your email body:

```
<m:body>
   <f:facet name="alternative">Sorry, your email reader can't show our fancy email,
please go to http://labs.jboss.com/jbossseam to explore Seam.</f:facet>
</m:body>
```

21.1.3. Multiple recipients

Often you'll want to send an email to a group of recipients (for example your users). All of the recipient mail tags can be placed inside a `<ui:repeat>`:

```
<ui:repeat value="#{allUsers} var="user">
   <m:to name="#{user.firstname} #{user.lastname}" address="#{user.emailAddress}" />
</ui:repeat>
```

21.1.4. Multiple messages

Sometimes, however, you need to send a slightly different message to each recipient (e.g. a password reset). The best way to do this is to place the whole message inside a `<ui:repeat>`:

```
<ui:repeat value="#{people}" var="p">
   <m:message>
      <m:from name="#{person.firstname} #{person.lastname}">#{person.address}</m:from>
      <m:to name="#{p.firstname}">#{p.address}</m:to>
         ...
   </m:message>
</ui:repeat>
```

21.1.5. Templating

The mail templating example shows that facelets templating just works with the Seam mail tags.

Our `template.xhtml` contains:

```
<m:message>
  <m:from name="Seam" address="do-not-reply@jboss.com" />
```

```
  <m:to name="#{person.firstname} #{person.lastname}">#{person.address}</m:to>
  <m:subject>#{subject}</m:subject>
  <m:body>
     <html>
        <body>
           <ui:insert name="body">This is the default body, specified by the template.</ui:insert>
        </body>
     </html>
  </m:body>
</m:message>
```

Our `templating.xhtml` contains:

```
<ui:param name="subject" value="Templating with Seam Mail"/>
<ui:define name="body">
   <p>This example demonstrates that you can easily use <i>facelets templating</i> in email!</p>
</ui:define>
```

You can also use facelets source tags in your email, but you must place them in a jar in `WEB-INF/lib` - referencing the `.taglib.xml` from `web.xml` isn't reliable when using Seam Mail (if you send your mail asynchrounously Seam Mail doesn't have access to the full JSF or Servlet context, and so doesn't know about `web.xml` configuration parameters).

If you do need more configure Facelets or JSF when sending mail, you'll need to override the Renderer component and do the configuration programmatically - only for advanced users!

21.1.6. Internationalisation

Seam supports sending internationalised messages. By default, the encoding provided by JSF is used, but this can be overridden on the template:

```
<m:message charset="UTF-8">
   ...
</m:message>
```

The body, subject and recipient (and from) name will be encoded. You'll need to make sure facelets uses the correct charset for parsing your pages by setting encoding of the template:

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

21.1.7. Other Headers

Sometimes you'll want to add other headers to your email. Seam provides support for some (see *Section 21.5, « Tags »*). For example, we can set the importance of the email, and ask for a read receipt:

```
<m:message xmlns:m="http://jboss.com/products/seam/mail"
   importance="low"
   requestReadReceipt="true"/>
```

Otherwise you can add any header to the message using the `<m:header>` tag:

```
<m:header name="X-Sent-From" value="JBoss Seam"/>
```

21.2. Receiving emails

If you are using EJB then you can use a MDB (Message Driven Bean) to receive email. JBoss provides a JCA adaptor — `mail-ra.rar` — but the version distributed with JBoss AS 4.x has a number of limitations (and isn't bundled in some versions) therefore we recommend using the `mail-ra.rar` distributed with Seam (it's in the `extras/` directory in the Seam bundle). `mail-ra.rar` should be placed in `$JBOSS_HOME/server/default/deploy`; if the version of JBoss AS you use already has this file, replace it.

Note

JBoss AS 5.x and newer has mail-ra.rar applied the patches, so there is no need to copy the mail-ra.rar from Seam distribution.

You can configure it like this:

```
@MessageDriven(activationConfig={
   @ActivationConfigProperty(propertyName="mailServer", propertyValue="localhost"),
   @ActivationConfigProperty(propertyName="mailFolder", propertyValue="INBOX"),
   @ActivationConfigProperty(propertyName="storeProtocol", propertyValue="pop3"),
   @ActivationConfigProperty(propertyName="userName", propertyValue="seam"),
   @ActivationConfigProperty(propertyName="password", propertyValue="seam")
})
@ResourceAdapter("mail-ra.rar")
@Name("mailListener")
public class MailListenerMDB implements MailListener {
```

```
   @In(create=true)
   private OrderProcessor orderProcessor;

   public void onMessage(Message message) {
     // Process the message
     orderProcessor.process(message.getSubject());
   }

}
```

Each message received will cause `onMessage(Message message)` to be called. Most Seam annotations will work inside a MDB but you musn't access the persistence context.

You can find more information on `mail-ra.rar` at *http://www.jboss.org/community/wiki/InboundJavaMail*.

If you aren't using JBoss AS you can still use `mail-ra.rar` or you may find your application server includes a similar adapter.

21.3. Configuration

To include Email support in your application, include `jboss-seam-mail.jar` in your `WEB-INF/lib` directory. If you are using JBoss AS there is no further configuration needed to use Seam's email support. Otherwise you need to make sure you have the JavaMail API, an implementation of the JavaMail API present (the API and impl used in JBoss AS are distributed with seam as `lib/mail.jar`), and a copy of the Java Activation Framework (distributed with Seam as `lib/activation.jar`.

Note

The Seam Mail 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 `mailSession` component uses JavaMail to talk to a 'real' SMTP server.

21.3.1. `mailSession`

A JavaMail Session may be available via a JNDI lookup if you are working in an JEE environment or you can use a Seam configured Session.

The mailSession component's properties are described in more detail in *Section 32.9, « Mail-related components »*.

21.3.1.1. JNDI lookup in JBoss AS

The JBossAS `deploy/mail-service.xml` configures a JavaMail session binding into JNDI. The default service configuration will need altering for your network. *http://www.jboss.org/community/wiki/JavaMail* describes the service in more detail.

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

   <mail:mail-session session-jndi-name="java:/Mail"/>

</components>
```

Here we tell Seam to get the mail session bound to `java:/Mail` from JNDI.

21.3.1.2. Seam configured Session

A mail session can be configured via `components.xml`. Here we tell Seam to use `smtp.example.com` as the smtp server:

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

   <mail:mail-session host="smtp.example.com"/>

</components>
```

21.4. Meldware

Seam's mail examples use Meldware (from *buni.org* [http://buni.org]) as a mail server. Meldware is a groupware package that provides `SMTP`, `POP3`, `IMAP`, webmail, a shared calendar and an graphical admin tool; it's written as a JEE application so can be deployed onto JBoss AS alongside your Seam application.

Attention

The version of Meldware distributed with Seam (downloaded on demand) is specially tailored for development - mailboxes, users and aliases (email addresses)

are created every time the application deploys. If you want to use Meldware in production you should install the latest release from *buni.org* [http://buni.org].

21.5. Tags

Emails are generated using tags in the `http://jboss.com/products/seam/mail` namespace. Documents should always have the `message` tag at the root of the message. The message tag prepares Seam to generate an email.

The standard templating tags of facelets can be used as normal. Inside the body you can use any JSF tag; if it requires access to external resources (stylesheets, javascript) then be sure to set the `urlBase`.

<m:message>
Root tag of a mail message

- `importance` — low, normal or high. By default normal, this sets the importance of the mail message.
- `precedence` — sets the precedence of the message (e.g. bulk).
- `requestReadReceipt` — by default false, if set, a read receipt request will be will be added, with the read receipt being sent to the `From:` address.
- `urlBase` — If set, the value is prepended to the `requestContextPath` allowing you to use components such as `<h:graphicImage>` in your emails.
- `messageId` — Sets the Message-ID explicitly

<m:from>
Set's the From: address for the email. You can only have one of these per email.

- `name` — the name the email should come from.
- `address` — the email address the email should come from.

<m:replyTo>
Set's the Reply-to: address for the email. You can only have one of these per email.

- `address` — the email address the email should come from.

<m:to>
Add a recipient to the email. Use multiple <m:to> tags for multiple recipients. This tag can be safely placed inside a repeat tag such as <ui:repeat>.

- `name` — the name of the recipient.
- `address` — the email address of the recipient.

<m:cc>
Add a cc recipient to the email. Use multiple <m:cc> tags for multiple ccs. This tag can be safely placed inside a iterator tag such as <ui:repeat>.

- `name` — the name of the recipient.
- `address` — the email address of the recipient.

<m:bcc>
Add a bcc recipient to the email. Use multiple <m:bcc> tags for multiple bccs. This tag can be safely placed inside a repeat tag such as <ui:repeat>.

- `name` — the name of the recipient.
- `address` — the email address of the recipient.

<m:header>
Add a header to the email (e.g. `X-Sent-From: JBoss Seam`)

- `name` — The name of the header to add (e.g. `X-Sent-From`).
- `value` — The value of the header to add (e.g. `JBoss Seam`).

<m:attachment>
Add an attachment to the email.

- `value` — The file to attach:
  - `String` — A `String` is interpreted as a path to file within the classpath
  - `java.io.File` — An EL expression can reference a `File` object
  - `java.net.URL` — An EL expression can reference a `URL` object
  - `java.io.InputStream` — An EL expression can reference an `InputStream`. In this case both a `fileName` and a `contentType` must be specified.
  - `byte[]` — An EL expression can reference an `byte[]`. In this case both a `fileName` and a `contentType` must be specified.

  If the value attribute is ommitted:

  - If this tag contains a `<p:document>` tag, the document described will be generated and attached to the email. A `fileName` should be specified.
  - If this tag contains other JSF tags a HTML document will be generated from them and attached to the email. A `fileName` should be specified.
- `fileName` — Specify the file name to use for the attached file.
- `contentType` — Specify the MIME type of the attached file

<m:subject>

Set's the subject for the email.

<m:body>

Set's the body for the email. Supports an `alternative` facet which, if an HTML email is generated can contain alternative text for a mail reader which doesn't support html.

- `type` — If set to `plain` then a plain text email will be generated otherwise an HTML email is generated.

Asynchronisme et messagerie

Seam rends vraiment très simple de réaliser du travail de manière asynchrone depuis une requête web. Quand la plus part des gens pensent asynchronisme en Java EE, ils pensent à l'utilisation de JMS. C'est certainement un bonne façon d'approche le problème avec Seam, et c'est la bonne raison quand vous avez une qualité de service stricte et bien définie dans les prérequis de service. Seam rends facile d'envoyer et de recevoir des messages JMS en utilisant les composants de Seam.

Mais pour la plus part des cas d'utilisation, JSM est excessif. Les couches de Seam une méthode symple asynchrone et une fonction évènementiel par dessus votre choix d'un *dispatchers*:

- `java.util.concurrent.ScheduledThreadPoolExecutor` (par défaut)
- le service de temps EJB (pour les environements EJB 3.0)
- Quartz

Ce chapitre couvre en premier lieu comment exploiter Seam pour simplifier JMS et ensuite explique comment utiliser la méthode assynchrone la plus simple et la fonction évènementielle.

22.1. Messagerie dans Seam

Seam rends facile d'envoyer et de recevoir des messages JMS depuis et vers des composants de Seam. A la fois le publiant du message et le destinataire du message peuvent être des composants de Seam.

Vous allez en premier apprendre à configurer une file d'attente et un publiant de sujet de message et ensuite regarder les exemples qui illustre comment réaliser l'échange de messages.

22.1.1. La configuration

Pour configurer l'infrastructure de Seam pour l'expédition de messages JMS, vous avez besoin d'indiquer à Seam à propos de quel sujets et de quelle file d'attente vous voulez que les messages soient envoyés et aussi dire à Seam où il peut trouvrer le `QueueConnectionFactory` et/ou le `TopicConnectionFactory`.

Seam utilisant par défaut Seam `UIL2ConnectionFactory` qui est une fabrique de connection usuelle à utiliser avec JBossMQ. Si vous voulez utiliser un autre fournisseur de JMS, vous avez besoin de définir un ou deux `queueConnection.queueConnectionFactoryJndiName` et `topicConnection.topicConnectionFactoryJndiName` dans `seam.properties`, `web.xml` ou `components.xml`.

ous avez aussi besoin de lister les sujets et les files d'attentes dans `components.xml` pour installer `TopicPublishers` et `QueueSender` gérés par Seam:

```
<jms:managed-topic-publisher name="stockTickerPublisher"
```

```
                    auto-create="true"
                    topic-jndi-name="topic/stockTickerTopic"/>

<jms:managed-queue-sender name="paymentQueueSender"
                  auto-create="true"
                  queue-jndi-name="queue/paymentQueue"/>
```

22.1.2. L'expédition de messages

Maintenant, vous pouvez injecter un `TopicPublisher` et un `TopicSession` de JMS dans chaque composant de Seam pour publier un objet dans un sujet:

```
@Name("stockPriceChangeNotifier")
public class StockPriceChangeNotifier
{
  @In private TopicPublisher stockTickerPublisher;

  @In private TopicSession topicSession;

  public void publish(StockPrice price)
  {
     try
     {
       stockTickerPublisher.publish(topicSession.createObjectMessage(price));
     }
     catch (Exception ex)
     {
       throw new RuntimeException(ex);
     }
  }
}
```

Ou, dans une file d'attente:

```
@Name("paymentDispatcher")
public class PaymentDispatcher
{
  @In private QueueSender paymentQueueSender;

  @In private QueueSession queueSession;

  public void publish(Payment payment)
```

```
  {
    try
    {
      paymentQueueSender.send(queueSession.createObjectMessage(payment));
    }
    catch (Exception ex)
    {
      throw new RuntimeException(ex);
    }
  }
}
```

22.1.3. Réception des messages en utilisant tout bean conducteur de message

Vous pouvez manipuler les messages en utilisant tout EJB3 conducteur de message. Les beans conducteur de message peuvent même être des composants de Seam et dans ce cas il est possible d'injecter d'autres évènements et d'autres composants de Seam d'étendue applicatif. Voici un exemple de la réception d'un paiement qui délègue vers un processude paiement.

Note

Vous allez avoir besoin de créer un attribut avec l'annotation `@In` à vrai (i.e. create = true) pour faire que Seam créer une instance du composant injecté. Ce n'est pas nécéssaire si le composant dispose de l'auto-création (autrement dit, s'il est annoté avec `@Autocreate`).

En premier lieu, créez un MDB pour recevoir le message.

```
@MessageDriven(activationConfig = {
   @ActivationConfigProperty(
      propertyName = "destinationType",
      propertyValue = "javax.jms.Queue"
   ),
   @ActivationConfigProperty(
      propertyName = "destination",
      propertyValue = "queue/paymentQueue"
   )
})
@Name("paymentReceiver")
public class PaymentReceiver implements MessageListener
{
```

```
  @Logger private Log log;

  @In(create = true) private PaymentProcessor paymentProcessor;

  @Override
  public void onMessage(Message message)
  {
    try
    {
      paymentProcessor.processPayment((Payment) ((ObjectMessage) message).getObject());
    }
    catch (JMSException ex)
    {
      log.error("Message payload did not contain a Payment object", ex);
    }
  }
}
```

Ensuite, implémentz le composant de Seam vers le quel la réception de la délégation du processus de paiement.

```
@Name("paymentProcessor")
public class PaymentProcessor
{
  @In private EntityManager entityManager;

  public void processPayment(Payment payment)
  {
    // perhaps do something more fancy
    entityManager.persist(payment);
  }
}
```

Si vous essayeez de réaliser des opérations de transactions dans votre MDB, vous devriez vous assurer que vous travailler avec une source de données XA.Sinon, il ne sera pas possible d'invalider les modifications de la base de données si la validation de la transaction de la base de données et que les opérations sousjacentes qui sont réalisées par le message échouent.

22.1.4. La réception des message dans le client

Le Seam Remoting vous permet de vous abonner à un sujet JMS depuis le JavaScript côté client. Ceci est décrit dans *Chapitre 25, Remoting*.

22.2. Asynchronisme

Les évènements assynchrones et les appels de méthodes ont la même attente en qualité de services que le mécanisme de distribution sousjacent. Le distributeur par défaut, basé sur un `ScheduledThreadPoolExecutor` réalise de manière efficace mais ne fournissent pas de support pour les tâches assynchrones persistances et ne donne aucune garantie que la tâche ne sera réellement exécutée. Si vous travaillez dans un environnement qui supporte EJB 3.0 et ajoutez la ligne suivante à `components.xml`:

```
<async:timer-service-dispatcher/>
```

ainsi votre tâche assynchrone sera réalisée par le service de temps EJB du containeur. Si vous n'êtes pas familier avec le service Timer, ne vous inquiétez pas, vous n'allez pas voir besoin d'interragir directement si vous vouler utiliser les méthodes de manière assynchrone dans Seam. La chose importante à savoir et que tout bonne implémentation de EJB 3.0 aura l'option d'utiliser le cadenseur de persistance ce qui donne quelques garanties que la tâches sera éventuellement exécutée.

Une autre alternative est d'utiliser la bibliothèque opensource Quartz pour gérer la méthode assynchrone. Vous avez besoin de fournir le JAR de la bibliothèque Quartz (à mettre dans le dossier `lib`) dans votre EAR et de le déclarer dans un module Java dans `application.xml`. Le distributeur Quartz peut être configuré en ajoutant un fichier de propriété Quartz dans le chemin des clases. Il doit être dénomé `seam.quartz.properties`. De plus, vous allez avoir besoin d'ajouter la ligne suivante à `components.xml` pour installer le distributeur Quartz.

```
<async:quartz-dispatcher/>
```

L'API de Seam par défaut `ScheduledThreadPoolExecutor`, le `Timer` EJB3, et le `Scheduler` Quartzz sont largement identique. Ils peuvent simplement être "plug and play" en ajoutant une ligne à `components.xml`.

22.2.1. Le sméthodes assynchrones

Dans le formulaire le plus simple, un appel assynchrone permet simplement un appel d'une méthode être assynchrone (dans un processus d'exécution déifférent) depuis l'appelant. Nous utilisons habituellement un appel assynchrone quand nous voulons retourner une réponse immédaite au client, et avoir quelques travaux couteux être réalisé en arrière plan. Ce modèle fonctionne très bien dans les applications qui utilisent AHAX? quand le client peut être automatiquement questionner le serveur pour le résultat du travail.

Pour les composants EJB, nous annotons l'interface local pour spécifier qu'une méthode est un processus assynchrone.

```
@Local
public interface PaymentHandler
{
   @Asynchronous
   public void processPayment(Payment payment);
}
```

(Pour les composants JavaBean nous pouvons annoter la classe d'implémentation du composant si vous voulez.)

L'utilisation de l'assynchronisme est transparente à la classe du bean;

```
@Stateless
@Name("paymentHandler")
public class PaymentHandlerBean implements PaymentHandler
{
   public void processPayment(Payment payment)
   {
      //do some work!
   }
}
```

et aussi transparent pour le 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";
   }
}
```

La méthode assynchrone est exécuté dans un contexte d'évènement complètement nouveau et n'a pas d'accès à la session ou l'état du contexte de conversation de l'appelant. Cependant le contexte du processus métier *est* propagé.

Les appels de méthodes assynchrone peuvent être programmé pour une exécution décalée dans le temps en utilisant les annotaiotns `@Duration`, `@Expiration` et `@IntervalDuration`.

```
@Local
public interface PaymentHandler
{
   @Asynchronous
   public void processScheduledPayment(Payment payment, @Expiration Date date);

   @Asynchronous
   public void processRecurringPayment(Payment payment,
                                       @Expiration Date date,
                                       @IntervalDuration Long interval)'
}
```

```
@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";
   }
}
```

A la fois le client et le serveur peuvent avoir accès à l'objet `Timer` associé avec l'invocation. L'objet `Timer` visible ci-dessous est un timer EJB3 quand vous utilisez le dispatcher EJB3. Par défaut avec `ScheduledThreadPoolExecutor`, l'objet retourné est `Future` du JDK. Pour le dispatcher Quartz, il retourne le `QuartzTriggerHandle`, qui sera vu dans la prochaine section.

```
@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";
   }
}
```

Les méthodes assynchrones peuvent retourner une autre valeur à l'appelant.

22.2.2. Les méhtodes assynchrones avec le Dispatcher Quartz

Le dispatcher Quartz (voir plus haut sur comment l'installer) vous permet d'utiliser les annotations `@Asynchronous`, `@Duration`, `@Expiration`, et `@IntervalDuration` vue au-dessus. Mais il y a quelques fonctionnalités additionnelles puissantes. Le dispatcher Quartz supporte les trois nouvelles annotations.

L'annotation `@FinalExpiration` indique une date de fin pour la tâche récurrante. Notez que vous pouvez injecter le `QuartzTriggerHandle`.

```
    @In QuartzTriggerHandle timer;

// Defines the method in the "processor" component
@Asynchronous
public QuartzTriggerHandle schedulePayment(@Expiration Date when,
                    @IntervalDuration Long interval,
                    @FinalExpiration Date endDate,
                    Payment payment)
{
    // do the repeating or long running task until endDate
}

... ...

// Schedule the task in the business logic processing code
// Starts now, repeats every hour, and ends on May 10th, 2010
Calendar cal = Calendar.getInstance ();
cal.set (2010, Calendar.MAY, 10);
processor.schedulePayment(new Date(), 60*60*1000, cal.getTime(), payment);
```

Notez que la méthode retourne un objet `QuartzTriggerHandle`, qui vous pouvez utiliser pour arréter, mettre en pause et reprendre le plannificateur. L'objet `QuartzTriggerHandle` est sérialisable, ainsi vous pouvez le sauver dans la base de données si vous avez besoin de le conserver sous le coude pour une période de temps un peu plus longue.

```
QuartzTriggerHandle handle =
      processor.schedulePayment(payment.getPaymentDate(),
                    payment.getPaymentCron(),
                    payment);
    payment.setQuartzTriggerHandle( handle );
    // Save payment to DB
```

```
    // later ...

    // Retrieve payment from DB
    // Cancel the remaining scheduled tasks
    payment.getQuartzTriggerHandle().cancel();
```

L'annotation `@IntervalCron` supporte a syntaxe des tâches cron d'Unix pour la planification des tâches. Par exemple, la méthode assynchrone suivante s'exécutera à 14h10 et at 14h44 chaque mercredi du mois de Mars.

```
// Define the method
@Asynchronous
public QuartzTriggerHandle schedulePayment(@Expiration Date when,
                    @IntervalCron String cron,
                    Payment payment)
{
   // do the repeating or long running task
}

... ...

// Schedule the task in the business logic processing code
QuartzTriggerHandle handle =
  processor.schedulePayment(new Date(), "0 10,44 14 ? 3 WED", payment);
```

L'annotation `@IntervalBusinessDay` supporte l'invocation du scénario "n-ième jour ouvré" . Par exemple, la méthode assynchrone suivante sera exécuté à 14h00 le deuxième jour ouvré de chaque mois. Par défaut, cela exclu tous les week-end et les vacances fédérales des USA jusqu'en 2010 des jours ouvrés.

```
// Define the method
@Asynchronous
public QuartzTriggerHandle schedulePayment(@Expiration Date when,
                    @IntervalBusinessDay NthBusinessDay nth,
                    Payment payment)
{
   // do the repeating or long running task
}

... ...
```

```
// Schedule the task in the business logic processing code
QuartzTriggerHandle handle =
  processor.schedulePayment(new Date(),
    new NthBusinessDay(2, "14:00", WEEKLY), payment);
```

L'objet `NthBusinessDay` contient la configuration du déclencheur de l'invocation. Vous pouvez indiquer plus de vacances (par exemple, les vacances spécifiques à l'entreprise, des vacances autre que pour les USA, etc.) via la propriété `additionalHolidays`.

```
public class NthBusinessDay implements Serializable
{
    int n;
    String fireAtTime;
    List <Date
> additionalHolidays;
    BusinessDayIntervalType interval;
    boolean excludeWeekends;
    boolean excludeUsFederalHolidays;

    public enum BusinessDayIntervalType { WEEKLY, MONTHLY, YEARLY }

    public NthBusinessDay ()
    {
      n = 1;
      fireAtTime = "12:00";
      additionalHolidays = new ArrayList <Date
> ();
      interval = BusinessDayIntervalType.WEEKLY;
      excludeWeekends = true;
      excludeUsFederalHolidays = true;
    }
    ... ...
}
```

Les annotations `@IntervalDuration`, `@IntervalCron`, et `@IntervalNthBusinessDay` sont mutuellement exclusives. Si elles sont utilisées sur la même méthode, une `RuntimeException` sera déclenchée.

22.2.3. Les évènements assynchrones

Les évènements conducteurs de composants peuvent aussi être assynchrone. Pour déclencher un évènement pour une exécution assynchrone, un simple appel à la méthode `raiseAsynchronousEvent()` de la classe `Events`. Pour programmer un évènement dans le temps, l'appel à la méthode `raiseTimedEvent()`, en passant un objet *schedule* (pour le dispatcher par défaut ou pour le dispatcher de service de temps, utilisez `TimerSchedule`). Les composants peuvent observer des évènements assynchrones de la façon usuelle, mais souvenez vous que seule le contexte du processus métier est propagé dans le processus assynchrone.

22.2.4. La gestion des exceptions pour les appels assynchrones

Chaque dispatcher assynchrone fonctionne différemment quand une exception est propagé au travers. Par exemple, le dispatcher `java.util.concurrent` suspendra de future exécutions d'un appel qui se répète, et le service timer EJB3 avalera l'exception. Seam cependant capture toute les exceptions qui seront propagées à l'extérieur de l'appel assynchrone avant qu'elle n'atteigne le dispatcher.

Par défaut, tout exception qui sera propagée à l'extérieur de l'exécution assynchrone sera intercepté et enregistré dans le journal au niveau erreur. Vous pouvez personnaliser cette fonctionnalité globallement en surchargeant le composant `org.jboss.seam.async.asynchronousExceptionHandler`:

```
@Scope(ScopeType.STATELESS)
@Name("org.jboss.seam.async.asynchronousExceptionHandler")
public class MyAsynchronousExceptionHandler extends AsynchronousExceptionHandler {

  @Logger Log log;

  @In Future timer;

  @Override
  public void handleException(Exception exception) {
    log.debug(exception);
    timer.cancel(false);
  }

}
```

Voici, par exemple, en utilisant le dispatcher `java.util.concurrent`, nous l'injectons l'objet de controle et annulons toute invocation future quand une exception est rencontrée

Vous pouvez aussi altérer cette fonctionnalité pour un seul composant en implémentant la méthode `public void handleAsynchronousException(Exception exception);` du composant. Par exemple:

```
public void handleAsynchronousException(Exception exception) {
  log.fatal(exception);
}
```

Mise en cache

Dans presque toutes les applications d'entreprise, la base de données est le premier goulet d'étranglement et le moins scalable tierce-partie de l'environement d'exécution. Les gens venant des environements PHP/Ruby vont essayer de vous dire que les architectures surnommées "ne partage rien" se dimensionnent bien. Et bien c'est peut être littérallement vrai, je ne sait pas combien d'application intéréssantes multi-utilisateurs peuvent être implémentées avec aucun partage de ressources entre les différentes noeuds du cluster. Que pensent vraiment ces personnes loufoques d'une architecture "ne rien partager sauf la base de données". Bien sur, le partage de la base de données est un problème principal quand on dimensionne une application multi utilisateur donc réclammer que cette architecture soit hautement dimensionnable est absurde et dites vous que c'est pas mal de ce type d'application sur les quelles ces gens passent la plus part du temps à travailler dessus.

Le moins que nous puissions rendre faisable et de partager la base de données *le moins souvent* est d'une grande valeur.

Cela demande un cache. Et bien , pas juste un seul cache. Une application Seam bien désignée va fonctionner avec une stratégie de cache multi couche riche qui va impacter chaque couche de l'application:

- La base de données a , bien sûr, son propre cache. C'est super-important, mais ne peut pas être dimensionné comme un cache dans une application tierce.
- Votre solution ORM (Hibernate ou une autre implémentation JPA) a un cache de second niveau de données de la base de données. C'est une fonctionnalité très puissante mais elle est souvant mal utilisée. Dans une environement en cluster, la conservation des données dans le cache transactionnel consistant au travers du cluster entier et avec la base de données est assez couteux. Cela gagne en plus de sens s'il est partagé entre plusieurs utilisateurs et s'il est mis à jours rarement. Dans les architectures transactionnelles sans état, les gens essaye souvent d'utiliser le cache de second niveau pour les états conversationnel. C'est toujours très mauvais et c'est spécialement faux dans Seam.
- Le contexte de conversation est un cache de l'état conversationnel. Les composant que vous mettez dans le contexte conversationnel peuvent conserver et mettre en cache l'état relatif à l'intéraction de l'utilisateur courrant.
- En particulier, le contexte de persistance géré par Seam (ou un contexte géré par container EJB étendue s'associant avec un bean de session avec état d'étendue conversationnel) agit comme un cache de données qui a été lu pendant la conversation courrante. Ce cache tends à avoir une fréquence très élevé de solicitation! Seam optimise la réplication des contextes de persistance géré par Seam dans un environement en cluster et il n'y a pas de prérequis pour la consistance transactionnelle avec la base de données (un véroullage optimiste est suffisant) donc vous n'avez pas besoin de trop vous inquiéter au sujet de l'implication en terme de performations de ce cache, à moins que ne lisiez des milliers d'objets dans un seul contexte de persistance.

- L'application peu mettre en cache l'état non-transactionnel dans le contexte d'application de Seam. Convervez l'état dans le contexte applicatif est bien sur invisible pour les autres noeuds dans le cluster.
- L'application peut mettre en cache l'état transactionnel en utilisant le composant `cacheProvider` de Seam qui intègre JBossCache, Jboss POJO Cache ou EHCache dans l'environement de Seam. Cet état sera visible pour les autres noeuds si votre cache est capable de s'exécuter dans un mode en cluster.
- Enfin, Seam vous permet de mettre en cache les fragments rendus d'une page JSF. A la différence du cache de second niveau de l'ORM, ce cache n'est pas automatiquement invalidé quand les données changent, donc vous avez besoin d'écrire le code de l'application pour réaliser une invalidation explicite ou de mettre des politiques d'expirations appropriées.

Pour plus d'informations à propos du cache de second niveau, vous allez avoir besoin de vous référer à la document de votre solution ORM, car c'est un sujet extrèmement complexe. Dans cette section, nous allons parler de l'utilisation du chache directemnet via le composant `cacheProvider` , ou comme le cache de fragment de page, via le controle `<s:cache>`.

23.1. Utilisation de JBossCache dans Seam

Le composant livré `cacheProvider` gère une instance de:

JBoss Cache 1.x (disponible pour utilisation dans JBoss 4.2.x ou suppérieur et pour les autres containeurs)

```
org.jboss.cache.TreeCache
```

JBoss Cache 2.x (disponible pour utilisation dans JBoss 5.x et pour les autres containeurs)

```
org.jboss.cache.Cache
```

JBoss POJO Cache 1.x (disponible pour utilisation dans JBoss 4.2.x et pour les autres containeurs)

```
org.jboss.cache.aop.PojoCache
```

EHCache (disponible pour une utilisation dans tout containeur)

```
net.sf.ehcache.CacheManager
```

Vous pouvez de manière sure mettre tout objet Java immuable dans le cache, et il sera stocké dans le cache et répliqué au travers du cluster (en supposant que la réplication est disponible et activée). Si vous voulez conserver les objets mutables dans le cache, vous allez avoir besoin de lire la documentation nécéssaire à l'outil de cache pour comprendre commen lui notifier du changement dans la mise en cache auprès de l'outil du cache.

Pour utiliser `cacheProvider`, vous aller devoir inclure les fichiers jars de l'implémentation de ce cache dans votre projet:

JBoss Cache 1.x

- `jboss-cache.jar` - JBoss Cache 1.4.1

- `jgroups.jar` - JGroups 2.4.1

JBoss Cache 2.x

- `jboss-cache.jar` - JBoss Cache 2.2.0
- `jgroups.jar` - JGroups 2.6.2

JBoss POJO Cache 1.x

- `jboss-cache.jar` - JBoss Cache 1.4.1
- `jgroups.jar` - JGroups 2.4.1
- `jboss-aop.jar` - JBoss AOP 1.5.0

EHCache

- `ehcache.jar` - EHCache 1.2.3

Astuce

Si vous utilisez JBoss Cache dans un containeur autre que le JBoss Application Server, allez voir la page sur JBoss Cache *wiki* [http://wiki.jboss.org/wiki/JBossCache] pour les dépendances.

Pour un déploiement EAR de Seam, nous vous recommandaons que les fichiers jars de la mise en cache et la configuration soient directement mis dans l'EAR.

Vous allez avoir aussi besoinde fournir un fichier de configuration pour JBossCache. Placez `treecache.xml` avec une configuration de la mise en cache approprié dans le classpath (par exemple le fichier jar ejb ou `WEB-INF/classes`). JBossCache a de nombreuses horreurs et des réglages de configurations confus, donc nous n'allons pas en parler ici. Merci de vous référer à la document de JBossCache pour plus d'information.

Vous trouverez un exemple de `treecache.xml` dans `examples/blog/resources/treecache.xml`.

EHCache sera exécuté dans sa configuration par défaut sans fichier de configuration

Pour modifier le fichier de configuration en cours d'utilisation, configurez votre cache dans `components.xml`:

```
<components xmlns="http://jboss.com/products/seam/components"
        xmlns:cache="http://jboss.com/products/seam/cache">
  <cache:jboss-cache-provider configuration="META-INF/cache/treecache.xml" />
</components
>
```

Maintenant, vous pouvez injecter votre cache dans tout composant de Seam:

```
@Name("chatroomUsers")
@Scope(ScopeType.STATELESS)
public class ChatroomUsers
{
   @In CacheProvider cacheProvider;

   @Unwrap
   public Set<String
> getUsers() throws CacheException   {
      Set<String
> userList = (Set<String
>) cacheProvider.get("chatroom", "userList");
      if (userList==null) {
         userList = new HashSet<String
>();
         cacheProvider.put("chatroom", "userList", userList);
      }
      return userList;
   }
}
```

Si vous voulez avoir de multplies configurations de mises en cache dans votre application, utilisez `components.xml` pour configurer vos différents fournisseur de mise en cache:

```
<components xmlns="http://jboss.com/products/seam/components"
         xmlns:cache="http://jboss.com/products/seam/cache">
  <cache:jboss-cache-provider name="myCache" configuration="myown/cache.xml"/>
  <cache:jboss-cache-provider name="myOtherCache" configuration="myother/cache.xml"/>
</components
>
```

23.2. La mise en cache de fragment de page

L'utlisation la plus intéressante de JBossCache est le tag `<s:cache>` , la solution de Seam pour le problème de mettre en cache les fragments de pages.`<s:cache>` utilise `pojoCache` en interne, donc nous avez besoin de suivre les étapes listées ci dessous avant de pouvoir l'utiliser. (Mettez les jars dans le EAR, barboter dans les horribles options de configurations , etc.)

`<s:cache>` est utilisé pour mettre en cache quelques contenues rendues rarement. par exemple, la page bienvenue de votre blog qui affiche les entrées récentes du blog:

```
<s:cache key="recentEntries-#{blog.id}" region="welcomePageFragments">
  <h:dataTable value="#{blog.recentEntries}" var="blogEntry">
    <h:column>
      <h3
>#{blogEntry.title}</h3>
      <div>
        <s:formattedText value="#{blogEntry.body}"/>
      </div>
    </h:column>
  </h:dataTable>
</s:cache
>
```

La `key` permet d'avoir de multiples versions en cache de chaque fragment de page. Dans ce cas, il n'y a qu'une version mise en cache par blog. La `region` détermine le cache ou le noeud local pour toutes les versions qui seront à stocker. Les noeuds différents peuvent avoir des politiques d'expirations différentes. (C'est du boulot que vous configuriez tout en utilisant les horribles options de configurations susmentionnées.)

Bien sur, le gros problème avec `<s:cache>` est qu'il est trop stupide pour connaitre quand les données sousjacentes changent (par exemple, quand les billets d'un bloggeur ont une nouvelle entrée). Donc vous avez besoin de virer les fragment en cache manuellement:

```
public void post() {
   ...
   entityManager.persist(blogEntry);
   cacheProvider.remove("welcomePageFragments", "recentEntries-" + blog.getId() );
}
```

Autre alternative, si ce n'est pas critique que les modifications ne soient pas immédiatement visitble pour l'utilisateur, vous pouvez définir une periode d'expiration courte dans le noeud de JbossCache.

Web Services

Seam integrates with JBossWS to allow standard JEE web services to take full advantage of Seam's contextual framework, including support for conversational web services. This chapter walks through the steps required to allow web services to run within a Seam environment.

24.1. Configuration and Packaging

To allow Seam to intercept web service requests so that the necessary Seam contexts can be created for the request, a special SOAP handler must be configured; `org.jboss.seam.webservice.SOAPRequestHandler` is a `SOAPHandler` implementation that does the work of managing Seam's lifecycle during the scope of a web service request.

A special configuration file, `standard-jaxws-endpoint-config.xml` should be placed into the `META-INF` directory of the `jar` file that contains the web service classes. This file contains the following SOAP handler configuration:

```
<jaxws-config xmlns="urn:jboss:jaxws-config:2.0"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xmlns:javaee="http://java.sun.com/xml/ns/javaee"
              xsi:schemaLocation="urn:jboss:jaxws-config:2.0 jaxws-config_2_0.xsd">
  <endpoint-config>
    <config-name>Seam WebService Endpoint</config-name>
    <pre-handler-chains>
      <javaee:handler-chain>
        <javaee:protocol-bindings>##SOAP11_HTTP</javaee:protocol-bindings>
        <javaee:handler>
          <javaee:handler-name>SOAP Request Handler</javaee:handler-name>
                    <javaee:handler-class>org.jboss.seam.webservice.SOAPRequestHandler</
javaee:handler-class>
        </javaee:handler>
      </javaee:handler-chain>
    </pre-handler-chains>
  </endpoint-config>
</jaxws-config>
```

24.2. Conversational Web Services

So how are conversations propagated between web service requests? Seam uses a SOAP header element present in both the SOAP request and response messages to carry the conversation ID from the consumer to the service, and back again. Here's an example of a web service request that contains a conversation ID:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:seam="http://seambay.example.seam.jboss.org/">
  <soapenv:Header>
            <seam:conversationId    xmlns:seam='http://www.jboss.org/seam/webservice'>2</
seam:conversationId>
  </soapenv:Header>
  <soapenv:Body>
    <seam:confirmAuction/>
  </soapenv:Body>
</soapenv:Envelope>
```

As you can see in the above SOAP message, there is a `conversationId` element within the SOAP header that contains the conversation ID for the request, in this case `2`. Unfortunately, because web services may be consumed by a variety of web service clients written in a variety of languages, it is up to the developer to implement conversation ID propagation between individual web services that are intended to be used within the scope of a single conversation.

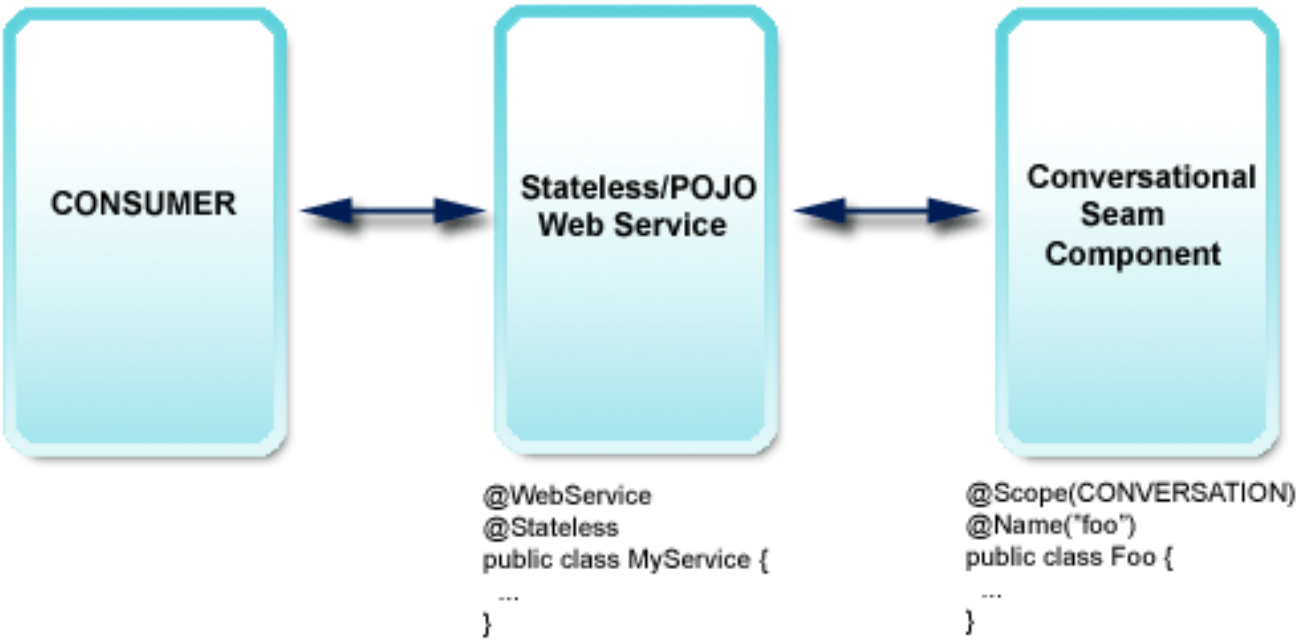
An important thing to note is that the `conversationId` header element must be qualified with a namespace of `http://www.jboss.org/seam/webservice`, otherwise Seam will not be able to read the conversation ID from the request. Here's an example of a response to the above request message:

```
<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'>
  <env:Header>
            <seam:conversationId    xmlns:seam='http://www.jboss.org/seam/webservice'>2</
seam:conversationId>
  </env:Header>
  <env:Body>
    <confirmAuctionResponse xmlns="http://seambay.example.seam.jboss.org/"/>
  </env:Body>
</env:Envelope>
```

As you can see, the response message contains the same `conversationId` element as the request.

24.2.1. A Recommended Strategy

As web services must be implemented as either a stateless session bean or POJO, it is recommended that for conversational web services, the web service acts as a facade to a conversational Seam component.



If the web service is written as a stateless session bean, then it is also possible to make it a Seam component by giving it a `@Name`. Doing this allows Seam's bijection (and other) features to be used in the web service class itself.

24.3. An example web service

Let's walk through an example web service. The code in this section all comes from the seamBay example application in Seam's `/examples` directory, and follows the recommended strategy as described in the previous section. Let's first take a look at the web service class and one of its web service methods:

```
@Stateless
@WebService(name = "AuctionService", serviceName = "AuctionService")
public class AuctionService implements AuctionServiceRemote
{
  @WebMethod
  public boolean login(String username, String password)
  {
    Identity.instance().setUsername(username);
    Identity.instance().setPassword(password);
    Identity.instance().login();
    return Identity.instance().isLoggedIn();
  }

  // snip
}
```

As you can see, our web service is a stateless session bean, and is annotated using the JWS annotations from the `javax.jws` package, as defined by JSR-181. The `@WebService` annotation tells the container that this class implements a web service, and the `@WebMethod` annotation on the `login()` method identifies the method as a web service method. The `name` and `serviceName` attributes in the `@WebService` annotation are optional.

As is required by the specification, each method that is to be exposed as a web service method must also be declared in the remote interface of the web service class (when the web service is a stateless session bean). In the above example, the `AuctionServiceRemote` interface must declare the `login()` method as it is annotated as a `@WebMethod`.

As you can see in the above code, the web service implements a `login()` method that delegates to Seam's built-in `Identity` component. In keeping with our recommended strategy, the web service is written as a simple facade, passing off the real work to a Seam component. This allows for the greatest reuse of business logic between web services and other clients.

Let's look at another example. This web service method begins a new conversation by delegating to the `AuctionAction.createAuction()` method:

```
@WebMethod
public void createAuction(String title, String description, int categoryId)
{
   AuctionAction action = (AuctionAction) Component.getInstance(AuctionAction.class, true);
   action.createAuction();
   action.setDetails(title, description, categoryId);
}
```

And here's the code from `AuctionAction`:

```
@Begin
public void createAuction()
{
   auction = new Auction();
   auction.setAccount(authenticatedAccount);
   auction.setStatus(Auction.STATUS_UNLISTED);
   durationDays = DEFAULT_AUCTION_DURATION;
}
```

From this we can see how web services can participate in long running conversations, by acting as a facade and delegating the real work to a conversational Seam component.

24.4. RESTful HTTP webservices with RESTEasy

Seam integrates the RESTEasy implementation of the JAX-RS specification (JSR 311). You can decide how "deep" the integration into your Seam application is going to be:

- Seamless integration of RESTEasy bootstrap and configuration, automatic detection of resources and providers.
- Serving HTTP/REST requests with the SeamResourceServlet, no external servlet or configuration in web.xml required.
- Writing resources as Seam components, with full Seam lifecycle management and interception (bijection).

24.4.1. RESTEasy configuration and request serving

First, get the RESTEasy libraries and the `jaxrs-api.jar`, deploy them with the other libraries of your application. Also deploy the integration library, `jboss-seam-resteasy.jar`.

In seam-gen based projects, this can be done by appending `jaxrs-api.jar`, `resteasy-jaxrs.jar` and `jboss-seam-resteasy.jar` to the `deployed-jars.list` (war deployment) or `deployed-jars-ear.list` (ear deployment) file. For a JBoss Tools based project, copy the libraries mentioned above to the `EarContent/lib` (ear deployment) or `WebContent/WEB-INF/lib` (war deployment) folder and reload the project in the IDE.

On startup, all classes annotated `@javax.ws.rs.Path` will be discovered automatically and registered as HTTP resources. Seam automatically accepts and serves HTTP requests with its built-in `SeamResourceServlet`. The URI of a resource is build as follows:

- The URI starts with the host and context path of your application, e.g. `http://your.hostname/myapp`.
- Then the pattern mapped in `web.xml` for the `SeamResourceServlet`, e.g `/seam/resource` if you follow the common examples, is appended. Change this setting to expose your RESTful resources under a different base. Note that this is a global change and other Seam resources (e.g. `s:graphicImage` and `s:captcha`) are then also served under that base path.
- The RESTEasy integration for Seam then appends a configurable string to the base path, by default this is `/rest`. Hence, the full base path of your resources would e.g. be `/myapp/seam/resource/rest`. We recommend that you change this string in your application (details below). You could for example add a version number to prepare for a future REST API upgrade of your services (old clients would keep the old URI base): `/myapp/seam/resource/restv1`.
- Finally, the actual resource is available under the defined `@Path`, e.g. a resource mapped with `@Path("/customer")` would be available under `/myapp/seam/resource/rest/customer`.

As an example, the following resource definition would return a plaintext representation for any GET requests using the URI `http://your.hostname/myapp/seam/resource/rest/customer/123`:

```
@Path("/customer")
public class MyCustomerResource {

   @GET
   @Path("/{customerId}")
   @Produces("text/plain")
   public String getCustomer(@PathParam("customerId") int id) {
       return ...;
   }

}
```

No additional configuration is required; you do not have to edit `web.xml` or any other setting if these defaults are acceptable. However, you can configure RESTEasy in your Seam application. First import the `resteasy` namespace into your XML configuration (`components.xml`) file header:

```
<components
  xmlns="http://jboss.com/products/seam/components"
  xmlns:resteasy="http://jboss.com/products/seam/resteasy"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    http://jboss.com/products/seam/resteasy
       http://jboss.com/products/seam/resteasy-2.2.xsd
    http://jboss.com/products/seam/components
       http://jboss.com/products/seam/components-2.2.xsd">
```

You can then change the `/rest` prefix as mentioned earlier:

```
<resteasy:application resource-path-prefix="/restv1"/>
```

The full base path to your resources is now `/myapp/seam/resource/restv1/{resource}` - note that your `@Path` definitions and mappings do NOT change. This is an application-wide switch usually used for versioning of the HTTP interface.

Seam will scan your classpath for any deployed `@javax.ws.rs.Path` resources and any `@javax.ws.rs.ext.Provider` classes. You can disable scanning and configure these classes manually:

```
<resteasy:application
    scan-providers="false"
    scan-resources="false"
    use-builtin-providers="true">

    <resteasy:resource-class-names>
        <value>org.foo.MyCustomerResource</value>
        <value>org.foo.MyOrderResource</value>
        <value>org.foo.MyStatelessEJBImplementation</value>
    </resteasy:resource-class-names>

    <resteasy:provider-class-names>
        <value>org.foo.MyFancyProvider</value>
    </resteasy:provider-class-names>

 </resteasy:application>
```

The `use-built-in-providers` switch enables (default) or disables the RESTEasy built-in providers. We recommend you leave them enabled, as they provide plaintext, JSON, and JAXB marshalling out of the box.

RESTEasy supports plain EJBs (EJBs that are not Seam components) as resources. Instead of configuring the JNDI names in a non-portable fashion in `web.xml` (see RESTEasy documentation), you can simply list the EJB implementation classes, not the business interfaces, in `components.xml` as shown above. Note that you have to annotate the `@Local` interface of the EJB with `@Path`, `@GET`, and so on - not the bean implementation class. This allows you to keep your application deployment-portable with the global Seam `jndi-pattern` switch on `<core:init/>`. Note that plain (non-Seam component) EJB resources will not be found even if scanning of resources is enabled, you always have to list them manually. Again, this whole paragraph is only relevant for EJB resources that are not also Seam components and that do not have an `@Name` annotation.

Finally, you can configure media type and language URI extensions:

```
<resteasy:application>

   <resteasy:media-type-mappings>
     <key>txt</key><value>text/plain</value>
   </resteasy:media-type-mappings>

   <resteasy:language-mappings>
     <key>deutsch</key><value>de-DE</value>
   </resteasy:language-mappings>
```

```
</resteasy:application>
```

This definition would map the URI suffix of `.txt.deutsch` to additional `Accept` and `Accept-Language` header values `text/plain` and `de-DE`.

24.4.2. Resources as Seam components

Any resource and provider instances are managed by RESTEasy by default. That means a resource class will be instantiated by RESTEasy and serve a single request, after which it will be destroyed. This is the default JAX-RS lifecycle. Providers are instantiated once for the whole application and are effectively singletons and supposed to be stateless.

You can write resources as Seam components and benefit from the richer lifecycle management of Seam, and interception for bijection, security, and so on. Simply make your resource class a Seam component:

```
@Name("customerResource")
@Path("/customer")
public class MyCustomerResource {

   @In
   CustomerDAO customerDAO;

   @GET
   @Path("/{customerId}")
   @Produces("text/plain")
   public String getCustomer(@PathParam("customerId") int id) {
       return customerDAO.find(id).getName();
   }

}
```

An instance of `customerResource` is now handled by Seam when a request hits the server. This is a Seam JavaBean component that is `EVENT`-scoped, hence no different than the default JAX-RS lifecycle. You get full Seam injection and interception support, and all other Seam components and contexts are available to you. Currently also supported are `APPLICATION` and `STATELESS` resource Seam components. These three scopes allow you to create an effectively stateless Seam middle-tier HTTP request-processing application.

You can annotate an interface and keep the implementation free from JAX-RS annotations:

```
@Path("/customer")
```

```
public interface MyCustomerResource {

   @GET
   @Path("/{customerId}")
   @Produces("text/plain")
   public String getCustomer(@PathParam("customerId") int id);

}
```

```
@Name("customerResource")
@Scope(ScopeType.STATELESS)
public class MyCustomerResourceBean implements MyCustomerResource {

   @In
   CustomerDAO customerDAO;

   public String getCustomer(int id) {
      return customerDAO.find(id).getName();
   }

}
```

You can use `SESSION`-scoped Seam components. By default, the session will however be shortened to a single request. In other words, when an HTTP request is being processed by the RESTEasy integration code, an HTTP session will be created so that Seam components can utilize that context. When the request has been processed, Seam will look at the session and decide if the session was created only to serve that single request (no session identifier has been provided with the request, or no session existed for the request). If the session has been created only to serve this request, the session will be destroyed after the request!

Assuming that your Seam application only uses event, application, or stateless components, this procedure prevents exhaustion of available HTTP sessions on the server. The RESTEasy integration with Seam assumes by default that sessions are not used, hence anemic sessions would add up as every REST request would start a session that will only be removed when timed out.

If your RESTful Seam application has to preserve session state across REST HTTP requests, disable this behavior in your configuration file:

```
<resteasy:application destroy-session-after-request="false"/>
```

Every REST HTTP request will now create a new session that will only be removed by timeout or explicit invalidation in your code through `Session.instance().invalidate()`. It is your responsibility to pass a valid session identifier along with your HTTP requests, if you want to utilize the session context across requests.

`CONVERSATION`-scoped resource components and mapping of conversations to temporary HTTP resources and paths is planned but currently not supported.

EJB Seam components are supported as REST resources. Always annotate the local business interface, not the EJB implementation class, with JAX-RS annotations. The EJB has to be `STATELESS`.

Sub-resources as defined in the JAX RS specification, section 3.4.1, can also be Seam component instances:

```
@Path("/garage")
@Name("garage")
public class GarageService
{
  ...

  @Path("/vehicles")
  public VehicleService getVehicles() {
    return (VehicleService) Component.getInstance(VehicleService.class);
  }
}
```

Note

RESTEasy components do not support hot redeployment. As a result, the components should never be placed in the `src/hot` folder. The `src/main` folder should be used instead.

Note

Provider classes can currently not be Seam components. Although you can configure an `@Provider` annotated class as a Seam component, it will at runtime be managed by RESTEasy as a singleton with no Seam interception, bijection, etc. The instance will not be a Seam component instance. We plan to support Seam component lifecycle for JAX-RS providers in the future.

24.4.3. Securing resources

You can enable the Seam authentication filter for HTTP Basic and Digest authentication in `components.xml`:

```
<web:authentication-filter url-pattern="/seam/resource/rest/*" auth-type="basic"/>
```

See the Seam security chapter on how to write an authentication routine.

After successful authentication, authorization rules with the common `@Restrict` and `@PermissionCheck` annotations are in effect. You can also access the client `Identity`, work with permission mapping, and so on. All regular Seam security features for authorization are available.

24.4.4. Mapping exceptions to HTTP responses

Section 3.3.4 of the JAX-RS specification defines how checked or unchecked exceptions are handled by the JAX RS implementation. In addition to using an exception mapping provider as defined by JAX-RS, the integration of RESTEasy with Seam allows you to map exceptions to HTTP response codes within Seam's `pages.xml` facility. If you are already using `pages.xml` declarations, this is easier to maintain than potentially many JAX RS exception mapper classes.

Exception handling within Seam requires that the Seam filter is executed for your HTTP request. Ensure that you do filter *all* requests in your `web.xml`, not - as some Seam examples might show - a request URI pattern that doesn't cover your REST request paths. The following example intercepts *all* HTTP requests and enables Seam exception handling:

```
<filter>
   <filter-name>Seam Filter</filter-name>
   <filter-class>org.jboss.seam.servlet.SeamFilter</filter-class>
</filter>

<filter-mapping>
   <filter-name>Seam Filter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

To convert the unchecked `UnsupportedOperationException` thrown by your resource methods to a `501 Not Implemented` HTTP status response, add the following to your `pages.xml` descriptor:

```
<exception class="java.lang.UnsupportedOperationException">
   <http-error error-code="501">
      <message>The requested operation is not supported</message>
   </http-error>
```

```
</exception>
```

Custom or checked exceptions are handled the same:

```
<exception class="my.CustomException" log="false">
  <http-error error-code="503">
    <message>Service not available: #{org.jboss.seam.handledException.message}</message>
  </http-error>
</exception>
```

You do not have to send an HTTP error to the client if an exception occurs. Seam allows you to map the exception as a redirect to a view of your Seam application. As this feature is typically used for human clients (web browsers) and not for REST API remote clients, you should pay extra attention to conflicting exception mappings in `pages.xml`.

Note that the HTTP response still passes through the servlet container, so an additional mapping might apply if you have `<error-page>` mappings in your `web.xml` configuration. The HTTP status code would then be mapped to a rendered HTML error page with status `200 OK`!

24.4.5. Exposing entities via RESTful API

Seam makes it really easy to use a RESTful approach for accessing application data. One of the improvements that Seam introduces is the ability to expose parts of your SQL database for remote access via plain HTTP calls. For this purpose, the Seam/RESTEasy integration module provides two components: `ResourceHome` and `ResourceQuery`, which benefit from the API provided by the Seam Application Framework (*Chapitre 13, Le serveur d'application Seam*). These components allow you to bind domain model entity classes to an HTTP API.

24.4.5.1. ResourceQuery

ResourceQuery exposes entity querying capabilities as a RESTful web service. By default, a simple underlying Query component, which returns a list of instances of a given entity class, is created automatically. Alternatively, the ResourceQuery component can be attached to an existing Query component in more sophisticated cases. The following example demonstrates how easily ResourceQuery can be configured:

```
<resteasy:resource-query
  path="/user"
  name="userResourceQuery"
  entity-class="com.example.User"/>
```

With this single XML element, a ResourceQuery component is set up. The configuration is straightforward:

- The component will return a list of `com.example.User` instances.
- The component will handle HTTP requests on the URI path `/user`.
- The component will by default transform the data into XML or JSON (based on client's preference). The set of supported mime types can be altered by using the `media-types` attribute, for example:

```
<resteasy:resource-query
  path="/user"
  name="userResourceQuery"
  entity-class="com.example.User"
  media-types="application/fastinfoset"/>
```

Alternatively, if you do not like configuring components using XML, you can set up the component by extension:

```
@Name("userResourceQuery")
@Path("user")
public class UserResourceQuery extends ResourceQuery<User>
{
}
```

Queries are read-only operations, the resource only responds to GET requests. Furthermore, ResourceQuery allows clients of a web service to manipulate the resultset of a query using the following path parameters:

Parameter name	Example	Description
start	/user?start=20	Returns a subset of a database query result starting with the 20th entry.
show	/user?show=10	Returns a subset of the database query result limited to 10 entries.

For example, you can send an HTTP GET request to `/user?start=30&show=10` to get a list of entries representing 10 rows starting with row 30.

Note

RESTEasy uses JAXB to marshall entities. Thus, in order to be able to transfer them over the wire, you need to annotate entity classes with `@XMLRootElement`. Consult the JAXB and RESTEasy documentation for more information.

24.4.5.2. ResourceHome

Just as ResourceQuery makes Query's API available for remote access, so does ResourceHome for the Home component. The following table describes how the two APIs (HTTP and Home) are bound together.

Tableau 24.1.

HTTP method	Path	Function	ResourceHome method
GET	{path}/{id}	Read	getResource()
POST	{path}	Create	postResource()
PUT	{path}/{id}	Update	putResource()
DELETE	{path}/{id}	Delete	deleteResource()

- You can GET, PUT, and DELETE a particular user instance by sending HTTP requests to /user/{userId}
- Sending a POST request to `/user` creates a new user entity instance and persists it. Usually, you leave it up to the persistence layer to provide the entity instance with an identifier value and thus an URI. Therefore, the URI is sent back to the client in the `Location` header of the HTTP response.

The configuration of ResourceHome is very similar to ResourceQuery except that you need to explicitly specify the underlying Home component and the Java type of the entity identifier property.

```
<resteasy:resource-home
  path="/user"
  name="userResourceHome"
  entity-home="#{userHome}"
  entity-id-class="java.lang.Integer"/>
```

Again, you can write a subclass of ResourceHome instead of XML:

```
@Name("userResourceHome")
```

```
@Path("user")
public class UserResourceHome extends ResourceHome<User, Integer>
{

   @In
   private EntityHome<User
> userHome;

   @Override
   public Home<?, User
> getEntityHome()
   {
      return userHome;
   }
}
```

For more examples of ResourceHome and ResourceQuery components, take a look at the *Seam Tasks* example application, which demonstrates how Seam/RESTEasy integration can be used together with a jQuery web client. In addition, you can find more code example in the *Restbay* example, which is used mainly for testing purposes.

24.4.6. Testing resources and providers

Seam includes a unit testing utility class that helps you create unit tests for a RESTful architecture. Extend the `SeamTest` class as usual and use the `ResourceRequestEnvironment.ResourceRequest` to emulate HTTP requests/response cycles:

```
import org.jboss.seam.mock.ResourceRequestEnvironment;
import org.jboss.seam.mock.EnhancedMockHttpServletRequest;
import org.jboss.seam.mock.EnhancedMockHttpServletResponse;
import static org.jboss.seam.mock.ResourceRequestEnvironment.ResourceRequest;
import static org.jboss.seam.mock.ResourceRequestEnvironment.Method;

public class MyTest extends SeamTest {

   ResourceRequestEnvironment sharedEnvironment;

   @BeforeClass
   public void prepareSharedEnvironment() throws Exception {
      sharedEnvironment = new ResourceRequestEnvironment(this) {
         @Override
         public Map<String, Object> getDefaultHeaders() {
            return new HashMap<String, Object>() {{
               put("Accept", "text/plain");
```

```
            }};
          }
        };
  }

  @Test
  public void test() throws Exception
  {
    //Not shared: new ResourceRequest(new ResourceRequestEnvironment(this), Method.GET,
 "/my/relative/uri)

    new ResourceRequest(sharedEnvironment, Method.GET, "/my/relative/uri)
    {
      @Override
      protected void prepareRequest(EnhancedMockHttpServletRequest request)
      {
        request.addQueryParameter("foo", "123");
        request.addHeader("Accept-Language", "en_US, de");
      }

      @Override
      protected void onResponse(EnhancedMockHttpServletResponse response)
      {
        assert response.getStatus() == 200;
        assert response.getContentAsString().equals("foobar");
      }

    }.run();
  }
}
```

This test only executes local calls, it does not communicate with the `SeamResourceServlet` through TCP. The mock request is passed through the Seam servlet and filters and the response is then available for test assertions. Overriding the `getDefaultHeaders()` method in a shared instance of `ResourceRequestEnvironment` allows you to set request headers for every test method in the test class.

Note that a `ResourceRequest` has to be executed in a `@Test` method or in a `@BeforeMethod` callback. You can not execute it in any other callback, such as `@BeforeClass`.

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.

25.1. Configuration

To use remoting, the Seam Resource servlet must first be configured in your `web.xml` file:

```
<servlet>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <servlet-class>org.jboss.seam.servlet.SeamResourceServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <url-pattern>/seam/resource/*</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/resource/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/resource/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/resource/remoting/interface.js?customerAction&accountAction"></script>
```

Alternatively, you may use the `s:remote` tag to import the required Javascript. Separate each component or class name you wish to import with a comma:

```
 <s:remote include="customerAction,accountAction"/>
```

25.2. 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.

25.2.1. 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.

Note

If you are performing a persistence operation in the method marked `@WebRemote` you will also need to add a `@Transactional` annotation to the method. Otherwise, your method would execute outside of a transaction without this extra hint.That's because unlike a JSF request, Seam does not wrap the remoting request in a transaction automatically.

Now for our web page - create a new page and import the `helloAction` component:

```
<s:remote include="helloAction"/>
```

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.

25.2.2. 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.

25.2.2.1. 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";
```

25.2.2.2. 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);
```

25.2.2.3. 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");
```

25.2.3. 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.

25.2.3.1. 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");
```

25.2.3.2. 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.

25.3. Client Interfaces

In the configuration section above, the interface, or "stub" for our component is imported into our page either via `seam/resource/remoting/interface.js`: or using the `s:remote` tag:

```
<script type="text/javascript"
      src="seam/resource/remoting/interface.js?customerAction"></script>
```

```
<s:remote include="customerAction"/>
```

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.

25.4. 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.

25.4.1. 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.

25.4.2. Remote calls within the current conversation scope

In some circumstances it may be required to make a remote call within the scope of the current view's conversation. To do this, you must explicitly set the conversation ID to that of the view before making the remote call. This small snippet of JavaScript will set the conversation ID that is used for remoting calls to the current view's conversation ID:

```
Seam.Remoting.getContext().setConversationId( #{conversation.id} );
```

25.5. 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`.

25.6. Working with Data types

25.6.1. 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.

25.6.1.1. String

Simply use Javascript String objects when setting String parameter values.

25.6.1.2. 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.

25.6.1.3. Boolean

Booleans are represented client side by Javascript Boolean values, and server side by a Java boolean.

25.6.2. 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:

```
<s:remote include="myAction,myWidget"/>
```

This will then allow a `myWidget` object to be created with `Seam.Component.newInstance("myWidget")`, which can then be passed to `myAction.doSomethingWithObject()`.

25.6.3. 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.

25.6.4. 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.

25.6.5. Collections

25.6.5.1. 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.

25.6.5.2. 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).

25.7. 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 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)`.

25.8. Handling Exceptions

When invoking a remote component method, it is possible to specify an exception handler which will process the response in the event of an exception during component invocation. To specify an exception handler function, include a reference to it after the callback parameter in your JavaScript:

```
var callback = function(result) { alert(result); };
var exceptionHandler = function(ex) { alert("An exception occurred: " + ex.getMessage()); };
Seam.Component.getInstance("helloAction").sayHello(name, callback, exceptionHandler);
```

If you do not have a callback handler defined, you must specify `null` in its place:

```
var exceptionHandler = function(ex) { alert("An exception occurred: " + ex.getMessage()); };
Seam.Component.getInstance("helloAction").sayHello(name, null, exceptionHandler);
```

The exception object that is passed to the exception handler exposes one method, `getMessage()` that returns the exception message which is produced by the exception thrown by the `@WebRemote` method.

25.9. 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.

25.9.1. 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...";
```

25.9.2. 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() {};
```

25.9.3. 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
```

```
};
```

25.10. 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
}
```

25.10.1. 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();
```

25.10.2. 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();
```

25.10.3. 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();
```

25.10.4. 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();
```

25.11. Transactional Requests

By default there is no active transaction during a remoting request, so if you wish to perform database updates during a remoting request, you need to annotate the `@WebRemote` method with `@Transactional`, like so:

```
@WebRemote @Transactional(TransactionPropagationType.REQUIRED)
public void updateOrder(Order order) {
  entityManager.merge(order);
}
```

25.12. 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.

25.12.1. 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 poll-timeout="5" poll-interval="1"/>
```

25.12.2. 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 `value` field (or call its `getValue()` method).

25.12.3. Unsubscribing from a Topic

To unsubscribe from a topic, call `Seam.Remoting.unsubscribe()` and pass in the topic name:

```
Seam.Remoting.unsubscribe("topicName");
```

25.12.4. 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:

```
<remoting:remoting poll-timeout="5" poll-interval="1"/>
```

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;
```

Seam et le Google Web Toolkit

Pour ceux qui p^réfère utiliser le Google Web Toolkit (GWT) pour développer des applications AJAX dynamique, Seam fourni une couche d'intégration qui permet aux widgets GWT d'interagir directement avecc les composants de Seam.

Pour utiliser GWT, nous allons postuler que vous êtes déjà familier avec les outils GWT - plus d'informations peut ête trouvée sur *http://code.google.com/webtoolkit/*. Ce chapitre n'essaye pas d'expliquer comment GWT fonctionne et comment l'utiliser.

26.1. La configuration

Il n'y a pas de configuration spécialé nécéssaire pour utiliser GWT dans une application Seam, cependant le servlet de ressource de Seam doit être installé. Voir *Chapitre 30, Configuring Seam and packaging Seam applications* pour les détails.

26.2. La préparation de votre composant

La première étape dans la préparation du composant dfe Seam est d'être appelé via GWT, cela créer deux interfaces de service synchrone et assynchrone pour les méthode que vous souhaitez appeler. Ces deux interfaces devraient étendre l'interface GWT `com.google.gwt.user.client.rpc.RemoteService`:

```
public interface MyService extends RemoteService {
   public String askIt(String question);
}
```

L'interface asynchrone devrait être identique, à la différence qu'il doit contenir un paramètre additionnel `AsyncCallback` pour chaque méthode qu'il déclare:

```
public interface MyServiceAsync extends RemoteService {
  public void askIt(String question, AsyncCallback callback);
}
```

L'interface assynchrone, dans cet exemple `MyServiceAsync`, sera implémenté par GWT et ne devrait pas être implémenté directement.

L'étape suivante, est de créer un composant de Seam qui implémente l'interface synchrone:

```
@Name("org.jboss.seam.example.remoting.gwt.client.MyService")
public class ServiceImpl implements MyService {
```

```
  @WebRemote
  public String askIt(String question) {

    if (!validate(question)) {
      throw new IllegalStateException("Hey, this shouldn't happen, I checked on the client, " +
      "but its always good to double check.");
    }
    return "42. Its the real question that you seek now.";
  }

  public boolean validate(String q) {
    ValidationUtility util = new ValidationUtility();
    return util.isValid(q);
  }
}
```

Le nom du composant seam *doit* correspondre au nom pleinement qualifié de l'interface client GWT (comme montré), ou ne servlet de ressource de seam ne sera pas capable de le trouver quand un client fait un appel GWT. Les méthodes qui sont accéssible via GWT doivent aussi être annotées avec l'annotation `@WebRemote`.

26.3. Interception deu widget GWT vers un composant de Seam

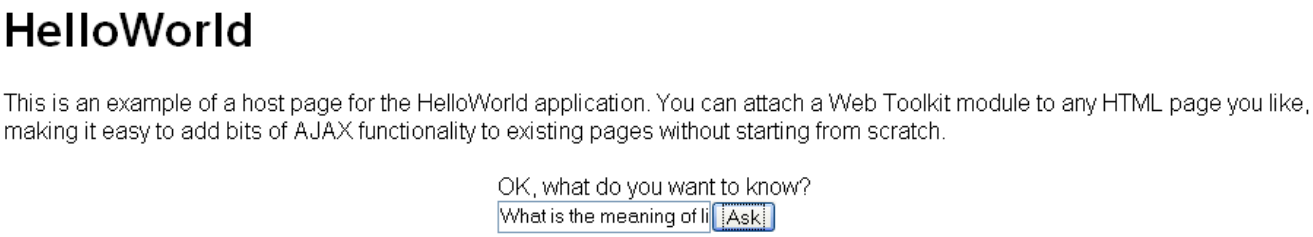
La prochaine étape est d'écrire une méthode qui retourne un itnerface assynchrone vers le composant. Cette méthode est localisée dans la classe widget, et sera utilisé avec le widget pour obtenir une référence vers un squelette de client assynchrone:

```
private MyServiceAsync getService() {
  String endpointURL = GWT.getModuleBaseURL() + "seam/resource/gwt";

  MyServiceAsync svc = (MyServiceAsync) GWT.create(MyService.class);
  ((ServiceDefTarget) svc).setServiceEntryPoint(endpointURL);
  return svc;
}
```

La dernière étape est d'écrire le code du widget qui invoque la méthode sur le squelette du client. L'exemple suivant créer un interface utilisateur simple avec un label, une zone de saisie et un bouton:

```
public class AskQuestionWidget extends Composite {
  private AbsolutePanel panel = new AbsolutePanel();

  public AskQuestionWidget() {
    Label lbl = new Label("OK, what do you want to know?");
    panel.add(lbl);
    final TextBox box = new TextBox();
    box.setText("What is the meaning of life?");
    panel.add(box);
    Button ok = new Button("Ask");
    ok.addClickListener(new ClickListener() {
      public void onClick(Widget w) {
        ValidationUtility valid = new ValidationUtility();
        if (!valid.isValid(box.getText())) {
          Window.alert("A question has to end with a '?'");
        } else {
          askServer(box.getText());
        }
      }
    });
    panel.add(ok);

    initWidget(panel);
  }

  private void askServer(String text) {
    getService().askIt(text, new AsyncCallback() {
      public void onFailure(Throwable t) {
        Window.alert(t.getMessage());
      }

      public void onSuccess(Object data) {
        Window.alert((String) data);
      }
    });
  }

  ...
```

Quand on clique, le bouton appelle la méthode `askServer()` en passant le contenu de la zone de saisie (dans cet exemple, la validation est aussi réalisé pour s'assurer que la zone de saiie contient une question valide). La méthode `askServer()` obtient une référencer vers un squelette de client assynchrone (retournée par la méthode `getService()`) et invoque la méthode `askIt()`. Le résultat (ou le message d'erreursi l'appel échoue) est affiché dans une fenètre d'alerte.



Le code complet pour cet exemple peut être trouvé dans la distribution de Seam dans le dossier `examples/remoting/gwt`.

26.4. Les cibles Ant GWT

Pour le déploiement des applications GWT, il ya une étape de compilation vers Javascript (avec compactage et cryptage du code). Il y a un utilitaire de ant qui peut être utilisé au lieu de la ligne de commande ou de l'utilitaire GUI que GWT fourni. Pour l'utiliser, vous allez avoir besoin d'avoit le jar de tâche ant dans votre classpath, tout comme le GWT téléchargé (avec ce que vous avez besoin pour votre mode hébergement).

Ensuite, dans votre fichier ant, mettez (presque en haut de votre fichier ant):

```
<taskdef uri="antlib:de.samaflost.gwttasks"
  resource="de/samaflost/gwttasks/antlib.xml"
  classpath="./lib/gwttasks.jar"/>

  <property file="build.properties"/>
```

Créez un fichier `build.properties`, qui va avoir ce contenu:

```
gwt.home=/gwt_home_dir
```

Ceci devrait bien sûr pointer vers le dossier où GWT doit être installé. Ensuite, utilisez le pour créer une cible:

```
<!-- the following are are handy utilities for doing GWT development.
  To use GWT, you will of course need to download GWT seperately -->
  <target name="gwt-compile">
    <!-- in this case, we are "re homing" the gwt generated stuff, so in this case
    we can only have one GWT module - we are doing this deliberately to keep the URL short -->
    <delete>
      <fileset dir="view"/>
    </delete>
```

```
    <gwt:compile outDir="build/gwt"
      gwtHome="${gwt.home}"
      classBase="${gwt.module.name}"
      sourceclasspath="src"/>
    <copy todir="view">
      <fileset dir="build/gwt/${gwt.module.name}"/>
    </copy>
  </target
>
```

La cible quand appelée va compiler l'application GWT et la copier dans le dossier spécifiée (ce qui devrait être dans le coin `webapp` de votre war - n'oubliez pas GWT génère des artefactes HTML et javascript). Vous n'éditez jamais le code résultant que `gwt-compile` génère - vous devez toujours éditer le source dans le dossier GWT.

N'oubliez pas que GWT vient avec un mode navigable hébergé - vous devriez l'utiliser pendant que vous développez avec GWT. Si vous ne le faite pas, et compilez à chaque fois, vous n'allez pas avoir le meilleurs de ce kit de développement (dans les fait, si vous ne voulez ou ne pouvez utiliser le mode navigable hébergé, je vous déconseille FORTEMENT d'utiliser GWT - est ce bien clair!).

Spring Framework integration

The Spring integration (part of the Seam IoC module) allows easy migration of Spring-based projects to Seam and allows Spring applications to take advantage of key Seam features like conversations and Seam's more sophisticated persistence context management.

Note! The Spring integration code is included in the jboss-seam-ioc library. This dependency is required for all seam-spring integration techniques covered in this chapter.

Seam's support for Spring provides the ability to:

- inject Seam component instances into Spring beans
- inject Spring beans into Seam components
- turn Spring beans into Seam components
- allow Spring beans to live in any Seam context
- start a spring WebApplicationContext with a Seam component
- Support for Spring PlatformTransactionManagement
- provides a Seam managed replacement for Spring's `OpenEntityManagerInViewFilter` and `OpenSessionInViewFilter`
- Support for Spring `TaskExecutors` to back `@Asynchronous` calls

27.1. Injecting Seam components into Spring beans

Injecting Seam component instances into Spring beans is accomplished using the `<seam:instance/>` namespace handler. To enable the Seam namespace handler, the Seam namespace must be added to the Spring beans definition file:

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:seam="http://jboss.com/products/seam/spring-seam"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
                  http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                  http://jboss.com/products/seam/spring-seam
                  http://jboss.com/products/seam/spring-seam-2.2.xsd">
```

Now any Seam component may be injected into any Spring bean:

```
<bean id="someSpringBean" class="SomeSpringBeanClass" scope="prototype">
```

```
  <property name="someProperty">
    <seam:instance name="someComponent"/>
  </property>
</bean>
```

An EL expression may be used instead of a component name:

```
<bean id="someSpringBean" class="SomeSpringBeanClass" scope="prototype">
  <property name="someProperty">
    <seam:instance name="#{someExpression}"/>
  </property>
</bean>
```

Seam component instances may even be made available for injection into Spring beans by a Spring bean id.

```
<seam:instance name="someComponent" id="someSeamComponentInstance"/>

<bean id="someSpringBean" class="SomeSpringBeanClass" scope="prototype">
  <property name="someProperty" ref="someSeamComponentInstance">
</bean>
```

Now for the caveat!

Seam was designed from the ground up to support a stateful component model with multiple contexts. Spring was not. Unlike Seam bijection, Spring injection does not occur at method invocation time. Instead, injection happens only when the Spring bean is instantiated. So the instance available when the bean is instantiated will be the same instance that the bean uses for the entire life of the bean. For example, if a Seam `CONVERSATION`-scoped component instance is directly injected into a singleton Spring bean, that singleton will hold a reference to the same instance long after the conversation is over! We call this problem *scope impedance*. Seam bijection ensures that scope impedance is maintained naturally as an invocation flows through the system. In Spring, we need to inject a proxy of the Seam component, and resolve the reference when the proxy is invoked.

The `<seam:instance/>` tag lets us automatically proxy the Seam component.

```
<seam:instance id="seamManagedEM" name="someManagedEMComponent" proxy="true"/>

<bean id="someSpringBean" class="SomeSpringBeanClass">
  <property name="entityManager" ref="seamManagedEM">
```

```
</bean>
```

This example shows one way to use a Seam-managed persistence context from a Spring bean. (For a more robust way to use Seam-managed persistence contexts as a replacement for the Spring `OpenEntityManagerInView` filter see section on *Using a Seam Managed Persistence Context in Spring*)

27.2. Injecting Spring beans into Seam components

It is even easier to inject Spring beans into Seam component instances. Actually, there are two possible approaches:

- inject a Spring bean using an EL expression
- make the Spring bean a Seam component

We'll discuss the second option in the next section. The easiest approach is to access the Spring beans via EL.

The Spring `DelegatingVariableResolver` is an integration point Spring provides for integrating Spring with JSF. This `VariableResolver` makes all Spring beans available in EL by their bean id. You'll need to add the `DelegatingVariableResolver` to `faces-config.xml`:

```
<application>
   <variable-resolver>
      org.springframework.web.jsf.DelegatingVariableResolver
   </variable-resolver>
</application>
```

Then you can inject Spring beans using `@In`:

```
@In("#{bookingService}")
private BookingService bookingService;
```

The use of Spring beans in EL is not limited to injection. Spring beans may be used anywhere that EL expressions are used in Seam: process and pageflow definitions, working memory assertions, etc...

27.3. Making a Spring bean into a Seam component

The `<seam:component/>` namespace handler can be used to make any Spring bean a Seam component. Just place the `<seam:component/>` tag within the declaration of the bean that you wish to be a Seam component:

```
<bean id="someSpringBean" class="SomeSpringBeanClass" scope="prototype">
   <seam:component/>
</bean>
```

By default, `<seam:component/>` will create a `STATELESS` Seam component with class and name provided in the bean definition. Occasionally, such as when a `FactoryBean` is used, the class of the Spring bean may not be the class appearing in the bean definition. In such cases the `class` should be explicitly specified. A Seam component name may be explicitly specified in cases where there is potential for a naming conflict.

The `scope` attribute of `<seam:component/>` may be used if you wish the Spring bean to be managed in a particular Seam scope. The Spring bean must be scoped to `prototype` if the Seam scope specified is anything other than `STATELESS`. Pre-existing Spring beans usually have a fundamentally stateless character, so this attribute is not usually needed.

27.4. Seam-scoped Spring beans

The Seam integration package also lets you use Seam's contexts as Spring 2.0 style custom scopes. This lets you declare any Spring bean in any of Seam's contexts. However, note once again that Spring's component model was never architected to support statefulness, so please use this feature with great care. In particular, clustering of session or conversation scoped Spring beans is deeply problematic, and care must be taken when injecting a bean or component from a wider scope into a bean of a narrower scope.

By specifying `<seam:configure-scopes/>` once in a Spring bean factory configuration, all of the Seam scopes will be available to Spring beans as custom scopes. To associate a Spring bean with a particular Seam scope, specify the Seam scope in the `scope` attribute of the bean definition.

```
<!-- Only needs to be specified once per bean factory-->
<seam:configure-scopes/>

...

<bean id="someSpringBean" class="SomeSpringBeanClass"
 scope="seam.CONVERSATION"/>
```

The prefix of the scope name may be changed by specifying the `prefix` attribute in the `configure-scopes` definition. (The default prefix is `seam.`)

By default an instance of a Spring Component registered in this way is not automatically created when referenced using `@In`. To have an instance auto-created you must either specify `@In(create=true)` at the injection point to identify a specific bean to be auto created or you can

use the `default-auto-create` attribute of `configure-scopes` to make all spring beans who use a seam scope auto created.

Seam-scoped Spring beans defined this way can be injected into other Spring beans without the use of `<seam:instance/>`. However, care must be taken to ensure scope impedance is maintained. The normal approach used in Spring is to specify `<aop:scoped-proxy/>` in the bean definition. However, Seam-scoped Spring beans are *not* compatible with `<aop:scoped-proxy/>`. So if you need to inject a Seam-scoped Spring bean into a singleton, `<seam:instance/>` must be used:

```
<bean id="someSpringBean" class="SomeSpringBeanClass"
 scope="seam.CONVERSATION"/>

...

<bean id="someSingleton">
   <property name="someSeamScopedSpringBean">
      <seam:instance name="someSpringBean" proxy="true"/>
   </property>
</bean>
```

27.5. Using Spring PlatformTransactionManagement

Spring provides an extensible transaction management abstraction with support for many transaction APIs (JPA, Hibernate, JDO, and JTA) Spring also provides tight integrations with many application server TransactionManagers such as Websphere and Weblogic. Spring transaction management exposes support for many advanced features such as nested transactions and supports full Java EE transaction propagation rules like REQUIRES_NEW and NOT_SUPPORTED. For more information see the spring documentation *here* [http://static.springframework.org/spring/docs/2.0.x/reference/transaction.html].

To configure Seam to use Spring transactions enable the SpringTransaction component like so:

```
<spring:spring-transaction platform-transaction-manager="#{transactionManager}"/>
```

The `spring:spring-transaction` component will utilize Springs transaction synchronization capabilities for synchronization callbacks.

27.6. Using a Seam Managed Persistence Context in Spring

One of the most powerful features of Seam is its conversation scope and the ability to have an EntityManager open for the life of a conversation. This eliminates many of the problems associated with the detachment and re-attachment of entities as well as mitigates occurrences of the dreaded `LazyInitializationException`. Spring does not provide a way to manage an persistence context beyond the scope of a single web request (`OpenEntityManagerInViewFilter`). So, it would be nice if Spring developers could have access to a Seam managed persistence context using all of the same tools Spring provides for integration with JPA(e.g. `PersistenceAnnotationBeanPostProcessor`, `JpaTemplate`, etc.)

Seam provides a way for Spring to access a Seam managed persistence context with Spring's provided JPA tools bringing conversation scoped persistence context capabilities to Spring applications.

This integration work provides the following functionality:

- transparent access to a Seam managed persistence context using Spring provided tools
- access to Seam conversation scoped persistence contexts in a non web request (e.g. asynchronous quartz job)
- allows for using Seam managed persistence contexts with Spring managed transactions (will need to flush the persistence context manually)

Spring's persistence context propagation model allows only one open EntityManager per EntityManagerFactory so the Seam integration works by wrapping an EntityManagerFactory around a Seam managed persistence context.

```
<bean                                                    id="seamEntityManagerFactory"
 class="org.jboss.seam.ioc.spring.SeamManagedEntityManagerFactoryBean">
        <property name="persistenceContextName" value="entityManager"/>
</bean>
```

Where 'persistenceContextName' is the name of the Seam managed persistence context component. By default this EntityManagerFactory has a unitName equal to the Seam component name or in this case 'entityManager'. If you wish to provide a different unitName you can do so by providing a persistenceUnitName like so:

```
<bean                                                    id="seamEntityManagerFactory"
 class="org.jboss.seam.ioc.spring.SeamManagedEntityManagerFactoryBean">
        <property name="persistenceContextName" value="entityManager"/>
```

```
    <property name="persistenceUnitName" value="bookingDatabase:extended"/>
</bean>
```

This EntityManagerFactory can then be used in any Spring provided tools. For example, using Spring's `PersistenceAnnotationBeanPostProcessor` is the exact same as before.

```
<bean
 class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"/>
```

If you define your real EntityManagerFactory in Spring but wish to use a Seam managed persistence context you can tell the `PersistenceAnnotationBeanPostProcessor` which persistenctUnitName you wish to use by default by specifying the `defaultPersistenceUnitName` property.

The `applicationContext.xml` might look like:

```
<bean                                                    id="entityManagerFactory"
 class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
    <property name="persistenceUnitName" value="bookingDatabase"/>
</bean>
<bean                                                    id="seamEntityManagerFactory"
 class="org.jboss.seam.ioc.spring.SeamManagedEntityManagerFactoryBean">
      <property name="persistenceContextName" value="entityManager"/>
    <property name="persistenceUnitName" value="bookingDatabase:extended"/>
</bean>
<bean
 class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor">
    <property name="defaultPersistenceUnitName" value="bookingDatabase:extended"/>
</bean>
```

The `component.xml` might look like:

```
<persistence:managed-persistence-context name="entityManager"
    auto-create="true" entity-manager-factory="#{entityManagerFactory}"/>
```

`JpaTemplate` and `JpaDaoSupport` are configured the same way for a Seam managed persistence context as they would be fore a Seam managed persistence context.

```
<bean id="bookingService" class="org.jboss.seam.example.spring.BookingService">
    <property name="entityManagerFactory" ref="seamEntityManagerFactory"/>
```

```
</bean>
```

27.7. Using a Seam Managed Hibernate Session in Spring

The Seam Spring integration also provides support for complete access to a Seam managed Hibernate session using spring's tools. This integration is very similar to the *JPA integration*.

Like Spring's JPA integration spring's propagation model allows only one open EntityManager per EntityManagerFactory per transaction??? to be available to spring tools. So, the Seam Session integration works by wrapping a proxy SessionFactory around a Seam managed Hibernate session context.

```
<bean                                                    id="seamSessionFactory"
 class="org.jboss.seam.ioc.spring.SeamManagedSessionFactoryBean">
      <property name="sessionName" value="hibernateSession"/>
</bean>
```

Where 'sessionName' is the name of the `persistence:managed-hibernate-session` component. This SessionFactory can then be used in any Spring provided tools. The integration also provides support for calls to `SessionFactory.getCurrentInstance()` as long as you call getCurrentInstance() on the `SeamManagedSessionFactory`.

27.8. Spring Application Context as a Seam Component

Although it is possible to use the Spring `ContextLoaderListener` to start your application's Spring ApplicationContext there are a couple of limitations.

- the Spring ApplicationContext must be started *after* the `SeamListener`
- it can be tricky starting a Spring ApplicationContext for use in Seam unit and integration tests

To overcome these two limitations the Spring integration includes a Seam component that will start a Spring ApplicationContext. To use this Seam component place the `<spring:context-loader/>` definition in the `components.xml`. Specify your Spring context file location in the `config-locations` attribute. If more than one config file is needed you can place them in the nested `<spring:config-locations/>` element following standard `components.xml` multi value practices.

```
<components xmlns="http://jboss.com/products/seam/components"
        xmlns:spring="http://jboss.com/products/seam/spring"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
        xsi:schemaLocation="http://jboss.com/products/seam/components
                    http://jboss.com/products/seam/components-2.2.xsd
                    http://jboss.com/products/seam/spring
                    http://jboss.com/products/seam/spring-2.2.xsd">

    <spring:context-loader config-locations="/WEB-INF/applicationContext.xml"/>

</components>
```

27.9. Using a Spring TaskExecutor for @Asynchronous

Spring provides an abstraction for executing code asynchronously called a `TaskExecutor`. The Spring Seam integration allows for the use of a Spring `TaskExecutor` for executing immediate `@Asynchronous` method calls. To enable this functionality install the `SpringTaskExecutorDispatchor` and provide a spring bean defined taskExecutor like so:

```
<spring:task-executor-dispatcher task-executor="#{springThreadPoolTaskExecutor}"/>
```

Because a Spring `TaskExecutor` does not support scheduling of an asynchronous event a fallback Seam `Dispatcher` can be provided to handle scheduled asynchronous event like so:

```
<!-- Install a ThreadPoolDispatcher to handle scheduled asynchronous event -->
<core:thread-pool-dispatcher name="threadPoolDispatcher"/>

<!-- Install the SpringDispatcher as default -->
<spring:task-executor-dispatcher task-executor="#{springThreadPoolTaskExecutor}"
 schedule-dispatcher="#{threadPoolDispatcher}"/>
```

L'intégration Guice

Google Guice est une bibliothèque qui fornie une injection de dépendance légère au travers d'une résolution de type en mode sûr. L'intégration de Guice (la partie du module IoC de Seam) permet l'utilisation de tous les composants de Seam annoté avec l'annotation `@Guice`. En plus de la bijection classique que Seam réalise (qui devient optionnelle), Seam délègue aussi pour savoir si les injecteurs de Guice satisfons les dependances du composant. Guice peut êtrre utile pour inclure des parties non-Seam de grande applications validées en accord avec Seam.

Note

L'intégration de Guice est livré dans la bibliothèque jboss-seam-ioc. Cette dépendance est nécéssaire pour toutes les techniques d'intégration couverte dans ce chapitre. Vous allez aussi avoir besoin du fichier JAR de Guice dans le classpath.

28.1. La création d'un composant hybride Seam-Guice

Le but est de créer un composant hybride Seam-Guice. La règle pour faire cela est vraiment très simple. Si vous voulez utiliser l'injection de Guice dans votre composant Seam, annoté le avec l'annotation de `@Guice` (après l'imporation du type `org.jboss.seam.ioc.guice.Guice`).

```
@Name("myGuicyComponent")
@Guice public class MyGuicyComponent
{
  @Inject MyObject myObject;
  @Inject @Special MyObject mySpecialObject;
  ...
}
```

L'injection de Guice intervient sur chaque appel de méthode tout comme avec la bijection. Guice injecte en se bassant sur le type et la correspondance. Pour satisfaire les dépencandes dans notre exemple précédent, vous devriez avoir lié ces implémentations suivante dans le module Guice, avec `@Special` comme annotation que vous défininissez dans votre application.

```
public class MyGuicyModule implements Module
{
  public void configure(Binder binder)
  {
    binder.bind(MyObject.class)
      .toInstance(new MyObject("regular"));
```

```
    binder.bind(MyObject.class).annotatedWith(Special.class)
      .toInstance(new MyObject("special"));
  }
}
```

Génial, mais avec l'injection de Guice qui va être utilisé pour injecter les dépendances? Et bien, vous avez besoin de réaliser quelques configurations en premier lieu.

28.2. La configuration d'une injection

Vous pouvez dire à Seam quel injecteur de Guice à utiliser quand il intercepte la propriété injectée du composant d'initialisation de Guice dans le descripteur de composant de Seam (components.xml):

```
<components xmlns="http://jboss.com/products/seam/components"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:guice="http://jboss.com/products/seam/guice"
  xsi:schemaLocation="
    http://jboss.com/products/seam/guice
    http://jboss.com/products/seam/guice-2.2.xsd
    http://jboss.com/products/seam/components
    http://jboss.com/products/seam/components-2.2.xsd">

  <guice:init injector="#{myGuiceInjector}"/>

</components
>
```

`myGuiceInjector` doit être résolue vers un composant de Seam qui implémente l'interface `Injector` de Guice.

Avoir à créer un injecteur est un code de type copier/coller. Quand vous voulez réellement être capable de simplement intercepter Seam vers vos modules Guice. Heureusement, il ya un composant livré de Seam qui implémente l'interface `Injector` pour faire exactement tout cela. Vous pouvez le configurer dans le descripteur de composant de Seam avec cette strophe additionnelle.

```
<guice:injector name="myGuiceInjector">
  <guice:modules
>
    <value
```

```
>com.example.guice.GuiceModule1</value
>
    <value
>com.example.guice.GuiceModule2</value
>
  </guice:modules
>
</guice:injector
>
```

Bien sur vous pouvez aussi utiliser un injecteur qui est déjà utilisé dans d'autres, particulièrement des parties non-Seam de votre application. Ceci est une des principales motivation de la création de cette intégration. Quand l'injecteur est définie avec une expression EL, vous pouvez obtenir par ce biais ce que vous voulez. Par exemple, vous pouvez utiliser le patron composant de fabrique de Seam pour fournir votre injecteur.

```
@Name("myGuiceInjectorFactory")
public InjectorFactory
{
  @Factory(name = "myGuiceInjector", scope = APPLICATION, create = true)
  public Injector getInjector()
  {
    // Your code that returns injector
  }
}
```

28.3. L'utilisation de multiples injecteurs

Par défaut, un injecteur configuré dans le descripteur de composant de Seam est utilisé. Si vous avez réllement besoin d'utiliser plusieurs injecteurs (A ma connaissance vous devriez plutôt utiliser plusieurs modules), vous pouvez spécifier différents injecteurs pour chaque composant de Seam dans l'annotation `@Guice`.

```
@Name("myGuicyComponent")
@Guice("myGuiceInjector")
public class MyGuicyComponent
{
  @Inject MyObject myObject;
  ...
}
```

C'est tout ce qu'il y a à faire! Consultez l'exemple guice dans la distribution de Seam pour voir l'intégration de Guice de Seam en action!

Hibernate Search

29.1. Introduction

Les moteurs de recherches plein texte comme Apache Lucene™ sont une technologie très puissante qui apporte la recherche plain texte et l'efficacité des requêtes dans les applications. Hibernate Search qui utilise Apache Lucene soujacent indexe votre modèle du domaine avec quelques annotations additionnelles, prends garde à la synchronisation base de données / indexes et retourne des objets opérationnels qui correspondent aux requêtes de recherche plain texte. Gardez à l'esprit qu'il peut avoir des erreurs qui apparaisssent avec un modèle de domaine objet par dessus un index de textes (conserver l'index à jours, erreur entre la structure de l'index et le modèle du domaine, et erreurs dans les requêtes). Mais les avantages de la vitesse et de l'efficacité va au delà de ces limitations.

Hibernate Search a été conçu pour s'intégrer facilement et naturellement que possible avec JPA et Hibernate. Comme une extension naturelle, JBoss Seam fourni une intégration avec Hibernate Search.

Please refer to the *Hibernate Search documentation* [] for information specific to the Hibernate Search project.

29.2. La configuration

Hibernate Search est configuré aussi bien dans le fichier `META-INF/persistence.xml` que dans le fichier `hibernate.cfg.xml`.

La configuration d'Hibernate Search configuration est par défaut judicieuse pour la plus part des paramètre de configuration. Ici n configuration d'une unité de persistance minimale pour démarrer.

```
<persistence-unit name="sample">
  <jta-data-source
>java:/DefaultDS</jta-data-source>
  <properties>
    [...]
    <!-- use a file system based index -->
    <property name="hibernate.search.default.directory_provider"
      value="org.hibernate.search.store.FSDirectoryProvider"/>
    <!-- directory where the indexes will be stored -->
    <property name="hibernate.search.default.indexBase"
      value="/Users/prod/apps/dvdstore/dvdindexes"/>
  </properties>
</persistence-unit
>
```

Si vous plannifier de cibler Hibernate Annotations ou EntityManager 3.2.x (embarqué dans JBoss AS 4.2.x et suppérieur), vous allez avoir besoin de configurer les écouteurs d'évènement appropriés.

```
<persistence-unit name="sample">
  <jta-data-source
>java:/DefaultDS</jta-data-source>
  <properties>
    [...]
    <!-- use a file system based index -->
    <property name="hibernate.search.default.directory_provider"
          value="org.hibernate.search.store.FSDirectoryProvider"/>
    <!-- directory where the indexes will be stored -->
    <property name="hibernate.search.default.indexBase"
          value="/Users/prod/apps/dvdstore/dvdindexes"/>

    <property name="hibernate.ejb.event.post-insert"
          value="org.hibernate.search.event.FullTextIndexEventListener"/>
    <property name="hibernate.ejb.event.post-update"
          value="org.hibernate.search.event.FullTextIndexEventListener"/>
    <property name="hibernate.ejb.event.post-delete"
          value="org.hibernate.search.event.FullTextIndexEventListener"/>

  </properties>
</persistence-unit
>
```

Note

Il n'est plus nécéssaire d'enregistrer les écouteurs d'évènements si Hibernate Annotations ou EntityManager 3.3.x sont utilisés. Quand on utilise Hibernate Search 3.1.x quelques écouteurs d'évènements sont nécéssaire, mais ils sont automatiquement enregistrés par Hibernate Annotations; référeez vous au guide d'Hibernate Search pour sa configuration sans EntityManager et les Annotations.

En plus du fichier de configuration, les jars suivants doivent être déployés:

- hibernate-search.jar
- hibernate-commons-annotations.jar
- lucene-core.jar

`hibernate-commons-annotations.jar` is not needed in JBossAS6. Some Hibernate Search extensions require additional dependencies, a commonly used is `hibernate-search-analyzers.jar`. For details, see the *Hibernate Search documentation* [http://www.hibernate.org/subprojects/search/docs] for details.

Note

Si vous déployez ceux là dans un EAR, n'oubliez pas de mettre à jours `application.xml`

29.3. Utilisation

Hibernate Search uses annotations to map entities to a Lucene index, check the *reference documentation* [http://www.hibernate.org/subprojects/search/docs] for more informations.

Hibernate Search est pleinement intégré avec les API et la sémentique JPA / Hibernate. Le basculement des requêtes HQL ou Criteria nécéssite juste quelques lignes de codes. L'API principale à intéragir avec est l'API `FullTextSession` (sousclasse de `Session` d'Hibernate).

Quand Hibernate Search est présent, JBoss Seam injecte un `FullTextSession`.

```
@Stateful
@Name("search")
public class FullTextSearchAction implements FullTextSearch, Serializable {

  @In FullTextSession session;

  public void search(String searchString) {
    org.apache.lucene.search.Query luceneQuery = getLuceneQuery();
    org.hibernate.Query query session.createFullTextQuery(luceneQuery, Product.class);
    searchResults = query
        .setMaxResults(pageSize + 1)
        .setFirstResult(pageSize * currentPage)
        .list();
  }
  [...]
}
```

Note

`FullTextSession` étend `org.hibernate.Session` ainsi il peut être utilisé avec une Hibernate Session habituelle

Si l'API de Persistence de Java est utilisée, une intégration en douceur est proposée.

```
@Stateful
@Name("search")
public class FullTextSearchAction implements FullTextSearch, Serializable {

  @In FullTextEntityManager em;

  public void search(String searchString) {
    org.apache.lucene.search.Query luceneQuery = getLuceneQuery();
    javax.persistence.Query query = em.createFullTextQuery(luceneQuery, Product.class);
    searchResults = query
        .setMaxResults(pageSize + 1)
        .setFirstResult(pageSize * currentPage)
        .getResultList();
  }
  [...]
}
```

Quand Hibernate Search est présent, un `FulltextEntityManager` est injecté. `FullTextEntityManager` étends `EntityManager` avec des méthodes de recherche spécifique, de la même façon que `FullTextSession` étends `Session`.

Quand une Session EJB3.0 ou une injection de Message Driven Bean est utilisé (par exemple via l'annotation @PersistenceContext), il ets possible de remplacer l'interface de `EntityManager` par l'interface de `FullTextEntityManager` dans la partie déclaration. Cependant, l'implémentation injectée sera une implémentation de `FullTextEntityManager`: la conversion est ensuite possible.

```
@Stateful
@Name("search")
public class FullTextSearchAction implements FullTextSearch, Serializable {

  @PersistenceContext EntityManager em;

  public void search(String searchString) {
    org.apache.lucene.search.Query luceneQuery = getLuceneQuery();
    FullTextEntityManager ftEm = (FullTextEntityManager) em;
    javax.persistence.Query query = ftEm.createFullTextQuery(luceneQuery, Product.class);
    searchResults = query
        .setMaxResults(pageSize + 1)
        .setFirstResult(pageSize * currentPage)
        .getResultList();
```

```
  }
  [...]
}
```

Attention

Pour les personnes abituées à Hibernate Search sans Seam, notez que l'utilisation de `Search.getFullTextSession` n'est pas nécéssaire.

Voyez le DVDStore dans les exemples de blog de la distribution de JBoss Seam pour une utilisation concrète de Hibernate Search.

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 use seam-gen to start your application or you can copy and paste from the example applications!

30.1. Basic Seam configuration

First, let's look at the basic configuration that is needed whenever we use Seam with JSF.

30.1.1. Integrating Seam with JSF and your servlet container

Of course, you need a faces servlet!

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

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

(You can adjust the URL pattern to suit your taste.)

In addition, 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.

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>
```

There is a minor gray area in the JSF specification regarding the mutability of view state values. Since Seam uses the JSF view state to back its PAGE scope this can become an issue in some cases. If you're using server side state saving with the JSF-RI and you want a PAGE scoped bean to keep its exact value for a given view of a page you will need to specify the following context-param. Otherwise if a user uses the "back" button a PAGE scoped component will have the latest value if it has changed not the value of the "back" page. (see *Spec Issue* [https://javaserverfaces-spec-public.dev.java.net/issues/show_bug.cgi?id=295]). This setting is not enabled by default because of the performance hit of serializing the JSF view with every request.

```
<context-param>
   <param-name>com.sun.faces.serializeServerState</param-name>
   <param-value>true</param-value>
</context-param>
```

30.1.2. 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>
```

If you are using facelets in JBoss AS, you'll find that Facelets logging is broken (the log messages don't make it to the server log). Seam provides a bridge to fix this, to use it copy `lib/interop/jboss-seam-jul.jar` to `$JBOSS_HOME/server/default/deploy/jboss-web.deployer/jsf-libs/` and include the `jboss-seam-ui.jar` in the `WEB-INF/lib` of your application. The Facelets logging catagories are itemized in the *Facelets Developer Documentation* [https://facelets.dev.java.net/nonav/docs/dev/docbook.html#config-logging].

30.1.3. Seam Resource Servlet

The Seam Resource Servlet provides resources used by Seam Remoting, captchas (see the security chapter) and some JSF UI controls. Configuring the Seam Resource Servlet requires the following entry in `web.xml`:

```
<servlet>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <servlet-class>org.jboss.seam.servlet.SeamResourceServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <url-pattern>/seam/resource/*</url-pattern>
</servlet-mapping>
```

30.1.4. Seam servlet filters

Seam doesn't need any servlet filters for basic operation. However, there are several features which depend upon the use of filters. To make things easier, Seam lets you add and configure servlet filters just like you would configure other built-in Seam components. To take advantage of this feature, we must first install a master filter in `web.xml`:

```
<filter>
   <filter-name>Seam Filter</filter-name>
   <filter-class>org.jboss.seam.servlet.SeamFilter</filter-class>
</filter>

<filter-mapping>
   <filter-name>Seam Filter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

The Seam master filter *must* be the first filter specified in `web.xml`. This ensures it is run first.

The Seam filters share a number of common attributes, you can set these in `components.xml` in addition to any parameters discussed below:

- `url-pattern` — Used to specify which requests are filtered, the default is all requests. `url-pattern` is a Tomcat style pattern which allows a wildcard suffix.
- `regex-url-pattern` — Used to specify which requests are filtered, the default is all requests. `regex-url-pattern` is a true regular expression match for request path.
- `disabled` — Used to disable a built in filter.

Note that the patterns are matched against the URI path of the request (see `HttpServletRequest.getURIPath()`) and that the name of the servlet context is removed before matching.

Adding the master filter enables the following built-in filters.

30.1.4.1. Exception handling

This filter provides the exception mapping functionality in `pages.xml` (almost all applications will need this). It also takes care of rolling back uncommitted transactions when uncaught exceptions occur. (According to the Java EE specification, the web container should do this automatically, but we've found that this behavior cannot be relied upon in all application servers. And it is certainly not required of plain servlet engines like Tomcat.)

By default, the exception handling filter will process all requests, however this behavior may be adjusted by adding a `<web:exception-filter>` entry to `components.xml`, as shown in this example:

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

   <web:exception-filter url-pattern="*.seam"/>

</components>
```

30.1.4.2. Conversation propagation with redirects

This filter allows Seam to propagate the conversation context across browser redirects. It intercepts any browser redirects and adds a request parameter that specifies the Seam conversation identifier.

The redirect filter will process all requests by default, but this behavior can also be adjusted in `components.xml`:

```
<web:redirect-filter url-pattern="*.seam"/>
```

30.1.4.3. URL rewriting

This filter allows Seam to apply URL rewriting for views based on configuration in the `pages.xml` file. This filter is not activate by default, but can be activated by adding the configuration to `components.xml`:

```
<web:rewrite-filter view-mapping="*.seam"/>
```

The `view-mapping` parameter must match the servlet mapping defined for the Faces Servlet in the `web.xml` file. If ommitted, the rewrite filter assumes the pattern `*.seam`.

30.1.4.4. Multipart form submissions

This feature is necessary when using the Seam file upload JSF control. It detects multipart form requests and processes them according to the multipart/form-data specification (RFC-2388). To override the default settings, add the following entry to `components.xml`:

```
<web:multipart-filter create-temp-files="true"
                      max-request-size="1000000"
                      url-pattern="*.seam"/>
```

- `create-temp-files` — If set to `true`, uploaded files are written to a temporary file (instead of held in memory). This may be an important consideration if large file uploads are expected. The default setting is `false`.
- `max-request-size` — If the size of a file upload request (determined by reading the `Content-Length` header in the request) exceeds this value, the request will be aborted. The default setting is 0 (no size limit).

30.1.4.5. Character encoding

Sets the character encoding of submitted form data.

This filter is not installed by default and requires an entry in `components.xml` to enable it:

```
<web:character-encoding-filter encoding="UTF-16"
                               override-client="true"
                               url-pattern="*.seam"/>
```

- `encoding` — The encoding to use.
- `override-client` — If this is set to `true`, the request encoding will be set to whatever is specified by `encoding` no matter whether the request already specifies an encoding or not. If set to `false`, the request encoding will only be set if the request doesn't already specify an encoding. The default setting is `false`.

30.1.4.6. RichFaces

If RichFaces is used in your project, Seam will install the RichFaces Ajax filter for you, making sure to install it before all other built-in filters. You don't need to install the RichFaces Ajax filter in `web.xml` yourself.

The RichFaces Ajax filter is only installed if the RichFaces jars are present in your project.

To override the default settings, add the following entry to `components.xml`. The options are the same as those specified in the RichFaces Developer Guide:

```
<web:ajax4jsf-filter force-parser="true"
                     enable-cache="true"
                     log4j-init-file="custom-log4j.xml"
                     url-pattern="*.seam"/>
```

- `force-parser` — forces all JSF pages to be validated by Richfaces's XML syntax checker. If `false`, only AJAX responses are validated and converted to well-formed XML. Setting `force-parser` to `false` improves performance, but can provide visual artifacts on AJAX updates.
- `enable-cache` — enables caching of framework-generated resources (e.g. javascript, CSS, images, etc). When developing custom javascript or CSS, setting to true prevents the browser from caching the resource.
- `log4j-init-file` — is used to setup per-application logging. A path, relative to web application context, to the log4j.xml configuration file should be provided.

30.1.4.7. Identity Logging

This filter adds the authenticated user name to the log4j mapped diagnostic context so that it can be included in formatted log output if desired, by adding %X{username} to the pattern.

By default, the logging filter will process all requests, however this behavior may be adjusted by adding a `<web:logging-filter>` entry to `components.xml`, as shown in this example:

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

```
    <web:logging-filter url-pattern="*.seam"/>
</components>
```

30.1.4.8. Context management for custom 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 other servlet that needs access to Seam components.

This filter allows custom servlets to interact with the Seam contexts. It sets up the Seam contexts at the beginning of each request, and tears them down at the end of the request. You should make sure that this filter is *never* applied to the JSF `FacesServlet`. Seam uses the phase listener for context management in a JSF request.

This filter is not installed by default and requires an entry in `components.xml` to enable it:

```
<web:context-filter url-pattern="/media/*"/>
```

The context filter 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`.

30.1.4.9. Enabling HTTP cache-control headers

Seam does *not* automatically add `cache-control` HTTP headers to any resources served by the Seam resource servlet, or directly from your view directory by the servlet container. This means that your images, Javascript and CSS files, and resource representations from Seam resource servlet such as Seam Remoting Javascript interfaces are usually not cached by the browser. This is convenient in development but should be changed in production when optimizing the application.

You can configure a Seam filter to enable automatic addition of `cache-control` headers depending on the requested URI in `components.xml`:

```
<web:cache-control-filter name="commonTypesCacheControlFilter"
                          regex-url-pattern=".*(\.gif|\.png|\.jpg|\.jpeg|\.css|\.js)"
                          value="max-age=86400"/> <!-- 1 day -->

<web:cache-control-filter name="anotherCacheControlFilter"
                          url-pattern="/my/cachable/resources/*"
```

```
                value="max-age=432000"/> <!-- 5 days -->
```

You do not have to name the filters unless you have more than one filter enabled.

30.1.4.10. Adding custom filters

Seam can install your filters for you, allowing you to specify *where* in the chain your filter is placed (the servlet specification doesn't provide a well defined order if you specify your filters in a `web.xml`). Just add the `@Filter` annotation to your Seam component (which must implement `javax.servlet.Filter`):

```
@Startup
@Scope(APPLICATION)
@Name("org.jboss.seam.web.multipartFilter")
@BypassInterceptors
@Filter(within="org.jboss.seam.web.ajax4jsfFilter")
public class MultipartFilter extends AbstractFilter {
```

Adding the `@Startup` annotation means thar the component is available during Seam startup; bijection isn't available here (`@BypassInterceptors`); and the filter should be further down the chain than the RichFaces filter (`@Filter(within="org.jboss.seam.web.ajax4jsfFilter")`).

30.1.5. Integrating Seam with your EJB container

In a Seam application, EJB components have a certain duality, as they are managed by both the EJB container and Seam. Actually, it's more that Seam resolves EJB component references, manages the lifetime of stateful session bean components, and also participates in each method call via interceptors. Let's start with the configuration of the Seam interceptor chain.

We need to apply the `SeamInterceptor` to our Seam EJB components. This interceptor delegates to a set of built-in server-side interceptors that handle such concerns as bijection, conversation demarcation, and business process signals. The simplest way to do this across an entire application is to add the following interceptor configuration 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>
```

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 (and may depend on your own naming conventions as well). We usually specify this option in `components.xml`.

For JBoss AS, the following pattern is correct:

```
<core:init jndi-name="earName/#{ejbName}/local" />
```

In this case, `earName` is the name of the EAR in which the bean is deployed, Seam replaces `#{ejbName}` with the name of the EJB, and the final segment represents the type of interface (local or remote).

Outside the context of an EAR (when using the JBoss Embeddable EJB3 container), the first segment is dropped since there is no EAR, leaving us with the following pattern:

```
<core:init jndi-name="#{ejbName}/local" />
```

How these JNDI names are resolved and somehow locate an EJB component might appear a bit like black magic at this point, so let's dig into the details. First, let's talk about how the EJB components get into JNDI.

The folks at JBoss don't care much for XML, if you can't tell. So when they designed JBoss AS, they decided that EJB components would get assigned a global JNDI name automatically, using the pattern just described (i.e., EAR name/EJB name/interface type). The EJB name is the first non-empty value from the following list:

- The value of the `<ejb-name>` element in ejb-jar.xml
- The value of the `name` attribute in the @Stateless or @Stateful annotation
- The simple name of the bean class

Let's look at an example. Assume that you have the following EJB bean and interface defined.

```
package com.example.myapp;
```

```
import javax.ejb.Local;

@Local
public interface Authenticator
{
   boolean authenticate();
}

package com.example.myapp;

import javax.ejb.Stateless;

@Stateless
@Name("authenticator")
public class AuthenticatorBean implements Authenticator
{
   public boolean authenticate() { ... }
}
```

Assuming your EJB bean class is deployed in an EAR named myapp, the global JNDI name myapp/AuthenticatorBean/local will be assigned to it on JBoss AS. As you learned, you can reference this EJB component as a Seam component with the name `authenticator` and Seam will take care of finding it in JNDI according to the JNDI pattern (or `@JndiName` annotation).

So what about the rest of the application servers? Well, according to the Java EE spec, which most vendors try to adhere to religiously, you have to declare an EJB reference for your EJB in order for it to be assigned a JNDI name. That requires some XML. It also means that it is up to you to establish a JNDI naming convention so that you can leverage the Seam JNDI pattern. You might find the JBoss convention a good one to follow.

There are two places you have to define the EJB reference when using Seam on non-JBoss application servers. If you are going to be looking up the Seam EJB component through JSF (in a JSF view or as a JSF action listener) or a Seam JavaBean component, then you must declare the EJB reference in web.xml. Here is the EJB reference for the example component just shown:

```
<ejb-local-ref>
   <ejb-ref-name>myapp/AuthenticatorBean/local</ejb-ref-name>
   <ejb-ref-type>Session</ejb-ref-type>
   <local>org.example.vehicles.action.Authenticator</local>
</ejb-local-ref>
```

This reference will cover most uses of the component in a Seam application. However, if you want to be able to inject a Seam EJB component into another Seam EJB component using `@In`, you

need to define this EJB reference in another location. This time, it must be defined in ejb-jar.xml, and it's a bit tricker.

Within the context of an EJB method call, you have to deal with a somewhat sheltered JNDI context. When Seam attempts to find another Seam EJB component to satisfy an injection point defined using `@In`, whether or not it finds it depends on whether an EJB reference exists in JNDI. Strictly speaking, you cannot simply resolve JNDI names as you please. You have to define the references explicitly. Fortunately, JBoss recognized how aggrevating this would be for the developer and all versions of JBoss automatically register EJBs so they are always available in JNDI, both to the web container and the EJB container. So if you are using JBoss, you can skip the next few paragraphs. However, if you are deploying to GlassFish, pay close attention.

For application servers that stubbornly adhere to the EJB specification, EJB references must always be defined explicitly. But unlike with the web context, where a single resource reference covers all uses of the EJB from the web environment, you cannot declare EJB references globally in the EJB container. Instead, you have to specify the JNDI resources for a given EJB component one-by-one.

Let's assume that we have an EJB named RegisterAction (the name is resolved using the three steps mentioned previously). That EJB has the following Seam injection:

```
@In(create = true)
Authenticator authenticator;
```

In order for this injection to work, the link must be established in the ejb-jar.xml file as follows:

```
<ejb-jar>
   <enterprise-beans>
      <session>
         <ejb-name>RegisterAction</ejb-name>
         <ejb-local-ref>
            <ejb-ref-name>myapp/AuthenticatorAction/local</ejb-ref-name>
            <ejb-ref-type>Session</ejb-ref-type>
            <local>com.example.myapp.Authenticator</local>
         </ejb-local-ref>
      </session>
   </enterprise-beans>

   ...

</ejb-jar>
```

Notice that the contents of the `<ejb-local-ref>` are identical to what we defined in web.xml. What we are doing is bringing the reference into the EJB context where it can be used by the RegisterAction bean. You will need to add one of these references for any injection of a Seam EJB compoenent into another Seam EJB component using `@In`. (You can see an example of this setup in the jee5/booking example).

But what about `@EJB`? It's true that you can inject one EJB into another using `@EJB`. However, by doing so, you are injecting the actual EJB reference rather than the Seam EJB component instance. In this case, some Seam features will work, while others won't. That's because Seam's interceptor is invoked on any method call to an EJB component. But that only invokes Seam's server-side interceptor chain. What you lose is Seam's state management and Seam's client-side interceptor chain. Client-side interceptors handle concerns such as security and concurrency. Also, when injecting a SFSB, there is no guarantee that you will get the SFSB bound to the active session or conversation, whatever the case may be. Thus, you definitely want to inject the Seam EJB component using `@In`.

That covers how JNDI names are defined and used. The lesson is that with some application servers, such as GlassFish, you are going to have to specify JNDI names for all EJB components explicitly, and sometimes twice! And even if you are following the same naming convention as JBoss AS, the JNDI pattern in Seam may need to be altered. For instance, the global JNDI names are automatically prefixed with java:comp/env on GlassFish, so you need to define the JNDI pattern as follows:

```
<core:init jndi-name="java:comp/env/earName/#{ejbName}/local" />
```

Finally, let's talk about transactions. In an EJB3 environment, we recommend the use of a special built-in component for transaction management, that is fully aware of container transactions, and can correctly process transaction success events registered with the `Events` component. If you don't add this line to your `components.xml` file, Seam won't know when container-managed transactions end:

```
<transaction:ejb-transaction/>
```

30.1.6. Don't forget!

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.

In a web archive (WAR) file, you must place a `seam.properties` file in the `WEB-INF/classes` directory if you have any Seam components included here.

That's why all the Seam examples have an empty `seam.properties` file. You can't just delete this file and expect everything to still work!

You might think this is silly and what kind of idiot framework designers would make an empty file affect the behavior of their software?? Well, this is a workaround for a limitation of the JVM — if we didn't use this mechanism, our next best option would be to force you to list every component explicitly in `components.xml`, just like some other competing frameworks do! I think you'll like our way better.

30.2. Using Alternate JPA Providers

Seam comes packaged and configured with Hibernate as the default JPA provider. If you require using a different JPA provider you must tell `seam` about it.

This is a workaround

Configuration of the JPA provider will be easier in the future and will not require configuration changes, unless you are adding a custom persistence provider implementation.

Telling seam about a different JPA provider can be be done in one of two ways:

Update your application's `components.xml` so that the generic `PersistenceProvider` takes precedence over the hibernate version. Simply add the following to the file:

```
<component name="org.jboss.seam.persistence.persistenceProvider"
        class="org.jboss.seam.persistence.PersistenceProvider"
        scope="stateless">
</component>
```

If you want to take advantage of your JPA provider's non-standard features you will need to write you own implementation of the `PersistenceProvider`. Use `HibernatePersistenceProvider` as a starting point (don't forget to give back to the community :). Then you will need to tell `seam` to use it as before.

```
<component name="org.jboss.seam.persistence.persistenceProvider"
        class="org.your.package.YourPersistenceProvider">
</component>
```

All that is left is updating the `persistence.xml` file with the correct provider class, and what ever properties your provider needs. Don't forget to package your new provider's jar files in the application if they are needed.

30.3. 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!

30.3.1. 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
  lib/
    jboss-el.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 should declare `jboss-seam.jar` as an ejb module in `META-INF/application.xml`; `jboss-el.jar` should be placed in the EAR's lib directory (putting it in the EAR classpath.

If you want to use jBPM or Drools, you must include the needed jars in the EAR's lib directory.

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 PDF or email tag libraries, you need to put `jboss-seam-pdf.jar` or `jboss-seam-mail.jar` in `WEB-INF/lib`.

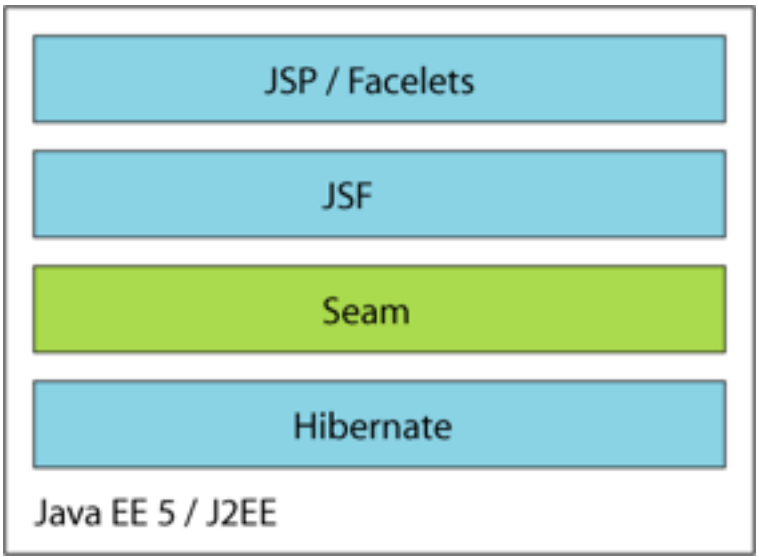
If you want to use the Seam debug page (only works for applications using facelets), you must include `jboss-seam-debug.jar` in the `WEB-INF/lib` directory of the WAR.

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.

30.4. Configuring Seam in J2EE

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 or JPA 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` or declaratively using Seam's `@Transactional` annotation. But most applications will just use Seam managed transactions when using Hibernate with JavaBeans.

The Seam distribution includes a version of the booking example application that uses Hibernate3 and JavaBeans instead of EJB3, and another version that uses JPA and JavaBeans. These example applications are ready to deploy into any J2EE application server.

30.4.1. Boostrapping Hibernate in Seam

Seam will bootstrap a Hibernate `SessionFactory` from your `hibernate.cfg.xml` file if you install a built-in component:

```
<persistence:hibernate-session-factory name="hibernateSessionFactory"/>
```

You will also need to configure a *managed session* if you want a Seam managed Hibernate `Session` to be available via injection.

```
<persistence:managed-hibernate-session name="hibernateSession"
                        session-factory="#{hibernateSessionFactory}"/>
```

30.4.2. Boostrapping JPA in Seam

Seam will bootstrap a JPA `EntityManagerFactory` from your `persistence.xml` file if you install this built-in component:

```
<persistence:entity-manager-factory name="entityManagerFactory"/>
```

You will also need to configure a *managed persistence context* if you want a Seam managed JPA `EntityManager` to be available via injection.

```
<persistence:managed-persistence-context name="entityManager"
                    entity-manager-factory="#{entityManagerFactory}"/>
```

30.4.3. 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
         jboss-el.jar
         jsf-facelets.jar
         hibernate3.jar
         hibernate-annotations.jar
         hibernate-validator.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-EE environment like Tomcat or TestNG, we need to do a little bit more work.

30.5. Configuring Seam in Java SE, without JBoss Embedded

It is possible to use Seam completely outside of an EE environment. In this case, you need to tell Seam how to manage transactions, since there will be no JTA available. If you're using JPA, you can tell Seam to use JPA resource-local transactions, ie. `EntityTransaction`, like so:

```
<transaction:entity-transaction entity-manager="#{entityManager}"/>
```

If you're using Hibernate, you can tell Seam to use the Hibernate transaction API like this:

```
<transaction:hibernate-transaction session="#{session}"/>
```

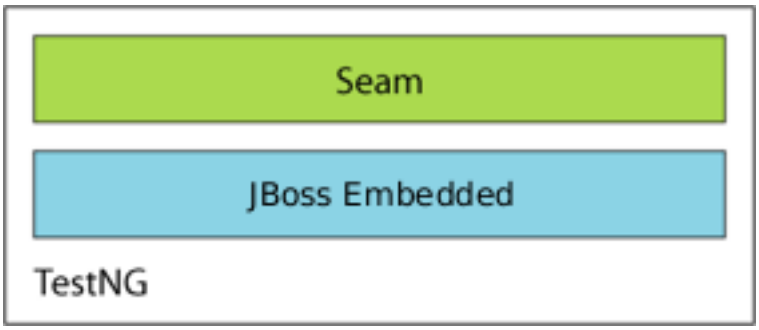
Of course, you'll also need to define a datasource.

A better alternative is to use JBoss Embedded to get access to the EE APIs.

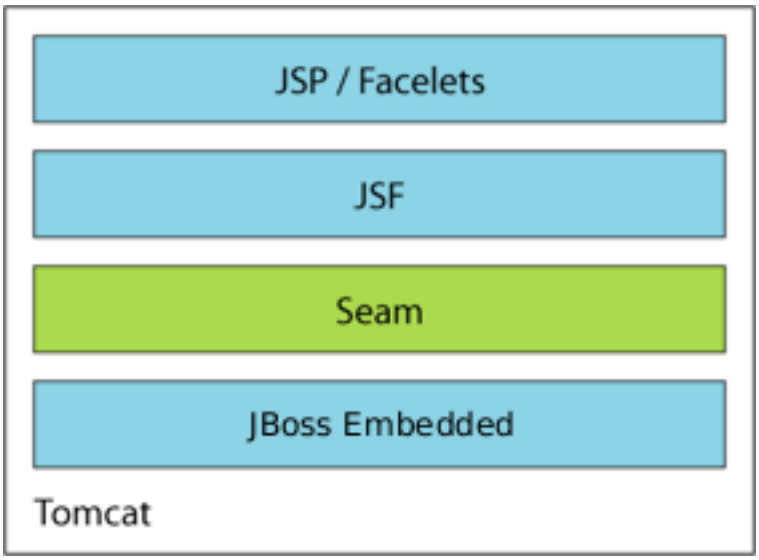
30.6. Configuring Seam in Java SE, with JBoss Embedded

JBoss Embedded 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 JBoss Embedded via `SeamTest`.



The booking example application may even be deployed to Tomcat.



30.6.1. Installing Embedded JBoss

Embedded JBoss must by installed into Tomcat for Seam applications to run correctly on it. Embedded JBoss runs with JDK 5 or JDK 6 (see *Section 42.1, « Les dépendances du JDK »* for details on using JDK 6). Embedded JBoss can be downloaded *here* [http://sourceforge.net/projects/jboss/files/Embedded%20JBoss/Embedded%20JBoss%20Beta%203/]. The process for installing Embedded JBoss into Tomcat 6 is quite simple. First, you should copy the Embedded JBoss JARs and configuration files into Tomcat.

- Copy all files and directories under the Embedded JBoss `bootstrap` and `lib` directories, except for the `jndi.properties` file, into the Tomcat `lib` directory.
- Remove the `annotations-api.jar` file from the Tomcat `lib` directory.

Next, two configuration files need to be updated to add Embedded JBoss-specific functionality.

- Add the Embedded JBoss listener `EmbeddedJBossBootstrapListener` to `conf/server.xml`. It must appear after all other listeners in the file:

```
<Server port="8005" shutdown="SHUTDOWN">

  <!-- Comment these entries out to disable JMX MBeans support used for the
       administration web application -->
  <Listener className="org.apache.catalina.core.AprLifecycleListener" />
  <Listener className="org.apache.catalina.mbeans.ServerLifecycleListener" />
  <Listener className="org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" />
  <Listener className="org.apache.catalina.storeconfig.StoreConfigLifecycleListener" />
```

```
  <!-- Add this listener -->
```

```
<Listener className="org.jboss.embedded.tomcat.EmbeddedJBossBootstrapListener" />
```

- WAR file scanning should be enabled by adding the `WebinfScanner` listener to `conf/context.xml`:

```
<Context>
  <!-- Default set of monitored resources -->
  <WatchedResource>WEB-INF/web.xml</WatchedResource>

  <!-- Uncomment this to disable session persistence across Tomcat restarts -->
  <!--
  <Manager pathname="" />
  -->
```

```
  <!-- Add this listener -->
  <Listener className="org.jboss.embedded.tomcat.WebinfScanner" />
```

```
</Context>
```

- If you are using Sun JDK 6, you need to set the Java option `sun.lang.ClassLoader.allowArraySyntax` to `true` in the JAVA_OPTS environment variable used by the Catalina startup script (catalina.bat on Windows or catalina.sh on Unix).

  Open the script appropriate for your operating system in a text editor. Add a new line immediately below the comments at the top of the file where you will define the JAVA_OPTS environment variable. On Windows, use the following syntax:

```
set JAVA_OPTS=%JAVA_OPTS% -Dsun.lang.ClassLoader.allowArraySyntax=true
```

  On Unix, use this syntax instead:

```
JAVA_OPTS="$JAVA_OPTS -Dsun.lang.ClassLoader.allowArraySyntax=true"
```

For more configuration options, please see the Embedded JBoss Tomcat integration *wiki entry* [http://wiki.jboss.org/wiki/Wiki.jsp?page=EmbeddedAndTomcat].

30.6.2. 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
         jboss-el.jar
         jsf-facelets.jar
         jsf-api.jar
         jsf-impl.jar
         ...
         my-application.jar/
            META-INF/
               MANIFEST.MF
               persistence.xml
            seam.properties
            org/
               jboss/
                  myapplication/
                     User.class
                     Login.class
                     LoginBean.class
                     Register.class
                     RegisterBean.class
                     ...
   login.jsp
   register.jsp
   ...
```

Most of the Seam example applications may be deployed to Tomcat by running `ant deploy.tomcat`.

30.7. 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`:

```
<bpm:jbpm>
   <bpm:pageflow-definitions>
      <value>createDocument.jpdl.xml</value>
      <value>editDocument.jpdl.xml</value>
      <value>approveDocument.jpdl.xml</value>
   </bpm:pageflow-definitions>
   <bpm:process-definitions>
      <value>documentLifecycle.jpdl.xml</value>
   </bpm:process-definitions>
</bpm: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="tx" factory="org.jbpm.tx.TxServiceFactory" />
   <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.DefaultAuthenticationServiceFactory" />
 </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.

30.7.1. 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
  lib/
    jboss-el.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
```

30.8. Configuring SFSB and Session Timeouts in JBoss AS

It is very important that the timeout for Stateful Session Beans is set higher than the timeout for HTTP Sessions, otherwise SFSB's may time out before the user's HTTP session has ended. JBoss Application Server has a default session bean timeout of 30 minutes, which is configured in `server/default/conf/standardjboss.xml` (replace *default* with your own configuration).

The default SFSB timeout can be adjusted by modifying the value of `max-bean-life` in the `LRUStatefulContextCachePolicy` cache configuration:

```
<container-cache-conf>
   <cache-policy>org.jboss.ejb.plugins.LRUStatefulContextCachePolicy</cache-policy>
   <cache-policy-conf>
      <min-capacity>50</min-capacity>
      <max-capacity>1000000</max-capacity>
      <remover-period>1800</remover-period>

      <!-- SFSB timeout in seconds; 1800 seconds == 30 minutes -->
      <max-bean-life>1800</max-bean-life>

      <overager-period>300</overager-period>
      <max-bean-age>600</max-bean-age>
      <resizer-period>400</resizer-period>
      <max-cache-miss-period>60</max-cache-miss-period>
      <min-cache-miss-period>1</min-cache-miss-period>
      <cache-load-factor>0.75</cache-load-factor>
   </cache-policy-conf>
</container-cache-conf>
```

The default HTTP session timeout can be modified in `server/default/deploy/jbossweb-tomcat55.sar/conf/web.xml` for JBoss 4.0.x, or in `server/default/deploy/jboss-web.deployer/conf/web.xml` for JBoss 4.2.x or later. The following entry in this file controls the default session timeout for all web applications:

```
<session-config>
   <!-- HTTP Session timeout, in minutes -->
   <session-timeout>30</session-timeout>
</session-config>
```

To override this value for your own application, simply include this entry in your application's own `web.xml`.

30.9. Running Seam in a Portlet

If you want to run your Seam application in a portlet, take a look at the JBoss Portlet Bridge, an implementation of JSR-301 that supports JSF within a portlet, with extensions for Seam and RichFaces. See *http://labs.jboss.com/portletbridge* for more.

30.10. Deploying custom resources

Seam scans all jars containing `/seam.properties`, `/META-INF/components.xml` or `/META-INF/seam.properties` on startup for resources. For example, all classes annotated with `@Name` are registered with Seam as Seam components.

You may also want Seam to handle custom resources. A common use case is to handle a specific annotation and Seam provides specific support for this. First, tell Seam which annotations to handle in `/META-INF/seam-deployment.properties`:

```
# A colon-separated list of annotation types to handle
org.jboss.seam.deployment.annotationTypes=com.acme.Foo:com.acme.Bar
```

Then, during application startup you can get hold of all classes annotated with `@Foo`:

```
@Name("fooStartup")
@Scope(APPLICATION)
@Startup
public class FooStartup {

   @In("#{deploymentStrategy.annotatedClasses['com.acme.Foo']}")
   private Set<Class<Object>> fooClasses;

   @In("#{hotDeploymentStrategy.annotatedClasses['com.acme.Foo']}")
   private Set<Class<Object>> hotFooClasses;

   @Create
   public void create() {
```

```
    for (Class clazz: fooClasses) {
      handleClass(clazz);
    }
    for (Class clazz: hotFooClasses) {
      handleClass(clazz);
    }
  }

  public void handleClass(Class clazz) {
     // ...
  }

}
```

You can also handle *any* resource. For example, you process any files with the extension `.foo.xml`. To do this, we need to write a custom deployment handler:

```
public class FooDeploymentHandler implements DeploymentHandler {
   private static DeploymentMetadata FOO_METADATA = new DeploymentMetadata()
   {

      public String getFileNameSuffix() {
         return ".foo.xml";
      }
   };

  public String getName() {
    return "fooDeploymentHandler";
  }

   public DeploymentMetadata getMetadata() {
      return FOO_METADATA;
   }
}
```

Here we are just building a list of any files with the suffix `.foo.xml`.

Then, we need to register the deployment handler with Seam in `/META-INF/seam-deployment.properties`. You can register multiple deployment handler using a comma separated list.

```
# For standard deployment
```

```
org.jboss.seam.deployment.deploymentHandlers=com.acme.FooDeploymentHandler
# For hot deployment
org.jboss.seam.deployment.hotDeploymentHandlers=com.acme.FooDeploymentHandler
```

Seam uses deployment handlers internally to install components and namespaces. You can easily access the deployment handler during an `APPLICATION` scoped component's startup:

```
@Name("fooStartup")
@Scope(APPLICATION)
@Startup
public class FooStartup {

  @In("#{deploymentStrategy.deploymentHandlers['fooDeploymentHandler']}")
  private FooDeploymentHandler myDeploymentHandler;

  @In("#{hotDeploymentStrategy.deploymentHandlers['fooDeploymentHandler']}")
  private FooDeploymentHandler myHotDeploymentHandler;

  @Create
  public void create() {
    for (FileDescriptor fd: myDeploymentHandler.getResources()) {
        handleFooXml(fd);
    }

    for (FileDescriptor f: myHotDeploymentHandler.getResources()) {
        handleFooXml(fd);
    }
  }

  public void handleFooXml(FileDescriptor fd) {
     // ...
  }
}
```

Les annotations Seam

Quand vous écrivez une application Seam, vous serez amené à utiliser de nombreuses annotations. Avec les annotations, Seam vous permettra de programmer de façon déclarative. La plupart des annotations que vous utiliserez appartiennent à la spécification EJB3. Les annotations pour la validation des données appartiennent au package Hibernate Validator. Enfin Seam définit lui-même son propre sous-ensemble d'annotations, que nous allons décrire dans ce chapitre.

Toutes ces annotations sont définis dans le package `org.jboss.seam.annotations`.

31.1. Les annotations pour la définition des composants.

Le premier groupe d'annotations vous permet de définir un composant Seam. Ces annotations sont déclarées au niveau de la classe.

`@Name`

```
@Name("componentName")
```

Définit le nom du composant Seam pour une classe donnée. Cette annotation est obligatoire pour tout composant Seam.

`@Scope`

```
@Scope(ScopeType.CONVERSATION)
```

Définit le scope (contexte) par défaut auquel ce composant appartient. Les valeurs qui peuvent être prises sont définies par l'énumération `ScopeType` : `EVENT, PAGE, CONVERSATION, SESSION, BUSINESS_PROCESS, APPLICATION, STATELESS`.

Quand aucun scope n'est défini, la valeur par défaut dépend du type du composant. Pour les beans session stateless, la valeur par défaut est `STATELESS`. Pour les beans entité et les bean session stateful la valeur par défaut est `CONVERSATION`. Pour les JavaBeans, la valeur par défaut est `EVENT`.

`@Role`

```
@Role(name="roleName", scope=ScopeType.SESSION)
```

Permet à un composant Seam d'être lié à plusieurs variables de contexte à la fois. Les annotations `@Name`/`@Scope` définissent en quelques sorte un "role par défaut". Mais chaque annotation `@Role` permet de définir alors un rôle supplémenataire.

- `name` — Le nom de la variable dans le contexte.
- `scope` — Le scope auquel cette variable appartient. Quand aucun scope n'est spécifié, la valeur par défaut dépend du type de composant comme décrit ci-dessus.

`@Roles`

```
@Roles({
      @Role(name="user", scope=ScopeType.CONVERSATION),
      @Role(name="currentUser", scope=ScopeType.SESSION)
  })
```

Permet de spécifier des rôles additionnels.

`@BypassInterceptors`

```
@BypassInterceptors
```

Désactive toutes interceptions de Seam sur le composant ou sur l'une de ses méthodes.

`@JndiName`

```
@JndiName("my/jndi/name")
```

Spécifie le nom JNDI que Seam utilisera pour rechercher le composant EJB. Si aucun nom JNDI n'est explicitement spécifié, Seam utilisera le pattern JNDI spécifié par `org.jboss.seam.core.init.jndiPattern`.

`@Conversational`

```
@Conversational
```

Spécifie que le scope du composant est conversationnel, cela signifie qu'aucune méthode du composant ne peut être invoquée à moins qu'une conversation longue soit active.

`@PerNestedConversation`

```
@PerNestedConversation
```

Limite la visibilité du composant au scope de la conversation dans laquelle il a été créé. Cette instance du composant ne sera pas visible par les conversations enfants, qui obtiendront alors leur propre instance de ce composant.

Attention, cette définition présente ses propres limites, car elle implique qu'un composant sera visible depuis certaines parties de la requète, mais plus après l'éxécution de ces dernières. Nous ne recommandons donc pas l'utilisation de cette fonctionnalité.

`@Startup`

```
@Scope(APPLICATION) @Startup(depends="org.jboss.seam.bpm.jbpm")
```

Spécifie qu'un composant ayant un scope application est démarré immédiatement pendant la phase d'initialisation de l'application. C'est principalement utilisé pour les composant natif de Seam qui ont des rôles structurels critiques comme jNDI, Datasources, etc.

```
@Scope(SESSION) @Startup
```

Spécifie qu'un composant ayant un scope session est démarré immédiatemnt durant la phase de création de la session.

- `depends` — spécifie que les composants nommés doivent être démarrés en premier, si ils sont installés.

`@Install`

```
@Install(false)
```

Spécifie si un composant doit ou ne doit pas par défaut être installé . L'absence de l'annotation `@Install` indique implicitement qie le composant doit être installé.

```
@Install(dependencies="org.jboss.seam.bpm.jbpm")
```

Spécifie qu'un composant ne doit être installé que si les composants listés comme des dépendances sont aussi installés.

```
@Install(genericDependencies=ManagedQueueSender.class)
```

Spécifie qu'un composant ne doit être installé que si des composants implémentés par certaines classes sont aussi installés. Ceci est utile quand on ne dispose pas des noms précis des composants présents dans l'application, mais que l'on connait leur classe.

```
@Install(classDependencies="org.hibernate.Session")
```

Spécifice qu'un composant de doit être installé que si la classe nommée est présente dans le classpath.

```
@Install(precedence=BUILT_IN)
```

Spécifie la précédence du composant. Si plusieurs composants existe avec le même nom, c'est celui qui aura la précédence la plus élevée qui sera installé. Les différentes valeurs de précédences sont (dans l'ordre croissant) :

- `BUILT_IN` — Précédence attribuée à tout les composants appartenant à Seam
- `FRAMEWORK` — Précédence attribuée à tous les coposants issus des frameworks qui étendent Seam.
- `APPLICATION` — Precedence of application components (the default precedence)
- `DEPLOYMENT` — Précedence attribuée à tous les composants qui surdéfinissent les composants application dans un scénario de déploiement particulier.
- `MOCK` — Précédence atribuée pour les objets "Mock" utilisés pendant les tests.

`@Synchronized`

```
@Synchronized(timeout=1000)
```

Spécifie qu'un composant est accédé de façon concurentielle par plusieurs clients et que Seam doit alors placer les requètes dans une file d'attente. Si une requète ne parvient pas à obtenir le verrou sur le composant dans un laps de temps égal au timeout, une exception est levée.

`@ReadOnly`

```
@ReadOnly
```

Spécifie qu'un composant JavaBean ou une méthode de composant de nécessite pas une réplication d'état à l'issu de son invocation.

`@AutoCreate`

```
@AutoCreate
```

Spécifie que le composant sera automatiquement créé, même si le code client ne spécifie pas `create=true`.

31.2. Les annotations pour les bijections

Les deux annotations suivantes contrôle le processus de bijection. Ces attributs peuvent être placés sur les propriétés du composants ou sur leurs accesseurs (ie getter et setter)

`@In`

```
@In
```

Spécifie qu'une variable de contexte doit être injecté dans un attribut de composant au début de chaque invocation du composant. Si la variable de contexte est nulle une exception est levée.

```
@In(required=false)
```

Spécifie qu'une variable de contexte doit être injecté dans un attribut de composant au début de chaque invocation du composant. Cette variable de contexte peut être nulle.

```
@In(create=true)
```

Spécifie qu'une variable de contexte doit être injecté dans un attribut de composant au début de chaque invocation du composant. Si la variable est nulle, alors Seam instanciera une instance de ce composant.

```
@In(value="contextVariableName")
```

Spécifie le nom de la variable de contexte explicitement au lieu d'utiliser le nom de la variable annotée.

```
@In(value="#{customer.addresses['shipping']}")
```

Spécifie que l'attribut du composant doit être injecté en évaluant une expression JSF EL au début de chaque invocation du composant.

- `value` — spécifie le nom de la variable de contexte. Par défaut le nom de l'attribut du composant sera utilisé. Alternativement une expression JSF EL peut être fourni en respectant la syntaxe `#{...}`.
- `create` — Spécifie que Seam devra instancié le composant avec le même nom que la variable de contexte si celle-ci ne peut être trouvé dans aucun contexte . Faux par défaut.
- `required` — Spécifie que Seam doit envoyer une exception, si la variable de contexte ne peut être trouvé dans aucun des contextes.

`@Out`

```
@Out
```

Spécifie que l'attribut de composant, qui est aussi un composant Seam, doit être "exposé" dans l'un des contextes (scopes) de Seam à la fin de chaque invocation du composant. Si l'attribut est null, une exception est levée.

```
@Out(required=false)
```

Spécifie qu'un attribut de composant, qui est aussi un composant Seam, doit être "exposé" dans l'un des contextes (scopes) de Seam à la fin de chaque invocation du composant. L'attribut peut-être null.

```
@Out(scope=ScopeType.SESSION)
```

Spécifie qu'un attribut de composant qui *n'est pas* un composant Seam doit être exposé dans un scope (context) spécifique de Seam à la fin de l'invocation du composant.

Mais si aucun scope (context) n'est spécifié, c'est le scope du composant qui porte l'attribut annoté `@Out` qui sera utilisé (Si le composant porteur est stateless alors l'attribut annoté sera oujecté dans le scope (context) `EVENT`).

```
@Out(value="contextVariableName")
```

Spécifie le nom de la variable de contexte explicitement au lieu d'utiliser le nom de la variable annotée.

- `value` — spécifie le nom de la variable de contexte. Par défaut c'est le nom de l'attribut du composant qui sera utilisé.
- `required` — spécifie que Seam doit lever une exception, si le composant est null durant l'outjection.

Remarquer qu'il est assez fréquent pour ces annotations d'être utilisées ensemble, par exemple :

```
@In(create=true) @Out private User currentUser;
```

The next annotation supports the *manager component* pattern; a Seam component manages the lifecycle of an instance of some other class that is to be injected. It appears on a component getter method.

`@Unwrap`

```
@Unwrap
```

Spécifie que l'objet retourné par la méthode getter annotée sera injecté à la place du composant lui-même.

The next annotation supports the *factory component* pattern; 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") public void createProcessInstance() { ... }
```

Spécifie que la méthode annotée du composant sera invoquée pour initialiser la valeur de la variable de contexte quand celle-ci n'a pas encore de valeur. Ce style est utilisée avec des méthodes ne renvoyant aucune valeur (type de retour `void`).

```
@Factory("processInstance", scope=CONVERSATION) public ProcessInstance
 createProcessInstance() { ... }
```

Spécifie que la méthode renvoie une valeur que Seam devra utiliser pour pour initialiser la variable de contexte lorsque celle-ci n'a pas de valeur. Ce style est utilisé avec les méthodes

renvoyant une valeur. Si aucun scope n'est spécifié, le scope du composant possedant la méthode annotée `@Factory` est utilisée (a moins que le composant soit stateless dans ce cas c'est le scope (context) `EVENT` qui sera utilisée).

- `value` — spécifie le nom de la variable de contexte. Si cette méthode est un getter, utilise par défaut le nom de la propriété JavaBeans.
- `scope` — spécifie le scope dans lequel Seam devra placer la variable. Ca n'a de sens que pour les méthodes annotées `@Factory` qui renvoient une valeur.
- `autoCreate` — spécifie que la méthode de fabrique doit être automatiquement invoquée a chaque fois que la variable est demandée, même si `@In` ne spécifie pas `create=true`.

Cette annottion vous permet d'injecter un `Log`:

`@Logger`

```
@Logger("categoryName")
```

Spécifie qu'il faut injecter dans un attribut du composant une instance de `org.jboss.seam.log.Log`. Pour un bean entité il faut que l'attribut injecté soit déclaré statique.

- `value` — spécifie le nom de la catégorie de log. Par défaut le nom du composant est utilisé.

La dernière annotation vous permet d'injecter un paramètre de requète.

`@RequestParameter`

```
@RequestParameter("parameterName")
```

Spécifie que l'on injecte dans l'attribut du composant un paramètre de requète. Les conversions des types de base sont faits automatiquement.

- `value` —spécifie le nom du paramètre de requète. Par défaut on utilise le nom de l'attribut dans le composant.

31.3. Les annotations pour les méthodes de cycle de vie.

Ces annotations permettent à un composant de réagir aux événement de son propre cycle de vie. On ne peut en déclarer qu'une par classe.

`@Create`

```
@Create
```

Spécifie que cette méthode doit être appellée quand une instance du composant a été instancié par Seam. Attention les méthodes annotés `@Create` ne nont applicable que pour les JavaBean et les beans session stateful.

`@Destroy`

```
@Destroy
```

Spécifie que la méthode doit être invoqué lorsque que le contexte se termine et que chaque variable de celui-ci va être supprimée. Attention seul les JavaBean et les session bean stateful peuvent déclaré des méthodes destroy.

Les méthodes destroy ne devrait être utilisés que pour la libération des ressources. *Seam intercepte et log toutes exceptions qu'une méthodes destroy pourrait propager*.

`@Observer`

```
@Observer("somethingChanged")
```

Spécifie que la méthode devrait être invoquée lorsqu'un événement issus d'un composant est émis.

```
@Observer(value="somethingChanged",create=false)
```

Spécifie que la méthode doit être invoquée si un évenement de ce type survient, mais que l'instance du composant écouteur ne devrait pas être créé si il n'existe pas. Si l'instance n'existe pas l'événement ne sera pas écouté. La valeur par défaut de create est true.

31.4. Les annotations pour la démarcation de contexte

Ces annotations permetent de démarquer de façon déclarative les conversations. Elles apparaissent le plus souvent sur des méthodes écouteurs.

Chaque requete web a un context de conversation qui lui est associé. La plupart de ces conversations s'acheve avec la requète. Si vous voulez que la conversation s'étende sur plusieurs requètes il vous faudra "Promouvoir" la conversation courante au rang de *longue conversation* en invoquent une méthode annotée `@Begin`.

`@Begin`

```
@Begin
```

Spécifie qu'une conversation longue démarre quand cette méthode renvoie une valeur non nulle sans lever d'exception.

```
@Begin(join=true)
```

Spécifie qu'une longue conversation a déjà démarré, le contexte the conversation est simplement propagé.

```
@Begin(nested=true)
```

Spéciifie qu'une longue conversation est déjà engagée, un nouveau contexte de *conversation enfant* est démarré. La conversation enfant se terminera à la prochaine invoquation d'une méthode annotée `@End`, la conversation parente reprendra alors. Il est tout à fait possible que plusieurs conversations enfants existent concurentiellement au sein de la même conversation parente.

```
@Begin(pageflow="process definition name")
```

Spécifie q'une définition de pageflow JBPM sera utilisée pour conduire cette conversation.

```
@Begin(flushMode=FlushModeType.MANUAL)
```

Spécifie le flush mode de tous les contextes de persistence managés par Seam. `flushMode=FlushModeType.MANUAL` apporte la notion de *conversation atomique* où toutes opérations d'écriture est mis en queue dans le contexte de persistence jusqu'à ce qu'un appel explicite à `flush()` ait lieu (en général à la fin de la conversation).

- `join` — détermine le mode de propagation lorsque qu'une conversation est déjà engagée. Si join vaut `true`,le contexte est propagé. Si join vaut `false`, un exception est levée. Par défaut join vaut `false`. Ce paramétrage est ignoré si `nested=true`est spécifié.
- `nested` — Spécifie qu'une conversation enfant devrait être démarrée si une longue conversation est déjà engagée.

- `flushMode` — fixe le flush mode de toutes les sessions Hibernate ou de tous les contextes de persistence managé par Seam qui seront créés durant cette conversation.
- `pageflow` — Le nom de la définition d'un process jBPM qui sera déployé via `org.jboss.seam.bpm.jbpm.pageflowDefinitions.`

`@End`

```
@End
```

Spécifie qu'une longue conversation s'achève si la méthode retourne une valeur non null sans lever d'exception.

- `beforeRedirect` — par défaut une longue conversation ne sera détruite qu'après qu'un redirect ait eu lieu. Choisir `beforeRedirect=true` spécifie que la conversation doit être détruite à la fin de la requête courante, et que la redirection aura lieu dans un nouveau contexte temporaire de conversation.
- `root` — by default, ending a nested conversation simply pops the conversation stack and resumes the outer conversation. Setting `root=true` specifies that the root conversation should be destroyed which effectively destroys the entire conversation stack. If the conversation is not nested, the current conversation is simply ended.

`@StartTask`

```
@StartTask
```

Démarre une nouvelle tâche JBPM. Spécifie aussi qu'une longue conversation démarre lorsque la méthode ainsi annotée a un retour non null sans lever d'exception. La tâche JBPM qui sera associée à ce contexte de conversation sera spécifiée grâce à un paramètre de requête nommé, le buisness process associé à cette tâche est lui aussi associé à cette conversation.

- The jBPM `TaskInstance` will be available in a request context variable named `taskInstance`. The jBPM `ProcessInstance` will be available in a request context variable named `processInstance`. (Of course, these objects are available for injection via `@In`.)
- `taskIdParameter` — le nom du paramètre de requète qui contient l'id de la tâche JBPM. Par défaut ce nom est `"taskId"`, ce qui est aussi le nom par défaut que le composant Seam JSF `taskList` utilise.
- `flushMode` — fixe le flush mode de toutes les sessions Hibernate ou de tous les contextes de persistence managé par Seam qui seront créés durant cette conversation.

`@BeginTask`

```
@BeginTask
```

Reprend une tâche JBPM qui n'a pas été terminé. Spécifie aussi qu'une longue conversation démarre lorsque la méthode ainsi annotée a un retour non null sans lever d'exception. La tâche JBPM qui sera associée à ce contexte de conversation sera spécifiée grâce à un paramètre de requête nommé, le buisness process associé à cette tâche est lui aussi associé à cette conversation.

- The jBPM `org.jbpm.taskmgmt.exe.TaskInstance` will be available in a request context variable named `taskInstance`. The jBPM `org.jbpm.graph.exe.ProcessInstance` will be available in a request context variable named `processInstance`.
- `taskIdParameter` — le nom du paramètre de requète qui contient l'id de la tâche JBPM. Par défaut ce nom est `"taskId"`, ce qui est aussi le nom par défaut que le composant Seam JSF `taskList` utilise.
- `flushMode` — fixe le flush mode de toutes les sessions Hibernate ou de tous les contextes de persistence managé par Seam qui seront créés durant cette conversation.

`@EndTask`

```
@EndTask
```

Termine une tache JBPM. Spécifie qu'une conversation longue termine si la méthode ainsi annotée renvoie un résultat non null sans lever d'exception, spécifie aussi que la tâche courante a été complétée. Ceci déclanche alors une transition JBPM. La transition déclanchée sera alors la transition par défaut à moins que l'application n'ait invoqué `Transition.setName()` sur le composant intégré dont le nom est `transition`.

```
@EndTask(transition="transitionName")
```

Déclanche la transition JBPM.

- `transition` — le nom de la transiton qui sera déclanchée lorsque l'on terminera la tâche. Si non spécifiée, c'est la transition par défaut qui sera utilisée.
- `beforeRedirect` — par défaut une longue conversation ne sera détruite qu'après qu'un redirect ait eu lieu. Choisir `beforeRedirect=true` spécifie que la conversation doit être détruite à la fin de la requête courante, et que la redirection aura lieu dans un nouveau contexte temporaire de conversation.

`@CreateProcess`

```
@CreateProcess(definition="process definition name")
```

Crée une nouvelle instance d'un JBPM process lorsque la méthode ainsi annotée retourne un résultat non null sans lever d'exception L'object `ProcessInstance` sera disponible dans une variable de contexte nomée `processInstance`.

- `definition` — le nom du process JBPM qui sera déployé via `org.jboss.seam.bpm.jbpm.processDefinitions`.

`@ResumeProcess`

```
@ResumeProcess(processIdParameter="processId")
```

Ré-entre dans le scope d'une instance de process JBPM lorsque la méthode ainsi annotée renvoie une valeur non null sans lever d'exception. L'object `ProcessInstance` sera alors accessible grâce à une variable de contexte nomée `processInstance`.

- `processIdParameter` — défini le nom du paramètre http qui détient l'id du process. Par défaut on utilise `"processId"`.

`@Transition`

```
@Transition("cancel")
```

Une méthode ainsi annotée déclenche une transition au sein du contexte JBPM courant à chaque fois qu'elle renvoie une valeur non null.

31.5. Les annotations utilisées avec les composants JavaBean Seam dans une environnement JEE

Seam met à disposition des annotations qui permettent de forcer un rollback JTA en fonction des retours renvoyer par un listener.

`@Transactional`

```
@Transactional
```

Spécifie que le composant JavaBean doit avoir un comportement similaire à celui d'un bean session. C'est à dire que l'invocation de la méthode doit rejoindre une transaction, et si aucune

transaction n'existe lors de la l'appel de la méthode, une transaction sera crée juste pour cette méthode. Cette annotation peut s'appliquer au niveau de la classe toute entière ou au niveau d'une méthode.

N'utiliser pas cette annotation sur un composant EJB 3.0, utiliser`@TransactionAttribute`!

`@ApplicationException`

```
@ApplicationException
```

Synonyme de javax.ejb.ApplicationException, à utiliser dans les environnements pre Java EE 5. Elle est appliquéee sur une exception pour qualifier l'exception "d'exception applicative" et doit être remontée directement au client (c'est à dire sans être encapsulée).

N'utiliser pas cette annotation sur les composants EJB 3.0, utiliser à la place `@javax.ejb.ApplicationException` instead.

- `rollback` — `false` par défaut, si `true` l'exception doit passer la transaction à rollback only
- `end` —`false` par défaut, si `true` l'exception doit terminer la conversation longue

`@Interceptors`

```
@Interceptors({DVDInterceptor, CDInterceptor})
```

Synonyme de javax.interceptors.Interceptors, à utiliser dans les environnements pre Java EE 5. Remarquer que c'est à utiliser comme une meta-annotation. Declare une liste d'intercepteurs pour une classe ou une méthode.

n'utiliser pas cette annotation sur les composants EJB 3.0, utiliser `@javax.interceptor.Interceptors` à la place.

Ces annotations sont essentiellement utilisées pour les composants Seam JavaBean. Si vous utiliser des composants EJB 3.0, vous devriez plutôt utiliser les annotations standards Java EE5.

31.6. Les annotations pour les exceptions

Ces annotations vous permettent de spécifier comment Seam doit prendre en charge les exceptions qui se propagent hors d'un composant Seam.

`@Redirect`

```
@Redirect(viewId="error.jsp")
```

Spécifie que l'exception annotée provoque une redirection du navigateur vers la vue ayant pour id viewID

- `viewId` — spécifie la vue JSF vers laquelle if faut faire la redirection. Vous pouvez utiliser ici une expression EL.
- `message` — le message qui doit être présenté, si non spécifié c'est le message par défaut qui est utilisé.
- `end` — spécifie que la longue conversation en cours doit se terminer, `false` par défaut.

`@HttpError`

```
@HttpError(errorCode=404)
```

Spécifie que l'exception annotées cause l'envoie d'une 'erreur HTTP.

- `errorCode` — le code de l'erreur HTTP, `500` par défaut.
- `message` — un message devant acompagner l'erreur HTTP, par défaut on utilise le message de l'exception.
- `end` — spécifie que la longue conversation en cours doit se terminer, `false` par défaut.

31.7. Les annotations pour Seam Remoting

Seam remoting exige que les interfaces locale d'un bean de session soient annotées avec les annotations suivantes :

`@WebRemote`

```
@WebRemote(exclude="path.to.exclude")
```

Indique que la méthode annotée peut être appelée depuis le code client JavaScript. La propriétété `exclude` est optionnelle, elle permet aux objets d'être exclu du graphe résultant (voir le chapitre Remoting pour plus de détails).

31.8. Les annotations pour les intercepteurs de Seam.

Les annotations suivantes sont utilisées pour les intercepteurs Seam.

Nous vous demandons de vous référer à la documentation EJB 3.0 pour connaître les spécifications des intercepteurs EJB.

`@Interceptor`

```
@Interceptor(stateless=true)
```

Spécifie que l'intercepteur est stateless et que Seam pourra optimiser leur réplication.

```
@Interceptor(type=CLIENT)
```

Spécifie que c'est un intercepteur "coté client" qui est appelé avant le conteneur EJB.

```
@Interceptor(around={SomeInterceptor.class, OtherInterceptor.class})
```

Spécifie que l'intercepteur est positionné plus haut dans la pile d'intercepteurs que les intercepteurs cités dans l'énumération.

```
@Interceptor(within={SomeInterceptor.class, OtherInterceptor.class})
```

Spécifie que l'intercepteur est positionné plus bas dans la pile d'intercepteurs que les intercepteurs cités dans l'énumération.

31.9. Les annotations pour l'asynchronicité

Les annotations suivantes sont utilisées pour déclarer des méthodes asynchrones, par exemple:

```
@Asynchronous public void scheduleAlert(Alert alert, @Expiration Date date) { ... }
```

```
@Asynchronous public Timer scheduleAlerts(Alert alert,
  @Expiration Date date,
  @IntervalDuration long interval) { ... }
```

`@Asynchronous`

```
@Asynchronous
```

Spécifie que l'invocation de la methode a lieu de façon asynchrone.

`@Duration`

```
@Duration
```

Spécifie que l'un des paramètres de l'appel asynchrone est la durée avant laquelle l'appel asynchrone aura lieu (ou en cas de récurrence le temps avant le premier appel)

`@Expiration`

```
@Expiration
```

Spécifie que l'un des paramètres de l'appel asynchrone est la date à laquelle l'appel asynchrone aura lieu (ou en cas de récurrence le temps avant le premier appel)

`@IntervalDuration`

```
@IntervalDuration
```

Spécifie que l'appel asynchrone de la méthode est récurrent, et que le paramètre annoté est la durée entre chaque appel.

31.10. Les annotations utilisés avec JSF

Les annotations suivantes permettent de travailler plus facilement avec JSF.

`@Converter`

Permet à un composant Seam d'agir comme un converter. La classe annoté doit être un composant Seam, et doit implémenter `javax.faces.convert.Converter`.

- `id` — l'id du converter JSF. Par défaut c'est le nom du composant qui est utilisé.
- `forClass` — si spécifié, enregistre le composant comme le converter par défaut pour ce type.

`@Validator`

Permet à un composant Seam d'agir comme un validator. La classe annotée doit être un composant Seam, et doit implémenter `javax.faces.validator.Validator`.

- `id` — l'id JSF du validator. Par défaut c'est le nom du composant qui est utilisé.

31.10.1. Les annotations pour `dataTable`

Les annotations suivantes facilitent l'implémentations de listes cliquables référencées par des bean session stateful Ils apparaissent sur les attributs.

`@DataModel`

```
@DataModel("variableName")
```

Expose une propriété de type `List`, `Map`, `Set` ou `Object[]` comme un `DataModel` JSF dans le scope du composant qui le porte (ou dans le scope `EVENT`si le scope du composant porteur est `STATELESS`). dans le cas d'une `Map`, chaque ligne du `DataModel` est une `Map.Entry`.

- `value` — nom de la variable dans le context de conversation. Par défaut on utilise le nom de l'attribut.
- `scope` — si `scope=ScopeType.PAGE` ext explicitemnt spécifié, le `DataModel` sera conservé dans le contexte `PAGE`.

`@DataModelSelection`

```
@DataModelSelection
```

Injecte la valeur du `DataModel` qui a été selectionnée (c'est a dire l'élément de la collection sous-jacente, ou de la map). Si un seul attribut `@DataModel` est défini sur le composant, la valeur sélectionnée du `DataModel` sera injectée. Le nom de chaque `@DataModel` doit être spécifié pour chaque `@DataModelSelection`.

Si le scope `PAGE` est spécifié pour le `@DataModel`, alors le DataModelSelection sera non seulement injecté mais aussi le DataModel , donc si l'attribut `@DataModel` annote une méthode getter, alors le composant devra aussi exposer un setter public.

- `value` — nom de la variable dans le contexte de conversation. Il n'est pas necessaire de le spécifier si il n'y a qu'un `@DataModel` sur le composant.

`@DataModelSelectionIndex`

```
@DataModelSelectionIndex
```

Expose l'index de la ligne selectionnée du `DataModel` comme un attribut du composant (c'est le numéro de la ligne de la collection sous-jacente, ou la valeur de la clé si le DataModel est une map). Si un seul attribut `@DataModel` est défini pour ce composant, la valeur selectionnée du`DataModel` sera injectée. Autrement le nom de chaque `@DataModel` doit être spécifié pour chaque `@DataModelSelectionIndex`.

- `value` — nom de la variable dans le contexte de conversation. Il n'est pas necessaire de le spécifier si il n'y a qu'un `@DataModel` sur le composant.

31.11. Les metas-annotations pour le databinding

Ces meta-annotations permettent d'implémenter des fonctionnalités similaires à `@DataModel` et `@DataModelSelection` pour des structures de données différentes des listes.

`@DataBinderClass`

```
@DataBinderClass(DataModelBinder.class)
```

Spécifie qu'une annotation est utilisée pour le databinding.

`@DataSelectorClass`

```
@DataSelectorClass(DataModelSelector.class)
```

Spécifie qu'une annotation est utilisée pour les dataselection.

31.12. Les annotations pour le packaging

Cette annotation fournit un mécanisme pour déclarer des informations sur un ensemble de composants qui sont packagés ensembles. Elle s'applique à n'importe quel package Java.

`@Namespace`

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

Spécifie que les composants présents dans le package courants sont associés avec le namespace (espace de nom). Le namespace ainsi déclaré peut-être utilisé directement comme un namespace XML dans un fichier `components.xml` pour simplifier la configuration de l'application.

```
@Namespace(value="http://jboss.com/products/seam/core", prefix="org.jboss.seam.core")
```

Spécifie l'espace de nommage associé à un package donné. Additionnellement, il spécifie un préfixe à ajouter au nom du composant lors de sa déclaration dans le fichier XML. Par exemple un élément XML noté `init` qui est associé avec cet espace de nommage serait interprété comme se référant au composant nommé `org.jboss.seam.core.init`.

31.13. Les annotations pour l'intégration avec le conteneur de Servlets

Ces annotations vous permettent d'intégrer vos composants seam avec le conteneur de servlets

`@Filter`

Utiliser ce coposant seam (qui devra implémenter `javax.servlet.Filter`) annoté avec `@Filter` comme un filter de servlet. Il sera exécuté par le filtre Seam principal.

```
@Filter(around={"seamComponent", "otherSeamComponent"})
```

Spécifie que le filtre est positionné plus haut dans la pile que la liste des filtres énumérés.

```
@Filter(within={"seamComponent", "otherSeamComponent"})
```

Spécifie que le filtre est positionné plus bas dans la pile que la liste des filtres énumérés.

Les composants livrés par Seam

Ce chapitre décrit les composants livrés par Seam et leur propriétés de configuration. Les composants livrés par Seam seront crées même si ils ne sont listés dans votre fichier `components.xml`, mais si vous avez besoin de surcharger les propriétés par défaut ou de spécifier plus d'un composant d'un certain type, `components.xml` est à utiliser.

Notez que vous pouvez remplacer tout les composants livrés par vos propres implémentations simplement en spécifiant le nom d'un des composants livrés dans votre propre classe en utilisant `@Name`.

32.1. Les composant d'injection de contexte

Le premier groupe de composants livré existe primitivement pour supporter l'injection des différents 'objets contextuel. Par exemple, les instances de composants suivantes devrait avoir l'objet contexte de session de Seam injecté:

```
@In private Context sessionContext;
```

`org.jboss.seam.core.contexts`
: Le composant qui fourni un accès aux objets de Contexte de Seam, par exemple `org.jboss.seam.core.contexts.sessionContext['user']`.

`org.jboss.seam.faces.facesContext`
: Le composant de gestion pour l'objet de contexte `FacesContext` (pas un vrai contexte de Seam)

Tous ces composants sont toujours installés.

32.2. Les composants liés à JSF

Le groupe suivant de composants est fourni pour suppléer JSF.

`org.jboss.seam.faces.dateConverter`
: Fourni un convertisseur JSF par défaut pour les propriétés de type `java.util.Date`.

  Ce convertisseur est automatiquement activé avec JSF. Il est fourni pour éviter au développeur d'avoir à spécifier un DateTimeConverter sur un champs de saisie ou en paramètre de page. Par défaut, il suppose que le type doit être une date (à la différence d'une heure ou de la date plus l'heure) et utilise le style de saisie de date courte ajustée à la Locale de l'utilisateur. Pour la Locale .US, le patron d'entrée est mm/DD/yy. Cependant pour être valide avec Y2K, l'année est modifié de deux chiffres vers quatre (autrement dit mm/DD/yyyy).

C'est possible de surcharger le patron d'entrée globallement en utilisation la configuration du composant. Consultez la JavaDoc pour cette classe pour voir les exemples.

`org.jboss.seam.faces.facesMessages`
Permet aux messages de succès faces de se propager au travers d'une redirection du navigateur.

- `add(FacesMessage facesMessage)` — ajoute un message faces, qui sera affiché pendant la porchaine phase de rendue de la réponse qui intervendra dans cette conversation courante.
- `add(String messageTemplate)` — ajoute un message faces, rendu depuis un modèle de message donnée qui peut contenir des expressions EL.
- `add(Severity severity, String messageTemplate)` — ajoute un message faces, rendu depuis un modèle de message donnée qui peut contenir des expressions EL.
- `addFromResourceBundle(String key)` —ajoute un message faces, rendu depuis un modèle de message défini dans le fichier de ressource livré de Seam qui peut contenir des expressions EL.
- `addFromResourceBundle(String key)` —ajoute un message faces, rendu depuis un modèle de message défini dans le fichier de ressource livré de Seam qui peut contenir des expressions EL.
- `clear()` — efface tous les messages.

`org.jboss.seam.faces.redirect`
Une API utile pour réaliser les redictions avec paramètres (c'est particulièrement utile pour les écrans de résultat de recherche à utiliser en favoris).

- `redirect.viewId` — l'identifiant de vue JSF à rediriger vers.
- `redirect.conversationPropagationEnabled` — détermines si la conversation sera propagé au travers d'une redirection.
- `redirect.parameters` — une table de hachage de nom de paramètre de requête et de valeur, doit être passé dans la requête de redirection.
- `execute()` — réalise la redirection immédiatement.
- `captureCurrentRequest()` — stoque l'identifiant de vue et les paramètres de la requête de la requête courrant GET (dans le contexte de conversation), pour l'utiliser plus tard en appelant `execute()`.

`org.jboss.seam.faces.httpError`
Une API utile pour les envoyer les erreurs HTTP.

`org.jboss.seam.ui.renderStampStore`
Un composant (d'étendue de session par défaut) responsable de la conservation d'une collection d'empreinte de rendue. Une empreinte de rendue est un indicateur fait pour savoir

si un formulaire qui a été rendu a été soumis. Ce stockage est utile quand la méthode de sauvegarde de l'état côté client de JSF a été utilisée par ce qu'il indique que l'objectif a été d'avoir un formulaire qui a été soumis dans le contrôle du serveur au lieu de l'avoir dans l'arbre de composant qui a été maintenu sur le client.

Pour détacher cette vérification depuis la session (ce qui est l'un des principaux objectifs du design d'une sauvegarde d'état côté client) une implémentation doit être fournie pour stocker les empreintes de rendue pour l'application (valide aussi longtemps que l'application est exécutée) ou pour la base de données (valide au travers des rédémarrage du serveur).

- `maxSize` — Le nombre maximun d'empreintes qui doient être conservées dans le stockage. Par défaut: 100

Ces composants sont installé quand la classe `javax.faces.context.FacesContext` est disponible dans le classpath.

32.3. Les composants utilitaires

Ces composants sont simplement utiles.

`org.jboss.seam.core.events`
: Une API pour lever les évènements qui peuvent être observé via les méthodes `@Observer`, ou en liaison avec une méthode dans `components.xml`.

- `raiseEvent(String type)` — lève un évènement d'un type particulier et le distribue à tous les observateurs.
- `raiseAsynchronousEvent(String type)` — lève un évènement qui doit être exécuté de manière assynchrone par le service de temps d'EJB3.
- `raiseTimedEvent(String type, ....)` — planifie un évènement qui doit être exécuté par le service de temps EJB3.
- `addListener(String type, String methodBinding)` — ajoute un observateur pour un type d'évènement particulier.

`org.jboss.seam.core.interpolator`
: Une API pour l'interpolation de valeurs des expressions EL de JSF en Strings.

- `interpolate(String template)` — analyse le modèle pour des expressions EL de JSF de la forme `#{...}` et les replace par leurs valeurs évaluées.

`org.jboss.seam.core.expressions`
: Une API pour la création de valeur et la liaison avec les méthodes.

- `createValueBinding(String expression)` — création d'un objet lié avec une valeur.
- `createMethodBinding(String expression)` — création d'un objet lié à une méthode.

`org.jboss.seam.core.pojoCache`
Le composant de gestion pour une instance de Jboss Cache `PojoCache`.

- `pojoCache.cfgResourceName` — le nom du fichier de configuration. Par défaut,`treecache.xml`.

Tous ces composants sont toujours installés.

32.4. Les composants pour l'internationnalisation et les thèmes

Le groupe suivant de composants rend facile la constructions d'interfaces utilisateurs internationnalisés en utilisant Seam.

`org.jboss.seam.core.locale`
La locale de Seam.

`org.jboss.seam.international.timezone`
Le fuseau horaire de Seam. Le fuseau horaire est d'étendue de session.

`org.jboss.seam.core.resourceBundle`
Le fichier de ressource livré dans Seam. Le fichier de ressource livré est sans état. Le fichier de ressource de Seam réalise une recherche du premier résultat dans les clefs dans la liste de fichiers de ressource Java livré.

`org.jboss.seam.core.resourceLoader`
Le chargeur de ressource permet l'accès aux ressources de l'application et aux ressources livrés.

- `resourceLoader.bundleNames` — les noms des ressources Java livrés pour savoir quand le fichier de ressources Seam livré est utilisé. Par défaut à `messages`.

`org.jboss.seam.international.localeSelector`
Permet la sélection de la locale aussi bien pendant la partie configuration ou par l'utilisateur à l'éxécution.

- `select()` — sélectionne la locale spécifiée.
- `localeSelector.locale` — La `java.util.Locale` actuelle.
- `localeSelector.localeString` — La représentation transformée en texte de la locale.
- `localeSelector.language` — la langue pour la locale spécifiée.
- `localeSelector.country` — le pays pour la locale spécifiée.
- `localeSelector.variant` — la variation de la langue pour la locale spécifiée.

- `localeSelector.supportedLocales` — une liste de `SelectItems` réprésentant les locales supportés et listées dans `jsf-config.xml`.
- `localeSelector.cookieEnabled` — spécifie que la sélection de la locale devrait être persistée via un cookie.

`org.jboss.seam.international.timezoneSelector`
: Permet la sélection d'un fuseau horaire pendant la partie configuration, ou par l'utilisateur à l'exécution.

- `select()` — sélectionne la locale spécifiée.
- `timezoneSelector.timezone` — La `java.util.TimeZone` actuelle.
- `timezoneSelector.timeZoneId` — la représentation en texte du fuseau horaire.
- `timezoneSelector.cookieEnabled` — spécifie que la sélection du fuseau horaire devrait êre persisté via un cookie.

`org.jboss.seam.international.messages`
: Une table de hachage contenant le rendu des messages internationnalisés depuis un modèle de message défini dans le fichier de ressource de Seam livré.

`org.jboss.seam.theme.themeSelector`
: Permet la sélection d'un thème aussi bien pendant la phase de configuration que par l'utilisateur pendant l'exécution.

- `select()` — select the specified theme.
- `theme.availableThemes` — the list of defined themes.
- `themeSelector.theme` — the selected theme.
- `themeSelector.themes` — a list of `SelectItems` 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.

Tous ces composants sont toujours installés.

32.5. 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.faces.switcher`
: The conversation switcher.

Tous ces composants sont toujours installés.

32.6. jBPM-related components

These components are for use with jBPM.

`org.jboss.seam.pageflow.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.bpm.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.bpm.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.bpm.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.bpm.taskInstance`
: Manager component for the jBPM `TaskInstance`.

`org.jboss.seam.bpm.processInstance`
: Manager component for the jBPM `ProcessInstance`.

`org.jboss.seam.bpm.jbpmContext`
: Manager component for an event-scoped `JbpmContext`.

`org.jboss.seam.bpm.taskInstanceList`
: Manager component for the jBPM task list.

`org.jboss.seam.bpm.pooledTaskInstanceList`
: Manager component for the jBPM pooled task list.

`org.jboss.seam.bpm.taskInstanceListForType`
: Manager component for the jBPM task lists.

`org.jboss.seam.bpm.pooledTask`
: Action handler for pooled task assignment.

`org.jboss.seam.bpm.processInstanceFinder`
: Manager for the process instance task list.

`org.jboss.seam.bpm.processInstanceList`
: The process instance task list.

All of these components are installed whenever the component `org.jboss.seam.bpm.jbpm` is installed.

32.7. Security-related components

These components relate to web-tier security.

`org.jboss.seam.web.userPrincipal`
: Manager component for the current user `Principal`.

`org.jboss.seam.web.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']}"/>`.

32.8. 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` .

32.9. Mail-related components

These components are for use with Seam's Email support

`org.jboss.seam.mail.mailSession`
: Manager component for a JavaMail `Session`. The session can be either looked up in the JNDI context (by setting the `sessionJndiName` property) or it can created from the configuration options in which case the `host` is mandatory.

  - `org.jboss.seam.mail.mailSession.host` — the hostname of the SMTP server to use
  - `org.jboss.seam.mail.mailSession.port` — the port of the SMTP server to use
  - `org.jboss.seam.mail.mailSession.username` — the username to use to connect to the SMTP server.
  - `org.jboss.seam.mail.mailSession.password` — the password to use to connect to the SMTP server
  - `org.jboss.seam.mail.mailSession.debug` — enable JavaMail debugging (very verbose)
  - `org.jboss.seam.mail.mailSession.ssl` — enable SSL connection to SMTP (will default to port 465)

    `org.jboss.seam.mail.mailSession.tls` — by default true, enable TLS support in the mail session
  - `org.jboss.seam.mail.mailSession.sessionJndiName` — name under which a javax.mail.Session is bound to JNDI. If supplied, all other properties will be ignored.

32.10. Infrastructural components

These components provide critical platform infrastructure. You can install a component which isn't installed by default by setting `install="true"` on the component in `components.xml`.

`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. This should be set to false when in production. You may see errors if the system is placed under any load and debug is enabled.
- `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.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.manager.defaultFlushMode` — set the flush mode set by default on any Seam Managed Persistence Context. By default `AUTO`.

`org.jboss.seam.navigation.pages`
Internal component for Seam workspace management. Always installed.

- `org.jboss.seam.navigation.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.navigation.pages.loginViewId` — global setting for the view id to redirect to when an unauthenticated user tries to access a protected view.
- `org.jboss.seam.navigation.pages.httpPort` — global setting for the port to use when the http scheme is requested.
- `org.jboss.seam.navigation.pages.httpsPort` — global setting for the port to use when the https scheme is requested.
- `org.jboss.seam.navigation.pages.resources` — a list of resources to search for `pages.xml` style resources. Defaults to `WEB-INF/pages.xml`.

`org.jboss.seam.bpm.jbpm`
Bootstraps a `JbpmConfiguration`. Install as class `org.jboss.seam.bpm.Jbpm`.

- `org.jboss.seam.bpm.jbpm.processDefinitions` — a list of resource names of jPDL files to be used for orchestration of business processes.
- `org.jboss.seam.bpm.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.faces.facesPage`
Internal page-scoped component recording the conversation context associated with a page.

`org.jboss.seam.persistence.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 `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 `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.validators`
Caches instances of Hibernate Validator `ClassValidator`.

`org.jboss.seam.faces.validation`
Allows the application to determine whether validation failed or was successful.

`org.jboss.seam.debug.introspector`
Support for the Seam Debug Page.

`org.jboss.seam.debug.contexts`
Support for the Seam Debug Page.

`org.jboss.seam.exception.exceptions`
Internal component for exception handling.

`org.jboss.seam.transaction.transaction`
API for controlling transactions and abstracting the underlying transaction management implementation behind a JTA-compatible interface.

`org.jboss.seam.faces.safeActions`
: Decides if an action expression in an incoming URL is safe. This is done by checking that the action expression exists in the view.

32.11. Miscellaneous components

These components don't fit into

`org.jboss.seam.async.dispatcher`
: Dispatcher stateless session bean for asynchronous methods.

`org.jboss.seam.core.image`
: Image manipulation and interrogation.

`org.jboss.seam.core.pojoCache`
: Manager component for a PojoCache instance.

`org.jboss.seam.core.uiComponent`
: Manages a map of UIComponents keyed by component id.

32.12. 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.persistence.ManagedPersistenceContext">
   <property name="persistenceUnitJndiName">java:/comp/emf/bookingPersistence</property>
</component>

<component name="userDatabase"
        class="org.jboss.seam.persistence.ManagedPersistenceContext">
   <property name="persistenceUnitJndiName">java:/comp/emf/userPersistence</property>
</component>
```

The Seam component names are `bookingDatabase` and `userDatabase`.

<entityManager>, `org.jboss.seam.persistence.ManagedPersistenceContext`
: Le composant de gestion d'un `EntityManager` géré par une étendue conversationnelle avec un contexte de persistence étendue.

  - *<entityManager>*.entityManagerFactory — une expression liant une valeur qui est évalué par une instance de `EntityManagerFactory`.

<*entityManager*>.persistenceUnitJndiName — le nom JNDI de la fabrique d'entity manager, par défaut à java:/<*managedPersistenceContext*>.

<*entityManagerFactory*>, `org.jboss.seam.persistence.EntityManagerFactory`
Gestion d'une `EntityManagerFactory` JPA. Le plus utilisé quand on utilise JPA sans environement supportant EJB 3.0.

- `entityManagerFactory.persistenceUnitName` — le nom de l'unité de persistance.

Voir l'API JavaDoc pour plus de propriétés de configuration.

<*session*>, `org.jboss.seam.persistence.ManagedSession`
Le composant de gestion d'un `EntityManager` géré par Hibernate d'étendue conversationnel.

- <*session*>.sessionFactory — une expression liée à une valeur à évaluer à une instance de `SessionFactory`.

  <*session*>.sessionFactoryJndiName — le nom JNDI d'une fabrique de session, par défaut à java:/<*managedSession*>.

<*sessionFactory*>, `org.jboss.seam.persistence.HibernateSessionFactory`
Gestion d'une `SessionFactory` d'Hibernate.

- `<sessionFactory>.cfgResourceName` — le chemin vers le fichier de configuration. Par défaut à `hibernate.cfg.xml`.

Voir l'API JavaDoc pour plus de propriétés de configuration.

<*managedQueueSender*>, `org.jboss.seam.jms.ManagedQueueSender`
Le composant de gestion d'une `QueueSender` de JMS géré d'étendue évènementiel.

- <*managedQueueSender*>.queueJndiName — le nom JNDI de la queue JMS.

<*managedTopicPublisher*>, `org.jboss.seam.jms.ManagedTopicPublisher`
Le composant de gestion d''un `TopicPublisher` JMS d'étendue évènementiel

- <*managedTopicPublisher*>.topicJndiName — le nom JNDI d'un topic JMS.

<*managedWorkingMemory*>, `org.jboss.seam.drools.ManagedWorkingMemory`
Le composant de gestion d'un `WorkingMemory` Drools géré par une étendue conversationnelle.

- <*managedWorkingMemory*>.ruleBase — une expression de valeur qui est évalué à une instance de `RuleBase`.

<*ruleBase*>, `org.jboss.seam.drools.RuleBase`
Le composant de gestion pour une `RuleBase` Drools géré d'étendue application. *Notez que ce n'est pas vraiment à essayer en production, surtout que cela ne permet pas l'installation dynamique de nouvelles règles.*

- <*ruleBase*>.ruleFiles — une liste de fichier contenant les règles Drools.

  <*ruleBase*>.dslFile — une définition de DSL de Drools.

<*entityHome*>, `org.jboss.seam.framework.EntityHome`

<*hibernateEntityHome*>, `org.jboss.seam.framework.HibernateEntityHome`

<*entityQuery*>, `org.jboss.seam.framework.EntityQuery`

<*hibernateEntityQuery*>, `org.jboss.seam.framework.HibernateEntityQuery`

Les contrôles JSF de Seam

Seam inclus de nombreux contrôles JSF qui sont très utile pour travailler avec Seam. Leur but est d'essayer de compléter les contrôles JSF livrés, et les contrôles provenant de bibliothèques tierces partites. Nous recommandons les bibliothèques de balises Ajax4JSF et ADF faces (maintenant Trinidad) à utiliser avec Seam. Nous ne recommandons pas l'utilisation de la bibliothèque de balises Tomahawk.

33.1. Les balises

Pour utiliser ces balises, définissez l'espace de nommage "s" dans votre page comme ci-dessous (seulement pour les facelets):

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:s="http://jboss.com/products/seam/taglib"
>
```

L'exemple ui (NdT: user interface=interface utilisateur) montre l'utilisation de nombreuses balises.

33.1.1. Les contrôle de navigation

33.1.1.1. `<s:button>`

Description

Un bouton qui permet l'invication d'un action qui prends le control sur la propagation de la conversation.*Ne soumet pas le formulaire.*

Attribus

- `value` — le label.
- `action` — une méthode en liaison qui spécifie l'écouteur d'action.
- `view` — l'identifiant de vue JSF à lier avec.
- `fragment` — l'identifier de fragment à lier avec.
- `disabled` — ce lien est il désactivé?
- `propagation` — détermine le style de propagation de la conversation: `begin`, `join`, `nested`, `none`, `end` ou `endRoot`.
- `pageflow` — une définition d'enchainement de page pour démarrer. (Ceci est seulement utile quand `propagation="begin"` ou `propagation="join"` sont utilisés).

- `includePageParams` — when set to false, page parameters defined in `pages.xml` will be excluded from rendering.

Utilisation

```
<s:button id="cancel"
        value="Cancel"
        action="#{hotelBooking.cancel}"/>
```

Viys oiuvez spécifier à la fois la `view` et l'`action` sur un `<s:link />`. Dans ce cas, l'action ne sera appelée qu'une fois quand la rediction vers la vue spécifique interviendra.

L'utilisation d'écouteurs d'action (incluant l'écouteur d'action par défaut de JSF) n'est pas supporté avec `<s:button />`.

33.1.1.2. `<s:conversationId>`

Description

Ajoute l'identifiant de conversation au lien JSF ou au bouton (autrement dit. `<h:commandLink />` , `<s:button />`).

Attribus

Aucun

33.1.1.3. `<s:taskId>`

Description

Ajoute l'identifiant de tâche à un lien sortant (ou à un controle JSF similaire) quand la tâche est disponible via `#{task}`.

Attribus

Aucun.

33.1.1.4. `<s:link>`

Description

Un lien qui permet l'invocation d'une action avec un contrôle sur la propagation de la conversation. *Ne soumet pas le formulaire.*

L'utilisation d'écouteurs d'action (incluant l'écouteur d'action par défaut de JSF) n'est pas supporté avec `<s:link />`.

Attribus

- `value` — le label.

- `action` — une méthode en liaison qui spécifie l'écouteur d'action.
- `view` — l'identifiant de vue JSF à lier avec.
- `fragment` — l'identifier de fragment à lier avec.
- `disabled` — ce lien est il désactivé?
- `propagation` — détermine le style de propagation de la conversation: `begin`, `join`, `nested`, `none`, `end` ou `endRoot`.
- `pageflow` — un définition d'enchainement de page commence. (C'est seulement utile avec l'utilisation `propagation="begin"` ou `propagation="join"`.)
- `includePageParams` — when set to false, page parameters defined in `pages.xml` will be excluded from rendering.

Utilisation

```
<s:link id="register" view="/register.xhtml"
      value="Register New User"/>
```

Viys oiuvez spécifier à la fois la `view` et l'`action` sur un `<s:link />`. Dans ce cas, l'action ne sera appelée qu'une fois quand la rediction vers la vue spécifique interviendra.

33.1.1.5. `<s:conversationPropagation>`

Description

Personnalise la propagation de la conversation d'un lien de commande ou d'un bouton (ou d'un control JSF similaire). *seulement avec les Facelets.*

Attribus

- `type` — détermine le style de la propagation de la conversation : `begin`, `join`, `nested`, `none`, `end` ou `endRoot`.
- `pageflow` — un définition d'enchainement de page commence. (C'est seulement utile avec l'utilisation `propagation="begin"` ou `propagation="join"`.)

Utilisation

```
<h:commandButton value="Apply" action="#{personHome.update}">
  <s:conversationPropagation type="join" />
</h:commandButton
>
```

33.1.1.6. `<s:defaultAction>`

Description

Spécifie l'action par défaut à exécuter quand le formulaire est soumis en utilisant la touche entrée.

Actuellement, vous ne pouvez l'embarquer qu'au sein des boutons (autrement dit. `<h:commandButton />`, `<a:commandButton />` ou `<tr:commandButton />`).

Vous devez spécifier un identifiant pour la source de l'action. Vous pouvez seulement avoir une action par défaut par formulaire.

Attribus

Aucun.

Utilisation

```
<h:commandButton id="foo" value="Foo" action="#{manager.foo}">
  <s:defaultAction />
</h:commandButton
>
```

33.1.2. Les convertisseurs et les validateurs

33.1.2.1. `<s:convertDateTime>`

Description

Réalise les conversation de date et d'horaire dans le fuseau horaire de Seam.

Attribus

Aucun.

Utilisation

```
<h:outputText value="#{item.orderDate}">
  <s:convertDateTime type="both" dateStyle="full"/>
</h:outputText
>
```

33.1.2.2. `<s:convertEntity>`

Description

Affecte un convertisseur d'entité au composant courrant. C'est vraiment utile pour un bouton radio ou un controle de type combobox.

Le convertisseur fonctionne avec des entités gérées - qu'elle soit simple ou composée. Le convertisseur devrait être capable de trouver les éléments déclaré dans le controle de JSF dans la soumission du formulaire, autrement vous allez recevoir une erreur de validation.

Attribus

Aucun.

La configuration.

Vous devez utiliser *des transactions géré par Seam* (voir *Section 9.2, « Seam managed transactions »*) avec le `<s:convertEntity />`.

Si votre *Contexte du Managed Persistence* n'appelle pas `entityManager`, alors vous allez avoir besoin de le définir dans components.xml:

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

  <ui:jpa-entity-loader entity-manager="#{em}" />
```

Si votre *Session du Managed Persistence* n'appelle pas `entityManager`, alors vous allez avoir besoin de le définir dans components.xml:

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

  <ui:hibernate-entity-loader />
```

Si votre *Session du Managed Persistence* n'appelle pas `session`, alors vous allez avoir besoin de le définir dans components.xml:

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

  <ui:hibernate-entity-loader session="#{hibernateSession}" />
```

Si vous voulez utiliser plus d'un gestionnaire d'entité avec le convertisseur d'entité, vous devez créer une copie du convertisseur d'entité pour chaque gestionnaire d'entité dans `components.xml`

- notez comment le convertisseur d'entité va déléguer au chargeur de l'entité la réalisation des opérations de persistance:

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

    <ui:entity-converter name="standardEntityConverter" entity-loader="#{standardEntityLoader}"
 />

                                    <ui:jpa-entity-loader            name="standardEntityLoader"            entity-
manager="#{standardEntityManager}" />

   <ui:entity-converter name="restrictedEntityConverter" entity-loader="#{restrictedEntityLoader}"
 />

                                    <ui:jpa-entity-loader            name="restrictedEntityLoader"            entity-
manager="#{restrictedEntityManager}" />
```

```
<h:selectOneMenu value="#{person.continent}">
   <s:selectItems value="#{continents.resultList}" var="continent"
                  label="#{continent.name}" />
    <f:converter converterId="standardEntityConverter" />
</h:selectOneMenu
>
```

Utilisation

```
<h:selectOneMenu value="#{person.continent}" required="true">
   <s:selectItems value="#{continents.resultList}" var="continent"
                  label="#{continent.name}"
                  noSelectionLabel="Please Select..."/>
   <s:convertEntity />
</h:selectOneMenu
>
```

33.1.2.3. `<s:convertEnum>`

Description

Affecte un convertisseur d'enumérateur au composant courrant. C'est en premier lieu utile pour les boutons radio et les controles de type combobox.

Attribus

Aucun.

Utilisation

```
<h:selectOneMenu value="#{person.honorific}">
  <s:selectItems value="#{honorifics}" var="honorific"
                 label="#{honorific.label}"
                 noSelectionLabel="Please select" />
  <s:convertEnum />
</h:selectOneMenu
>
```

33.1.2.4. `<s:convertAtomicBoolean>`

Description

`javax.faces.convert.Converter` pour `java.util.concurrent.atomic.AtomicBoolean`.

Attribus

Aucun.

Utilisation

```
<h:outputText value="#{item.valid}">
  <s:convertAtomicBoolean />
</h:outputText
>
```

33.1.2.5. `<s:convertAtomicInteger>`

Description

`javax.faces.convert.Converter` pour `java.util.concurrent.atomic.AtomicInteger`.

Attribus

Aucun.

Utilisation

```
<h:outputText value="#{item.id}">
  <s:convertAtomicInteger />
</h:outputText
```

```
>
```

33.1.2.6. `<s:convertAtomicLong>`

Description

`javax.faces.convert.Converter` pour `java.util.concurrent.atomic.AtomicLong`.

Attribus

Aucun.

Utilisation

```
<h:outputText value="#{item.id}">
  <s:convertAtomicLong />
</h:outputText
>
```

33.1.2.7. `<s:validateEquality>`

Description

La balise à insérer dans un controle de saisie pour valider que la valeur parente est équale (ou différente) à la valeur du controle référencé.

Attribus

- `for` — L'identifiant du controle à comparer pour validation.
- `message` — Le message à afficher en cas d'echec.
- `required` — Faux va désactiver la validation d'une valeur pour toutes les champs de saisies.
- `messageId` — L'identifiant de message à afficher en cas d'echec.
- `operator` — Quel opérateur à utiliser avec la comparaison des valeurs. Les opérateurs valident sont :
  - `equal` — Valide que value.equals(forValue)
  - `not_equal` — Valide que !value.equals(forValue)
  - `greater` — Valide que ((Comparable)value).compareTo(forValue) > 0
  - `greater_or_equal` — Valide que ((Comparable)value).compareTo(forValue) >= 0
  - `less` — Valide que ((Comparable)value).compareTo(forValue) <0
  - `less_or_equal` — Valide que ((Comparable)value).compareTo(forValue) <= 0

Utilisation

```
<h:inputText id="name" value="#{bean.name}"/>
<h:inputText id="nameVerification" >
  <s:validateEquality for="name" />
</h:inputText
>
```

33.1.2.8. `<s:validate>`

Description

Un controle sans partie visible, qui valide un champ de saisi JSF par rapport à une propriété liée en utilisant un Validator d'Hibernate.

Attribus

Aucun.

Utilisation

```
<h:inputText id="userName" required="true"
             value="#{customer.userName}">
  <s:validate />
</h:inputText>
<h:message for="userName" styleClass="error" />
```

33.1.2.9. `<s:validateAll>`

Description

Un controle sans partie visible, qui valide tous les champs de saisie enfants de JSF par rapport à une propriété liée en utilisant un Validator d'Hibernate.

Attribus

Aucun.

Utilisation

```
<s:validateAll>
  <div class="entry">
    <h:outputLabel for="username"
>Username:</h:outputLabel>
    <h:inputText id="username" value="#{user.username}"
```

```
                required="true"/>
    <h:message for="username" styleClass="error" />
  </div>
  <div class="entry">
    <h:outputLabel for="password"
>Password:</h:outputLabel>
    <h:inputSecret id="password" value="#{user.password}"
                  required="true"/>
    <h:message for="password" styleClass="error" />
  </div>
  <div class="entry">
    <h:outputLabel for="verify"
>Verify Password:</h:outputLabel>
    <h:inputSecret id="verify" value="#{register.verify}"
                  required="true"/>
    <h:message for="verify" styleClass="error" />
  </div>
</s:validateAll
>
```

33.1.3. Le formatage

33.1.3.1. `<s:decorate>`

Description

"Décore" un champs de saisie JSF quand la validation échoue ou quand `required="true"` est définie.

Attribus

- `template` — le modèle de facelets à utiliser pour décorer le composant
- `enclose` — si vrai, le modèle utilisé pour décorer le champs de saisie est encadré par l'élément spécifié par l'attribut "element". Par défaut, c'est un élément div.
- `element` — l'élément qui englobe ce modèle utilisé pour décorer le champs de sasie. Par défaut, le modèl esr inclus dans un élément div.

`#{invalid}` et `#{required}` sont disponible dans `s:decorate`; `#{required}` est évalué à `true` si vous avez défini que le composant de saisie est décoré comme prévus , et `#{invalid}` évalué à `true` si une erreur de validation apparait.

Utilisation

```
<s:decorate template="edit.xhtml">
```

```
  <ui:define name="label"
>Country:</ui:define>
    <h:inputText value="#{location.country}" required="true"/>
  </s:decorate
>
```

```
<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"
  xmlns:s="http://jboss.com/products/seam/taglib">

  <div
>

    <s:label styleClass="#{invalid?'error':''}">
      <ui:insert name="label"/>
      <s:span styleClass="required" rendered="#{required}"
>*</s:span>
    </s:label>

    <span class="#{invalid?'error':''}">
      <s:validateAll>
        <ui:insert/>
      </s:validateAll>
    </span>

    <s:message styleClass="error"/>

  </div
>

</ui:composition
>
```

33.1.3.2. `<s:div>`

Description

Rends un`<div>` HTML.

Attribus

Aucun.

Utilisation

```
<s:div rendered="#{selectedMember == null}">
  Sorry, but this member does not exist.
</s:div
>
```

33.1.3.3. `<s:span>`

Description

Rend un `<span>` HTML.

Attribus

- `title` — Le titre d'un span.

Utilisation

```
<s:span styleClass="required" rendered="#{required}" title="Small tooltip"
>*</s:span
>
```

33.1.3.4. `<s:fragment>`

Description

Un composant non-rendu utile pour activer/désactiver le rendu de ces enfants.

Attribus

Aucun.

Utilisation

```
<s:fragment rendered="#{auction.highBidder ne null}">
  Current bid:
</s:fragment
>
```

33.1.3.5. `<s:label>`

Description

"Décore" un champs de saisie JSF avec ce label. Ce label sera placé dans un `<label>`de la balise HTM, et il sera associé avec le composant de saisie JSF le plus proche. C'est souvent utilisé avec `<s:decorate>`.

Attribus

- `style` — Le style du controle
- `styleClass` — La classe du style du controle

Utilisation

```
<s:label styleClass="label">
  Country:
</s:label>
<h:inputText value="#{location.country}" required="true"/>
```

33.1.3.6. `<s:message>`

Description

"Décore" un champs de saisie JSF avec le mesage d'erreur de la validation.

Attribus

Aucun.

Utilisation

```
<f:facet name="afterInvalidField">
  <s:span>
    &amp;#160;Error:&amp;#160;
    <s:message/>
  </s:span>
</f:facet
>
```

33.1.4. Le texte de Seam

33.1.4.1. `<s:validateFormattedText>`

Description

La vérification que les valeurs soumises sont du Texte de Seam valide

Attribus

Aucun.

33.1.4.2. `<s:formattedText>`

Description

Affiche un *Seam Text*, un texte avec un formatage riche contenant des balises très utiles pour les blogs, les wikis and toute application qui pourait avoir besoin de textes avec un formatage riche. Allez voir le chapitre sur Seam Text pour voir toutes les utilisations.

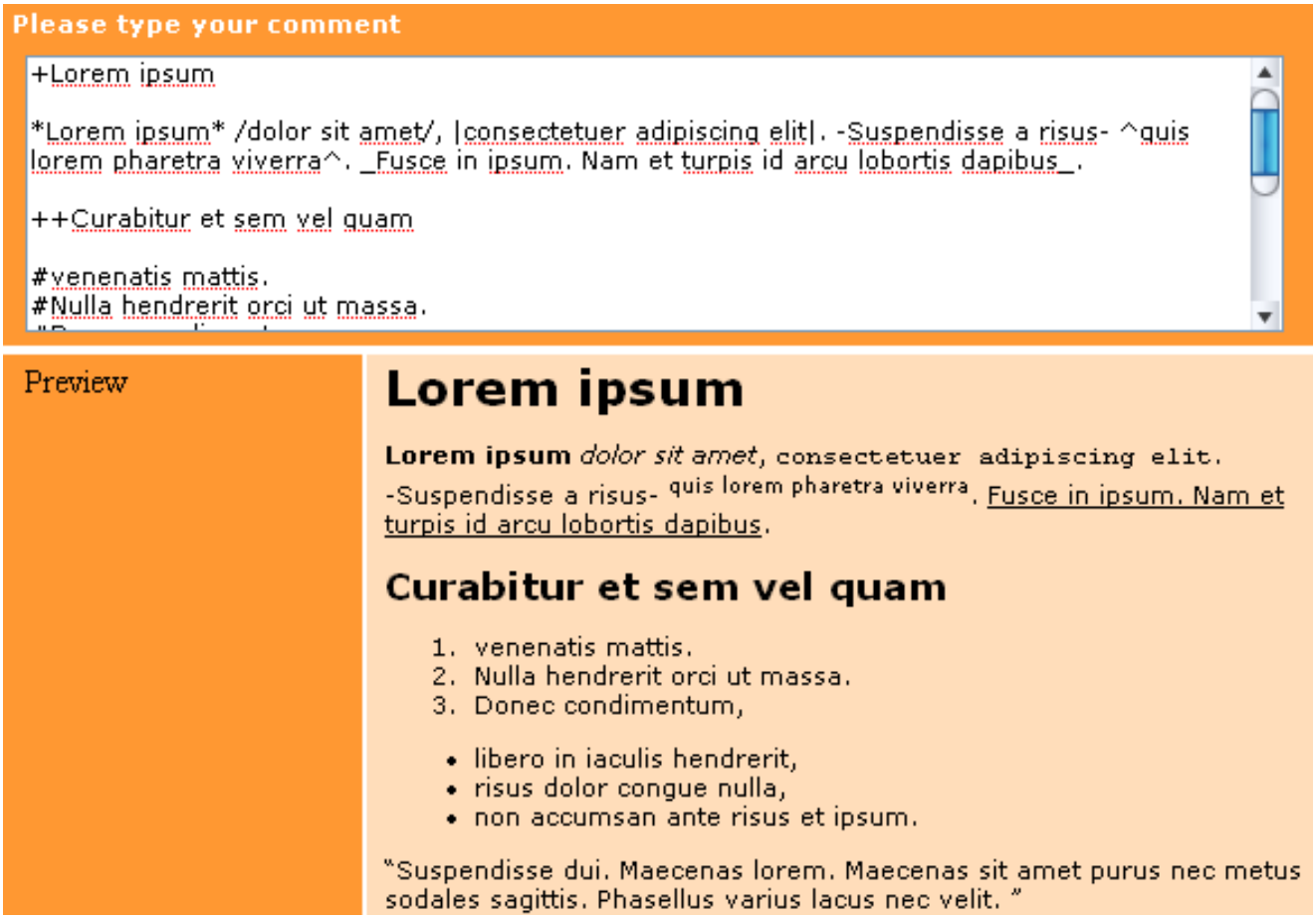
Attribus

- `value` — un exprèssion EL spécifiant que le texte avec un formatage riche à rendre.

Utilisation

```
<s:formattedText value="#{blog.text}"/>
```

L'exemple



33.1.5. Le support de formulaire

33.1.5.1. `<s:token>`

Description

Produit un jeton aléatoire qui doit être inséré dans un champs caché de formulaire pour aider à sécuriser l'envoie de 'un formulaire contre les attaques "cross-site request forgery (XSRF)". Notez que le navigateur doit avoir les cookies actifs pour permettre de soumettre le formulaire qui est inlus avec ce composant.

Attribus

- `requireSession` — indique que l'identifiant de session devrait être inclus dans la signature du formulaire, améliorant la liaison entre le jeton et la session. Cette valeur peut être défini à faux si le mode "construire avant de restorer" des Facelets est activé (par défaut en JSF 2.0). (par défaut: false)
- `enableCookieNotice` — indiqye que la vérification en JavaScript peut être insérée dans la page pour vérifier que les cookies sont activés dans le navigateur. Si les cookies ne sont pas activés, affiche une information pour l'utilisateur ques les formulaires postées ne font pas fonctionner. (par déefaut: false)
- `allowMultiplePosts` — indique que cela autorise le même formulaire à être soumis plusieurs fois avec la même signature (aussi longtemps qye la vue ne change pas). C'est un besoin classique dans le formulaire qui réalise des appels Ajax mais ne se ré-affiche pas lui-même ou, en dernière extrémitée, le composant UIToken. TM'approche a préférer est d'avoir le composant UIToken ré-affichant tout appel à Ajax où le composant UIToken devrait être exécuté. (par défaut: false)

Utilisation

```
<h:form>
  <s:token enableCookieNotice="true" requireSession="false"/>
  ...
</h:form
>
```

33.1.5.2. `<s:enumItem>`

Description

Création d'un `SelectItem` depuis une valeur énumérée.

Attribus

- `enumValue` — la représentation en texte d'une valeur énumérée.
- `label` — le label à utiliser pendant le rendu de `SelectItem`.

Utilisation

```
<h:selectOneRadio id="radioList"
                  layout="lineDirection"
                  value="#{newPayment.paymentFrequency}">
  <s:convertEnum />
  <s:enumItem enumValue="ONCE"         label="Only Once" />
  <s:enumItem enumValue="EVERY_MINUTE" label="Every Minute" />
  <s:enumItem enumValue="HOURLY"       label="Every Hour" />
  <s:enumItem enumValue="DAILY"        label="Every Day" />
  <s:enumItem enumValue="WEEKLY"       label="Every Week" />
</h:selectOneRadio
>
```

33.1.5.3. `<s:selectItems>`

Description

Créer un `List<SelectItem>` depuis un type suivant List, Set, DataModel ou Array.

Attribus

- `value` — une expression EL spécifiant que les données qui encadrent le `List<SelectItem>`
- `var`— défini le nom de la variable locale qui conserve l'objet courrant pendant l'itération
- `label` — le label à utiliser pendant le rendue de `SelectItem`. Peut référencer la variable `var`.
- `itemValue` — La valeur à retourner au serveur si l'option est sélectionné. Optionnellement, par défaut l'objet `var` est utilisé. Peut référencer la variable `var`.
- `disabled` — si vrai le `SelectItem` sera rendu désactivé. Peut référencer la variable `var`.
- `noSelectionLabel` — spécifie le label (optionnel) à placer en haut de la liste (si `required="true"` est aussi spécifié alors la sélection de la valeur déclenchera une erreur de validation).
- `hideNoSelectionLabel` — si vrai, le `noSelectionLabel` sera masqué quand une valeur est sélectionné

Utilisation

```
<h:selectOneMenu value="#{person.age}"
```

```
            converter="ageConverter">
  <s:selectItems value="#{ages}" var="age" label="#{age}" />
</h:selectOneMenu
>
```

33.1.5.4. `<s:fileUpload>`

Description

Rend un control de téléchargement de fichier. Ce control doit être utilisé dans un formulaire avec un type d'encodage `multipart/form-data`, autrement dit:

```
<h:form enctype="multipart/form-data"
>
```

Pour les requêtes multipart, le filtre de servlet Multipart de Seam doit aussi être configuré dans `web.xml`:

```
<filter>
  <filter-name
>Seam Filter</filter-name>
  <filter-class
>org.jboss.seam.servlet.SeamFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name
>Seam Filter</filter-name>
  <url-pattern
>/*</url-pattern>
</filter-mapping
>
```

La configuration.

Les options de configurations suivant pour les requêtes multipart peuvent être configuré dans components.xml:

- `createTempFiles` — si cette option est définie à true, les fichiers téléchargé seront stocké dans un fichier temporaire au lieu d'être en mémoire.
- `maxRequestSize` — la taille maximale de la requête de téléchargement de fichier, en octets.

Voici un exemple:

```
<component class="org.jboss.seam.web.MultipartFilter">
  <property name="createTempFiles"
>true</property>
  <property name="maxRequestSize"
>1000000</property>
</component
>
```

Attribus

- `data` — ette valeur correspondante reçoit les données binaire du fichier. Le champs de réception doit êtree déclaré comme un `byte[]` ou un `InputStream` (nécéssaire).
- `contentType` — cette valeur correspondante recoit le type du contenu de fichier (optionnel).
- `fileName` — cette valeur correspondate reçois le nom du fichier (optionnel).
- `fileSize` — cette valeur correspondante reçois la taille du fichier (optionnelle).
- `accept` — une liste séparée de virgules de types de contenu, peut ne pas être supporté par le navigateur. Par exemple `"images/png,images/jpg"`, `"images/*"`.
- `style` — Le style du controle
- `styleClass` — La classe du style du controle

Utilisation

```
<s:fileUpload id="picture" data="#{register.picture}"
             accept="image/png"
             contentType="#{register.pictureContentType}" />
```

33.1.6. Divers

33.1.6.1. `<s:cache>`

Description

Met en cache le fragment de page rendue en utilisant JBoss Cache. Notez que `<s:cache>` utilise actuellement l'instance de JBoss Cache géré par le composant livré `pojoCache`.

Attribus

- `key` — la clef vers le contenu rendu mit en cache, souvent une expression de valeur. Par exemple, si nous mettons en cache un fragment de page qui affiche un document, nous devrions utiliser `key="Document-#{document.id}"`.
- `enabled` — une expression de valeur qui détermine si le couche devrait être utilisé.
- `region` — un noeud JBoss Cache à utiliser (differents noeuds peuvent avoir des politiques d'expirations différentes).

Utilisation

```
<s:cache key="entry-#{blogEntry.id}" region="pageFragments">
  <div class="blogEntry">
    <h3
>#{blogEntry.title}</h3>
    <div>
      <s:formattedText 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>
</s:cache
>
```

33.1.6.2. `<s:resource>`

Description

Ue balise qui fonctionne comme un fournissuer de téléchargement de fichier. Elle doit être seule dans la page JSF. Pour pouvoir utiliser ce control, web.xml doit être configuré comme ceci.

La configuration.

```
<servlet>
   <servlet-name
>Document Store Servlet</servlet-name>
   <servlet-class
>org.jboss.seam.document.DocumentStoreServlet</servlet-class>
</servlet>
```

```
<servlet-mapping>
  <servlet-name
>Document Store Servlet</servlet-name>
  <url-pattern
>/seam/docstore/*</url-pattern>
</servlet-mapping>
```

Attribus

- `data` — Les données qui devrait être téléchargées. Peuvent être un java.util.File, un InputStream ou un tableau d'octet.
- `fileName` — Nom de fichier qui doit être servis
- `contentType` — type contenu du fichier à télécharger
- `disposition` — la disposition à utiliser. Par défaut c'est inline

Utilisation

Voici un exemple de comment utiliser ces balises:

```
<s:resource xmlns="http://www.w3.org/1999/xhtml"
      xmlns:s="http://jboss.com/products/seam/taglib"
      data="#{resources.data}"
      contentType="#{resources.contentType}"
      fileName="#{resources.fileName}" />
```

Le bean dénommé `resources` est le bean de renfort qui donne des paramètres de requête et délivre un fichier spécifique, voir `s:download`.

33.1.6.3. `<s:download>`

Description

Construit un lien RESTful vers une `<s:resource>`. Encapsule `f:param` pour construire l'url.

- `src` — Fichier de ressource distribuant les fichiers.

Attribus

```
<s:download src="/resources.xhtml">
   <f:param name="fileId" value="#{someBean.downloadableFileId}"/>
</s:download
```

```
>
```

Sera produira quelque chose comme: `http://localhost/resources.seam?fileId=1`

33.1.6.4. `<s:graphicImage>`

Description

Une extension de `<h:graphicImage>` qui permet à l'image d'être cré dans un Seam Component; d'autres transformation peuvent être appliqué à l'image.

Tous les attributs de `<h:graphicImage>` sont supportés, à l'identique:

Attribus

- `value` — image to display. Can be a path `String` (loaded from the classpath), a `byte[]`, a `java.io.File`, a `java.io.InputStream` or a `java.net.URL`. Currently supported image formats are `image/png`, `image/jpeg`, `image/gif` and `image/bmp`.
- `fileName` — s'il n'est pas indiqué l'image servie aura un nom de fichier générée. Si vous voulez nommé votre fichier, vous devriez l'indiquer ici. Ce nom devrait être unique

Les transformations

Pour appliquer une transormation à une image, vous pouvez inclure une balise spécifiant la transformation à appliquer. Seam supporte actuellement ces transformations:

`<s:transformImageSize>`

- `width` — nouvelle largeur de l'image
- `height` —nouvelle hauteur de l'image
- `maintainRatio` — Si `true`, et *one* de `width`/`height` sont spécifiés, l'image sera retaillée, la dimension non-identiquée sera calculé pour maintenir le ratio.
- `factor` — met à l'echelle l'image du facteur indiqué

`<s:transformImageBlur>`

- `radius` — réaliser un flou de convolution avec un rayon donné

`<s:transformImageType>`

- `contentType` — altère le type de l'image à `image/jpeg` ou à `image/png`

Il est facile de créer votre propre transformation - créez un `UIComponent` qui implémente `org.jboss.seam.ui.graphicImage.ImageTransform`. A l'intérieur d'une méthode

`applyTransform()` utilisant `image.getBufferedImage()` pour obtenir l'image originelle et `image.setBufferedImage()` pour affecter votre image transformée. Les transformations sont appliqué dans l'odre spécifié dans la vue.

Utilisation

```
<s:graphicImage rendered="#{auction.image ne null}"
                value="#{auction.image.data}">
  <s:transformImageSize width="200" maintainRatio="true"/>
</s:graphicImage
>
```

33.1.6.5. `<s:remote>`

Description

Génération des squelettes Javascript nécéssaire pour l'utilisation de Seam Remoting.

Attribus

- `include` — une liste séparée par des virrgules de noms de composants (ou de nomes de classe pleinement qualifiés) pour lesquels il faut générer les squelettes de Seam Remoting Javascript stubs. Voir *Chapitre 25, Remoting* pour plus de détails.

Utilisation

```
<s:remote include="customerAction,accountAction,com.acme.MyBean"/>
```

33.2. Les annotations

Seam fourni aussi des annotations pour vous permettre d'utiliser les composants de Seam comme un convertisseur de JSF et les validateurs:

`@Converter`

```
@Name("itemConverter")
@BypassInterceptors
@Converter
public class ItemConverter implements Converter {

   @Transactional
   public Object getAsObject(FacesContext context, UIComponent cmp, String value) {
```

```
                                    EntityManager entityManager = (EntityManager)
 Component.getInstance("entityManager");
      entityManager.joinTransaction();
      // Do the conversion
    }

    public String getAsString(FacesContext context, UIComponent cmp, Object value) {
      // Do the conversion
    }

  }
```

```
<h:inputText value="#{shop.item}" converter="itemConverter" />
```

Enregistre le composant de Seam comme un convertisseur JSF. Visible ici un convertisseur qui est capable d'accéder au JPA EntityManager dans une transaction JTA, pendant la conversion d'une valeur vers sa représentation comme objet.

`@Validator`

```
@Name("itemValidator")
@BypassInterceptors
@org.jboss.seam.annotations.faces.Validator
public class ItemValidator implements javax.faces.validator.Validator {

  public void validate(FacesContext context, UIComponent cmp, Object value)
      throws ValidatorException {
    ItemController ItemController = (ItemController) Component.getInstance("itemController");
    boolean valid = itemController.validate(value);
    if (!valid) {
      throw ValidatorException("Invalid value " + value);
    }
  }
}
```

```
<h:inputText value="#{shop.item}" validator="itemValidator" />
```

Enregistre le composant de Seam comme un validateur JSF. Visible ici une validateur qui injecte un autre composant de Seam, le composant injecté est utilisé pour valider la valeur.

JBoss EL

Seam fourni un extention pour le Langage d'Expression Unifiée (EL) standardisé appelé JBoss EL. JBoss EL fournit un nombre d'amélioration qui augmente l'expressivité et la puissance des expressions EL.

34.1. Les expressions paramétisées

Le EL standard ne vous permet d'utiliser une méthode où l'utilisateur définir les paramètres — bien sur, les méthodes d'écoute de JSF (par exemple, un `valueChangeListener`) prends les paramètres fournis par JSF.

JBoss EL retire cette restriction. Par exemple:

```
<h:commandButton action="#{hotelBooking.bookHotel(hotel)}" value="Book Hotel"/>
```

```
@Name("hotelBooking")
public class HotelBooking {

  public String bookHotel(Hotel hotel) {
    // Book the hotel
  }
}
```

34.1.1. Utilisation

Appelez juste cette méthode depuis Java, les paramètres sont entourés par des parenthèses, et séparés par des virgules:

```
<h:commandButton action="#{hotelBooking.bookHotel(hotel, user)}" value="Book Hotel"/>
```

Les paramètres `hotel` et `user` seront évalués comme des expressions de valeur et passés à la méthode du composant `bookHotel()`.

Tout expression de valeur peut être utilisé comme un paramètre:

```
<h:commandButton
  action="#{hotelBooking.bookHotel(hotel.id, user.username)}"
  value="Book Hotel"/>
```

Il est important de bien comprends comme cette extension de EL fonctionne. Quand la page est rendu, le paramètres *names* est stocké (par exemple, `hotel.id` et `user.username`) et évalués (comme des expressions de valeur) quand la page est soumise. Vous ne pouvez passer des objets comme des paramètres!

Vous devez vous assurer que les paramètres sont disponibles pas seulement quand la page est rendue mais aussi quand elle est soumis. Si l'argument ne peut être résolu quand la page est soumise la méthode d'action sera appelée avec des arguments `null`!

Vous pouvez aussi passer des chaines de caractères en utilisant les apostrophes:

```
<h:commandLink action="#{printer.println('Hello world!')}" value="Hello"/>
```

EL unifié supporte aussi les expressions de valeur, utilisés pour trouver un champs d'un bean arrière. Les expressions de valeurs utilise les conventions de nommage des JavaBeans et s'attends à une paire d'assesseurs. Souvent JSF attends une expression de valeur avec seulement une lecture (get) est nécéssaire (par exemple l'attribut `rendered`). Beaucoup d'objets, cependant, n'ont pas les asseceurs de propriétés dénommé de manière approprié ou nécéssite des paramètres.

JBoss EL retire cette restriction qui permet aux valeurs d'êre retrouvé en utilisant la syntaxe des méthodes. Par exemple:

```
<h:outputText value="#{person.name}" rendered="#{person.name.length()
> 5}" />
```

Vous pouvez accéder à la taille de la collection de manière similaire:

```
#{searchResults.size()}
```

En égénral, toute expression de la forme #{obj.property} devrait être indentique à l'expression #{obj.getProperty()}.

Les paramètres sont aussi permis. L'exemple suivant appelle `productsByColorMethod` avec un argument chaine de caractère litérale:

```
#{controller.productsByColor('blue')}
```

34.1.2. Les limitations et les astuces

Avec l'utilisation de JBoss EL, vous devriez garder les points suivants en tête:

- *Incompatibilité avec JSP 2.1* — JBoss EL ne peut actuellement pas être utilisé avec JSP 2.1 car le compilateur rejette les epxressions avec des paramètres. Donc, si vous voulez utiliser cette extension avec JSF 1.2, vous allez devoir utiliser les Facelets. L'extension fonctionne correctement avec JSP 2.0.

- *Utilisation de composants itératif à l'intérieur* — Les composants comme `<c:forEach />` et `<ui:repeat />`itérate au travers d'une List ou d'un tableau, exposant chaque élément de la liste au composant lié. Cela fonctionne bien si vous avez sélectionné une ligne en utilisant `<h:commandButton />` ou `<h:commandLink />`:

```
@Factory("items")
public List<Item
> getItems() {
  return entityManager.createQuery("select ...").getResultList();
}
```

```
<h:dataTable value="#{items}" var="item">
  <h:column>
    <h:commandLink value="Select #{item.name}" action="#{itemSelector.select(item})" />
  </h:column>
</h:dataTable
>
```

  Cependant si vous voulez utiliser `<s:link />` ou `<s:button />` vous *devez* exposer les élements au `DataModel`, et utiliser une `<dataTable />` (ou une équivalence d'un composant défini comme `<rich:dataTable />`). Ni `<s:link />` ou `<s:button />` soumettent le formulaire (et ainsi produisent un lien capable d'être en favori) donc un paramètre "magique" est nécéssaire pour recréer l'élement quand la méthode d'action est appelée. Ce paramètre magique peut seuelement êrte ajouté quand le tableau de donnée liée avec un un `DataModel` est utilisé.

- *L'appel à `MethodExpression` depuis du code Java* — Normallement, quand une `MethodExpression` est crée, les types des paramètres sont passés par JSF. Dans le cas d'une liaison de méthode, JSF suppose qu'il n'y a aucun paramètres à passer. Avec cette extension, nous ne pouvons pas savoir les types de paramètres tant que l'expression n'a pas été évaluée. Ceci a deux conséquences mineures:

  - Quand vous invoquez une `MethodExpression` dans du code Java, les paramètres que vous passez sont ignorés. Les paramètres définis dans l'expression prendront la priorité.

  - Ordinairement, il est plus sûr it d'appeler `methodExpression.getMethodInfo().getParamTypes()` à chaque fois. Pour une

expression avec des paramètres, vous devez en premier invoquer la `MethodExpression` avant l'apperl de `getParamTypes()`.

Ces deux cas sont extrêmement rare et ne s'applique que quand vous voulez invoquer`MethodExpression` à la main dans du code Java.

34.2. La projection

JBoss EL supporte une syntaxe de projection limitée. Une expression de projection correspond une sous-expression au travers d'expression de multiples valeurs (list, set, etc...). Par exemple, l'expression:

```
#{company.departments}
```

doit retourner une liste de département. Si vous avez seulement besoin d'une liste de noms de département, votre seule option est d'itérer au travers de la liste pour retrouver les valeurs. JBoss EL permet cela avec une expression de projection:

```
#{company.departments.{d|d.name}}
```

La sous-expression est entouré par des accolades. Dans cet exemple, l'expression `d.name` est évalué pour chaque département, en utilisant `d` comme un alias sur l'objet département. Le résultat de cette expression est une liste de valeurs String.

Toute expression valide peut être utilisée comme une expression, ainsi il est parfaitement valide d'écrire ce qui suit, en supposant que vous avez une utilisation de la longeurs de tous les noms de département dans une entreprise:

```
#{company.departments.{d|d.size()}}
```

Les projections peuvent être liée. L'expression suivante retourne les noms de familles de chaque employé dans chaque département:

```
#{company.departments.{d|d.employees.{emp|emp.lastName}}}
```

Les projections liées peuvent être légèrement épineuse, toutefois. L'expression suivante devrait faire comme si elle retourne une liste de tous les employés dans tous les départements:

```
#{company.departments.{d|d.employees}}
```

Cependant, elle retourne actuellement une liste contenant une liste des employés pour chaque département individuellement. Pour combiner les valeurs, il est nécéssaire d'utiliser une expression légèrement plus longue:

```
#{company.departments.{d|d.employees.{e|e}}}
```

Il est important de noter que la syntaxe ne peut être analysé par les Facelets ou par JSP et ainsi ne peut être utilisé dans les fichiers xhtml ou JSP. Nous anticipons que la syntaxe de projection va changer dans les futures versions de JBoss EL.

Mise en cluster et Mise en pause EJB

Merci de noter que ce chapitre est toujours en en cours d'écriture. A manipuler avec précaussion.

Ce chapitre couvre deux sujets discint qui heureusement partage une solution commun dans Seam , la mise en cluster (web) et la mise en pause EJB. Cependant, ils sont regroupés ensemble dans ce manuel de référence. Bien que la performance tende à être aussi regroupé dans cette catégorie, nous la conserverons à part cas le but de ce chapitre est le modèle de programmation et comment il est affecté par l'utilisation des fonctionalités susnommées.

Dans ce chapitre vous allez apprendre comment Seam gère la mise en pause de composant de Seam et des instances entité, comment activer cette fonctionnalité et comment cette fonctionnalité est en liaison avec la mise en cluster. Vous devriez aussi apprendre comment déployer une application Seam dans un cluster et vérifier que la réplication de la session HTTP fonctione correctement. Commençons par un peu de généralités sur la mise en cluster et voyons un exemple de comment vous allez déployer une application Seam dans un cluster JBoss AS.

35.1. Mise en cluster

La mise en cluster (plus formellement la mise en cluster pour le web) permet à une application de s'exécuter sur deux serveurs ou plus en parallèle (autrement dit des noeuds) tout en fournissant une vue uniforme de l'application aux clients. La charge est distrbué au travers des serveurs de façon que si un ou plusieurs de ces serrveurs tombent, l'application reste accéssible via les noeuds survivants. Cette topologie est crucial pour la construction d'une application d'entreprise évolutif et avec une disponibilité qui peut être amméliorer en ajoutant simplement des noeuds. Mais cela amène une question importante. *Qu'arrive t'il à l'état qui était sur le serveur qui est tombé?*

Depuis le jour un, Seam a toujours fourni un support pour les applications avec état s'exécutant sur un cluster. En partant de ce point, vous avez appris que Seam fournissait un gestionnaire d'état de la forme d'étendues additionnelles et en controlant le cycle de vie des composants (d'étendue) avec état. Mais le gestionnaire d'état de Seam va plus loin que la création, le stockage et la destructions des instances. Seam surveillle les modificaitions des composants JavaBean et stocke leurs modification aux moments stratégiques pendant la requète pour ainsi faire que les modifications soient restorées quand la requète bascule dans un noeud secondaire du cluster. Heureusement, la supervision et la replication des composants EJB avec état est déjà géré par le serveur EJB, donc cette fonctionnalité de Seam est d'essayer de mettre le JavaBeans avec état actif avec sa cohorte d'EJB.

Mais attendez, il y a mieux! Seam offre aussi une fonctionnalité merveilleuse et unique pour les applications avec mise en cluster. En plus de la suppervision des composants JavaBeans, Seam s'assure que les instances d'entité gérées (autrement dit les entités Hibernate et JPA) ne deviennent pas détachées pendant la réplication. Seam conserve un enregistrement des entités

qui ont été chargées et les charge automatiquement sur un noeud secondaire. Vous devez, cependant, utiliser le contexte de persistance géré par Seam pour avoir cette fonctionnalité. Plus en détails sur cette fonctionnalité est fournis dans la seconde partie de ce chapitre.

Maintenant que vous comprenez que ces fonctionnalités que Seam offre qui support un environnement mise en cluster, allons voir commennt votre programme est mis en cluster.

35.1.1. La programmation pour la mise en cluster

Toute session- ou composant JavaBean mutable d'étendue conversationnel qui sera utilisé dans un environnement clusturisé doit implémenter l'interface `org.jboss.seam.core.Mutable` de l'API de Seam. Un partie du contrat est que le composant doit maintenir un drapeau sale qui est retourné et réinitialisé par la méthode `clearDirty()`. Seam peut appeler cette méthode pour déterminer s'il est nécéssaire de repliquer le composant. Cela évite d'avoir à utiliser l'API Servlet emcombrante pour ajouter et retirer l'attribut de session à chaque modification sur l'objet.

Vous devez aussi vous assurer que toutes les sessions - et les composants JavaBean d'étendue conversationnel sont sérialisable. De plus, tous les champs d'un composant avec état (EJB ou JavaBean) doit être sérialisable à moins que le champ ne soit marqué comme transient ou mit à null dans une méthode `@PrePassivate` . Vous pouvez restorer la velerur d'un transient ou d'un champs mis à null dans une méthode `@PostActivate`.

Un endroit où les gens sont souvent bloqués est en utilisant `List.subList` pour créer une liste. La liste résultante n'est pas Sérialisable. Donc évitez les situations comme celle ci. Si vous avez une `java.io.NotSerializableException` et ne pouvez localiser le coupable au premier abord, vous pouvez mettre un point d'arrêt sur cette exception, exécutez le serveur d'application en mode débogage et y attacher un debogbeur (comme ave Eclipse) pour voir où la désérialisation s'étouffe.

Note

Merci de noter que la mise en cluster ne fonctionne pas avec les composants deployables à chaud. Mais pour rappel, vous ne devriez jamais utiliser les composants déployables à chaud dans un environnement de non-développement.

35.1.2. Le déploiement d'une application Seam dans un cluster de JBoss AS avec la réplication de session

La procédure mise en avant dans ce tutoriel a été validé avec une appllication seam-gen et avec l'exemple de la réservation d'hotel de Seam

Dans ce tutoriel, je suppose que l'adresse IP du maitre et de l'esclave sont respectivement 192.168.1.2 et 192.168.1.3. J'ai intentionnellement refusé d'utiliser le gestionnaire de charge mod_jk pour rendre plus facile la validation des deux noeuds qui vont répondre aux requêtes et partager les sessions.

Je vais utiliserr la méthode de déploiement pour une ferme dans ces instructions, en pensant que vous pourriez aussi déployer normallement l'application et permettre aux deux serveur de négocier une relation maitre/esclave basé sur l'odre de démarrage.

Note

La mise en cluster de JBoss AS se fait avec le multicasting UDP fournis par jGroups. La configuration SELinux qui est livrée par RHEL/Fedora bloque ces paquets par défaut. Vous devez autoriser leur passage en modificaiton les règles des iptables (en root). Les commandes suivantes s'appliquent à l'IP adresse suivante 192.168.1.x.

```
/sbin/iptables -I RH-Firewall-1-INPUT 5 -p udp -d 224.0.0.0/4 -j ACCEPT
/sbin/iptables -I RH-Firewall-1-INPUT 9 -p udp -s 192.168.1.0/24 -j ACCEPT
/sbin/iptables -I RH-Firewall-1-INPUT 10 -p tcp -s 192.168.1.0/24 -j ACCEPT
/etc/init.d/iptables save
```

Des informations détaillés peuvent être trouvées sur *cette page* [http://www.jboss.org/community/docs/DOC-11935] sur le Wiki de JBoss.

- La création de deux instances de JBoss AS (extraction du fichier zip simplement à faire deux fois)
- Deploiement du pilote JDBC dans le serveur /all/lib sur les deux instances en cas de non-utilisation de HSQLDB
- Ajout d'un `<distributable/>` comme premier élément fils dans WEB-INF/web.xml
- Définir la propriété `distributable` de `org.jboss.seam.core.init` à vrai pour activer le ManagedEntityInterceptor (autrement dit, `<core:init distributable="true"/>`)
- Assurrer vous d'avoir deux adresses IP disponible (deux ordinateurs, deux cartes réseau ou deux adresses IP liées sur le même interface). Je pars du principe que les deux adresses IP sont 192.168.1.2 et 192.168.1.3
- Démarrer l'instance maitre de JBoss AS sur la première IP.

```
./bin/run.sh -c all -b 192.168.1.2
```

  Le fichier de log doit montrer qu'il y a 1 membre du cluster et 0 autre membre.

- Vérifieez que le dossier du serveur /all/farm est vide dans l'instance esclave de JBoss AS.

- Démarrez l'instance esclave de JBoss AS sur la seconde IP.

```
./bin/run.sh -c all -b 192.168.1.3
```

  Le log devrait afficher qu'il y a 2 membres du cluster et 1 autre membre. Il devrait aussi montrer que l'état a été récupéré depuis le maitre.

- Déployer le -ds.xml dans le serveur /all.farm sur l'instance maitre

  Dans le log du maitre vous devriez voir l'acquitement du déploiement. Dans le log de l'esclave vous devriez voir un message correspondant à l'acquitement du déploiement vers l'esclave.

- Déployer l'application dans le dossier du server /all/farm

  Dans le log du maitre vous devriez voir l'acquitement du déploiement. Dans le log de l'esclave vous devriez voir un message correspondant à l'acuitement de l'esclave. Notez que vou devriez à avoir à attendre 3 minutes pour que le déploiement de l'archive ai été transféré.

Votre application est maintenant en train de s'exécuter avec la réplication de la session HTTP! Mais , bien sûr, vous voudriez valider que la mise en cluster fonctionne actuellement.

35.1.3. La validation de service distributée d'une application s'exécutant dans un cluster de JBoss AS

C'est super bien de voir l'application réussir à démarrer sur deux serveurs JBoss AS différent, mais voir c'est croire. Vous aimeriez vouloir valider que les deux instances échangent les sessions HTTP pour permettre à l'esclave de prendre le relais quand l'instance maitre est arrété.

Commençons par visiter l'application s'exécutant sur l'instance du maitre avec votre navigateur. Cela devrait produire la première session HTTP. Maintenant, ouvrons la console JMX de JBoss AS sur l'instance ou nous navigons sur le MBean suivant:

- *Category:* jboss.cache
- *Entry:* service=TomcatClusteringCache
- *Method:* printDetails()

Invocation de la méthode printDetails(). Vous allez voir un arbre de sessions HTTP actives. Vérifiez que la session de votre navigateur correspond à une des sessions dans cet arbre.

Maintenant basculer vers l'instance esclave et invoquer la même méthode dans la console JMX. Vous devriez voir une liste identique (au moins au dessous de cette application son chemin de contexte).

Donc vous pouvez voir au moins les deux serveurs clamant avoir les deux sessions identiques. Maintenant, il est temps de tester que les données sont sérialisées et désérialisées proprement.

Connectez vous en utilisant l'URL de l'instance du maitre. Ensuite, construisez une URL pour une seconde instance en mettant ;jsessionid=XXXX immédiatement après le chemin de la servlet et en changeant l'adresse IP. Vous devriez voir que la session a été transférée à l'autre instance. Maintenant, tuez l'instance maitre et regardez que vous pouvez continuer à utiliser l'application depuis l'instance esclave. Retirez le déploiement depuis le dossier du serveur /all/farm et redémmarez l'instance de nouveau. Re-modifiez l'IP dans l'URL pour avoir l'instance du maitre et visitez l'URL. Vous allez voir que la session orginelle est toujours en cours d'utilisation.

Une façon de voir les objets mis en pause et activé est de créer un composant Seam d'étendue conversationnel et d'implémenter les méthode du cylce de vie. Vous pouvez même utiliser les méthode de l'interface HttpSessionActivationListener (Seam enregistre automatiquement cette interface pour tous les composants non-EJB):

```
public void sessionWillPassivate(HttpSessionEvent e);
public void sessionDidActivate(HttpSessionEvent e);
```

Ou vous pouvez simplement indiquer les deux non-arguments de la méthode public void avec respectivement `@PrePassivate` et avec`@PostActivate`. Notez que cette étape de la mise en pause intervient à la fin de chaque requête, alors que l'étape d'activetion intervient quand un noeud est solicité.

Maintenant que vous comprennez le grand tableau de l'exécution de Seam avec un cluster, il est temps de s'addresser au plus mystérieux , mais remarquable agent, de Seam le ManagedEntityInterceptor.

35.2. Mise en pause EJB et le ManagedEntityInterceptor

Le ManagedEntityInterceptor (MEI) est un intercepteur optionnel de Seam qu s'applique sur les composants d'étendue conversationnel quand il est activé. L'activation est simple. Vous définissez juste la propriété distribuable de org.jboss.seam.init.core component à vrai. Ou plus simplement à faire, vous ajoutez (ou mettez à jours) la déclaration de composant suivante dans le descripteur de composants (autrement dit, components.xml).

```
<core:init distributable="true"/>
```

Notez que cela n'active pas la réplication des sessions HTTP, mais cela doit préparer Seam à être capable de gérer la mise en pause aussi bien des composants EJB que des composants de la session HTTP.

Le MEI permet à deux scénario différents (la mise en pause EJB et la mise en pause de session HTTP), de se réaliser avec le même objectif général. Cela s'assure que pendant toute la vie de la conversation au moins un contexte de persistance étendue, les instances d'entité chargées par le(les) contexte(s) de persistance restent gérés (ils ne deviennent pas détaché prématurément

par un évènement de mise en pause). Pour faire court, il s'assure de l'intégrité du contexte de persistance étendue (et par la même occasion il la garantie).

L'instruction précédente implique qu'il y a un défit qui menace ce contrat. Dans les fait, il y en a deux. Un est quand un bean de session avec état (SFSB) qui conserve un contexte de persistance étendue est mis en pause (pour préserver la mémoire ou pour le migrer vers un autre noeud du cluster) et le deuxième est quand la session HTTP est mise en pause (pour préaprer la migration vers un autre noeud dans le cluster).

En premie, jeux parler du problème général de la mise en pause et ensuite regarder les deux défis cités individuellement.

35.2.1. La frictions entre la mise en pause et la persistance

Le contexte de persistance est quand le gestionnaire de persistance (autrement dit, l'EntityManager ou la Session Hibernate) stocke les instances d'entité (autrement dit, les objets) qu'il a chargé depuis la base de données (via le mappage objet-relationnel). Avec un contexte de persistance, il n'y a pas plus d'un objet par enregistrement unique de la base de données. Le contexte de persistance est souvant référencé comme le cache de premier niveau si l'application demande un enregistrement par son identifiant unique qui a déjà été chargé dans le contexte de persistant, un appel à la base de données est évité. Mais c'est bien plus que juste une mise en cache.

Les objets qui conservent le contexte de persistance peuvent être modifié ce que traque le gestionnaire de persistance. Quand un objet est modifié, il est considéré comme "sale". Le gestionnaire de persistance va migrer ces modifications dans la base de données en utilisant une technique connu comme écrire-avec-retard (ce qui basiquement signifie seulement quand c'est nécéssaire). Donc, le contexte de persistance maintient un groupe de modificactions en attente de la base de données.

Les applications orientés base de données doivent faire bien plus que juste lire et écrire dans la base de données. Elles capturent les morceaux d'informations transactionnelle qui doivent être transférées dans la base de données automatiquement (en une seule fois). Ce n'est pas toujours possible de capturer toutes ces informations en une seul fois. De plus, l'utilisateur peut avoir besoin de prendre une décision pour approuver ou rejeter les modificaitons en attente.

Ce que nous avons ici est que l'idée d'une transaction depuis une perspective utilisateur a besoin d'être étendue. Et c'est pourquoi le contexte de persistence géré correspond parfaitement avec ce prérequis. Il peut conserver toutes les modificaitons aussi longtemps que l'application peut le conserver ouvert et ensuite utilisera les fonctionnalités livré du gestionnaire de persistance pour pousser ces modifications en attente vers la base de données sans demander au développeur de l'application de s'inquiéter à propos des détails bas-niveaux (un simple appel à `EntityManager#flush()` fait le truc).

Le lien entre le gestionnaire de persistance et les instance d'entité est maintenu en utilisant des références d'objet. Les instances d'entité sont sérialisable, mais le gestionnaire de persistance (et il fonctionne dans son contexte de persistance) ne l'est pas. Pourtant, le processus de sérialisation

fonctionne pour ce design. La sérialisation peut intervenir aussi quand une session SFSB que HTTP est rendue passive. Donc pour maintenir l'activité dans l'application, le gestionnaire de persistance et les instance d'entité qu'il gère doivent survivre à la sérialisation sans perdre leur relation. C'est l'aide que l'on obtient du MEI.

35.2.2. Cas d'utilisation #1: Survivre à la mise en pause EJB

Les conversations ont été initialement désigné dans l'esprit de beans de sessions avec état (SFSBs), premièrement parce que la spécification EJB 3 désigne les SFSBs comme l'hote diu contexte de persistance étendue. Seam introduit un complément au contexte de persistance étendue, connu comme un contexte de persistance géré par Seam, qui fonctionne autour d'un certain nombre de limitations dans la spécification (règles de propagation complexe et un manque de validation manuel). Les deux peuvent être utilisé avec un SFSB.

Un SFSB lie un client qui contient une référence sur lui dans le but de la conserver active. Seam a fourni un lieu idéal pour cette référence dans le contexte conversationneL. Ainsi, aussi longtemps que le contexte conversationnel est actif, le SFSB est actif. Si un EntityManager est injecté dans le SFSB en utilisant l'annocation @PersistenceContext(EXTENDED), alors cet EntityManager sera remoné au SFSB et restera ouvert pendant son cycle de vie, le cycle de vie de la conversation. Si un Entitymanager est injecté en utilisant @In, alors l'EntityManager est maintenu par Seam et stocké directement dans le contexte conversationnel, alors il vivra pendant la durée de vie de la conversation indépendamment du cycle de vie du SFSB.

Avec tout ce qui est dit, le containeur Java EE peut être mettre en pause un SFSB ce qui signifi qu'il va sérialiser l'objet dans une zone de stockage externe à la JVM. Quand cela arrive, ce qui dépend des réglages individuel du SFSB. Le processus peut même être désactivé. Cependant, le contexte de persistence n'est pas sérialisé (est ce toujours vrai avec SMPL?). dans les fait, ce qui arrive dépend grandement du containeur Java EE. La spécification n'est pas vraiment claire à propos de cette situation. Beaucoup de fournisseurs vous dise juste de ne pas faire en sorte que cela arrive, si vous avez besoin d'une garantie sur le contexte de persistance étendue. L'approche de Seam plus conservative. Seam par défaut ne croit pas en le SFSB avec le contexte de persistance ou avec les instances d'entité. Après chaque invocation du SFSB, Seam déplace sa référence dans l'instance d'entité conservé par le SFSB dans la conversation courrante (et ainsi dans la session HTTP), met à null ces champs dans le SFSB. Il restaure ensuite cette référence au début de la prochaine invocation. Bien sûr, Seam stocke déjà le gestionnaire de persistance dans la conversation. Ainsi, quand le SFSV est mis en pause et plus tard activé, il n'est absolument pas réfractaire à l'application.

Note

Si vous utilisez des SFSBs dans votre application qui conserve des références vers les contextes de persistanes étendue et si ces SFSB peuvent se mettre en pause, alors vous devez utiliser le MEI. Ce prérequis est nécéssaire même si vous utilisez une seule instance (et non pas un cluster). Cependant, si vous utiliser un SFSB en cluster, alors ce prérequis s'applique aussi.

Il est possible de désactiver la mise en pause sur un SFSB. Voyez la page *Ejb3DisableSfsbPassivation* [http://www.jboss.org/community/docs/DOC-9656] sur le Wiki de Jboss pour les détails.

35.2.3. Cas d'utilisation #2: La survie de la réplication de la session HTTP

La gestion de la mise en passif d'un SFSB fonctionne par l'exploitation de la session HTTP. Mais qu'arrive t'il quand la session HTTP devient passive? Cela arrive dans un environnement en clusteravec la réplication de session activé. Ce cas est beaucoup plus difficile à gérer et c'est là que la grosse infrastructure MEI rentre en jeu. Dans ce cas, le gestionnaire de persistance va être détruit parce qu'il ne peut pas être sérialisé. Seam gère sa destruction (donc s'assurant que la session HTTP est sérialisé proprement). Mais qu'arrive t'il de l'autre côté. Et bien, quand le MEI colle une instance d'entité dans la conversation, il embarque cette instance dans un bloc qui va fournir l'information de comment réassocier cette instance avec le gestionnaire de persistance en post-séralisation. Le gros défaut ici est que quand le contexte de persistance est reconstruit (depuis la base de données), les modifications en attente sont détruites. Cependant, que fait Seam pour s'assurer que si l'instance de l'entité est versionné, il garanti que le vérrou optimistique est à faire respecter. (pourquoi ce n'est pas l'état sale qui est transféré?)

Note

Si vous deployez dans votre application avec un cluster et en utilisant la réplication de la session HTTP, vous devez utiliser le MEI.

35.2.4. Emballage du ManagedEntityInterceptor

Le point important dans cette section est que le MEI est là pour cette raison. Il est là pour s'assurer que le contexte de persistance étendue peut revenir intact suite à uune mise en pause (avec aussi bien un SFSB qu'une session HTTP). Cela compte parce que le design naturel ddes application de Seam (et de l'état conversaionnel en général) résoud tout ce qui tourne autour de l'état de cette ressource.

Performance Tuning

This chapter is an attempt to document in one place all the tips for getting the best performance from your Seam application.

36.1. Bypassing Interceptors

For repetitive value bindings such as those found in a JSF dataTable or other iterative control (like `ui:repeat`), the full interceptor stack will be invoked for every invocation of the referenced Seam component. The effect of this can result in a substantial performance hit, especially if the component is accessed many times. A significant performance gain can be achieved by disabling the interceptor stack for the Seam component being invoked. To disable interceptors for the component, add the `@BypassInterceptors` annotation to the component class.

Avertissement

It is very important to be aware of the implications of disabling interceptors for a Seam component. Features such as bijection, annotated security restrictions, synchronization and others are unavailable for a component marked with `@BypassInterceptors`. While in most cases it is possible to compensate for the loss of these features (e.g. instead of injecting a component using `@In`, you can use `Component.getInstance()` instead) it is important to be aware of the consequences.

The following code listing demonstrates a Seam component with its interceptors disabled:

```
@Name("foo")
@Scope(EVENT)
@BypassInterceptors
public class Foo
{
  public String getRowActions()
  {
     // Role-based security check performed inline instead of using @Restrict or other security
 annotation
    Identity.instance().checkRole("user");

    // Inline code to lookup component instead of using @In
    Bar bar = (Bar) Component.getInstance("bar");

    String actions;
    // some code here that does something
```

```
    return actions;
  }
}
```

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.

37.1. 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 which creates a statement of account for a customer:

```
@Stateless
@Scope(EVENT)
@Name("statementOfAccount")
public class StatementOfAccount {

  @In(create=true) EntityManager entityManager

  private double statementTotal;

  @In
  private Customer customer;

  @Create
  public void create() {
    List<Invoice> invoices = entityManager
      .createQuery("select invoice from Invoice invoice where invoice.customer = :customer")
      .setParameter("customer", customer)
      .getResultList();
    statementTotal = calculateTotal(invoices);
  }

  public double calculateTotal(List<Invoice> invoices) {
    double total = 0.0;
    for (Invoice invoice: invoices)
    {
      double += invoice.getTotal();
    }
```

```
    return total;
  }

  // getter and setter for statementTotal

}
```

We could write a unit test for the calculateTotal method (which tests the business logic of the component) as follows:

```
public class StatementOfAccountTest {

  @Test
  public testCalculateTotal {
    List<Invoice> invoices = generateTestInvoices(); // A test data generator
    double statementTotal = new StatementOfAccount().calculateTotal(invoices);
    assert statementTotal = 123.45;
  }
}
```

You'll notice we aren't testing retrieving data from or persisting data to the database; nor are we testing any functionality provided by Seam. We are just testing the logic of our POJOs. Seam components don't usually depend directly upon container infrastructure, so most unit testing are as easy as that!

However, if you want to test the entire application, read on.

37.2. Integration testing Seam components

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.

The approach taken by Seam is to let you write tests that exercise your components while running inside a pruned down container environment (Seam, together with the JBoss Embedded container; see *Section 30.6.1, « Installing Embedded JBoss »* for configuration details)

```
public class RegisterTest extends SeamTest
{

  @Test
```

```
  public void testRegisterComponent() throws Exception
  {

    new ComponentTest() {

      protected void testComponents() throws Exception
      {
        setValue("#{user.username}", "1ovthafew");
        setValue("#{user.name}", "Gavin King");
        setValue("#{user.password}", "secret");
        assert invokeMethod("#{register.register}").equals("success");
        assert getValue("#{user.username}").equals("1ovthafew");
        assert getValue("#{user.name}").equals("Gavin King");
        assert getValue("#{user.password}").equals("secret");
      }

    }.run();

  }

  ...

}
```

37.2.1. Using mocks in integration tests

Occasionally, we need to be able to replace the implementation of some Seam component that depends upon resources which are not available in the integration test environment. For example, suppose we have some Seam component which is a facade to some payment processing system:

```
@Name("paymentProcessor")
public class PaymentProcessor {
   public boolean processPayment(Payment payment) { .... }
}
```

For integration tests, we can mock out this component as follows:

```
@Name("paymentProcessor")
@Install(precedence=MOCK)
public class MockPaymentProcessor extends PaymentProcessor {
   public boolean processPayment(Payment payment) {
      return true;
```

```
  }
}
```

Since the `MOCK` precedence is higher than the default precedence of application components, Seam will install the mock implementation whenever it is in the classpath. When deployed into production, the mock implementation is absent, so the real component will be installed.

37.3. Integration testing Seam application user interactions

An even harder 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.

`SeamTest` lets you write *scripted* tests, in a simulated JSF environment. The role of a scripted test is 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>
```

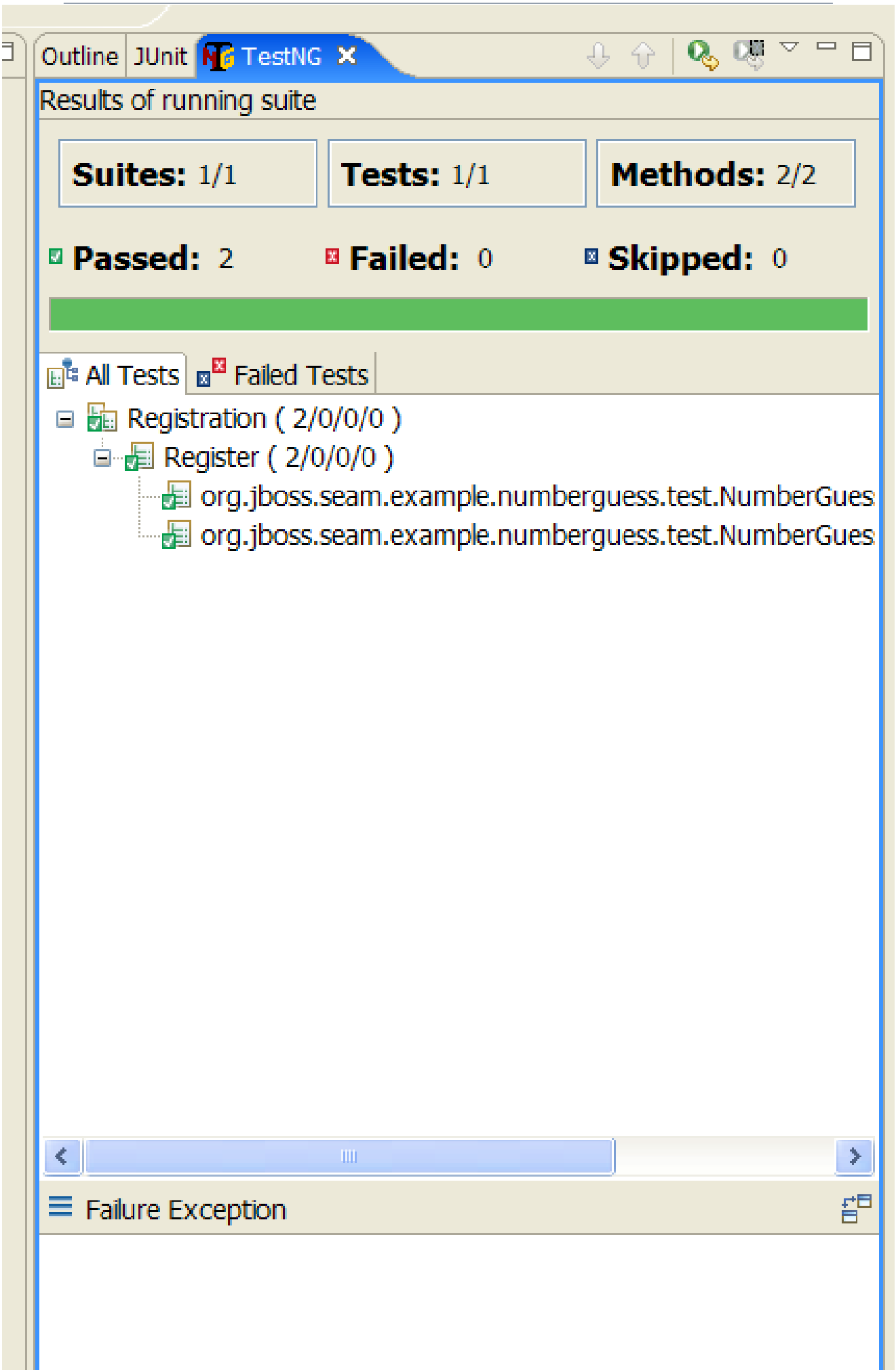
We want to test the registration functionality of our application (the stuff that happens when the user clicks the Register button). We'll reproduce the JSF request lifecycle in an automated TestNG test:

```
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

37.3.1. Configuration

If you used seam-gen to create your project you are ready to start writing tests. Otherwise you'll need to setup the testing environment in your favorite build tool (e.g. ant, maven, eclipse).

First, lets look at the dependencies you need at a minimum:

Tableau 37.1.

Group Id	Artifact Id	Location in Seam
`org.jboss.seam.embedded`	`hibernate-all`	`lib/test/hibernate-all.jar`
`org.jboss.seam.embedded`	`jboss-embedded-all`	`lib/test/jboss-embedded-all.jar`
`org.jboss.seam.embedded`	`thirdparty-all`	`lib/test/thirdparty-all.jar`
`org.jboss.seam.embedded`	`jboss-embedded-api`	`lib/jboss-embedded-api.jar`
`org.jboss.seam`	`jboss-seam`	`lib/jboss-seam.jar`
`org.jboss.el`	`jboss-el`	`lib/jboss-el.jar`
`javax.faces`	`jsf-api`	`lib/jsf-api.jar`
`javax.el`	`el-api`	`lib/el-api.jar`
`javax.activation`	`javax.activation`	`lib/activation.jar`

It's very important you don't put the compile time JBoss AS dependencies from `lib/` (e.g. `jboss-system.jar`) on the classpath, these will cause Embedded JBoss to not boot. So, just add the dependencies (e.g. Drools, jBPM)you need as you go.

You also need to include the `bootstrap/` directory on the classpath; `bootstrap/` contains the configuration for Embedded JBoss.

And, of course you need to put your built project and tests onto the classpath as well as jar for your test framework. Don't forget to put all the correct configuration files for JPA and Seam onto the classpath as well.Seam asks Embedded JBoss to deploy any resource (jar or directory) which has `seam.properties` in it's root. Therefore, if you don't assemble a directory structure that resembles a deployable archive containing your built project, you must put a `seam.properties` in each resource.

By default, a generated project will use the `java:/DefaultDS` (a built in HSQL datasource in Embedded JBoss) for testing. If you want to use another datasource place the `foo-ds.xml` into `bootstrap/deploy` directory.

37.3.2. Using SeamTest with another test framework

Seam provides TestNG support out of the box, but you can also use another test framework, such as JUnit, if you want.

You'll need to provide an implementation of `AbstractSeamTest` which does the following:

- Calls `super.begin()` before every test method.
- Calls `super.end()` after every test method.
- Calls `super.setupClass()` to setup integration test environment. This should be called before any test methods are called.
- Calls `super.cleanupClass()` to clean up the integration test environment.
- Calls `super.startSeam()` to start Seam at the start of integration testing.
- Calls `super.stopSeam()` to cleanly shut down Seam at the end of integration testing.

37.3.3. Integration Testing with Mock Data

If you want to insert or clean data in your database before each test you can use Seam's integration with DBUnit. To do this, extend `DBUnitSeamTest` rather than `SeamTest`.

You have to provide a dataset for DBUnit.

Attention

DBUnit supports two formats for dataset files, flat and XML. Seam's `DBUnitSeamTest` assumes the flat format is used, so make sure that your dataset is in this format.

```
<dataset>

  <ARTIST
    id="1"
    dtype="Band"
    name="Pink Floyd" />

  <DISC
    id="1"
    name="Dark Side of the Moon"
    artist_id="1" />
```

```
</dataset>
```

In your test class, configure your dataset with overriding `prepareDBUnitOperations()`:

```
protected void prepareDBUnitOperations() {
   beforeTestOperations.add(
      new DataSetOperation("my/datasets/BaseData.xml")
   );
}
```

`DataSetOperation` defaults to `DatabaseOperation.CLEAN_INSERT` if no other operation is specified as a constructor argument. The above example cleans all tables defined `BaseData.xml`, then inserts all rows declared in `BaseData.xml` before each `@Test` method is invoked.

If you require extra cleanup after a test method executes, add operations to `afterTestOperations` list.

You need to tell DBUnit which datasource you are using. This is accomplished by defining a *test parameter* [http://testng.org/doc/documentation-main.html#parameters-testng-xml] named `datasourceJndiName` in testng.xml as follows:

```
<parameter name="datasourceJndiName" value="java:/seamdiscsDatasource"/>
```

DBUnitSeamTest has support for MySQL and HSQL - you need to tell it which database is being used, otherwise it defaults to HSQL:

```
<parameter name="database" value="MYSQL" />
```

It also allows you to insert binary data into the test data set (n.b. this is untested on Windows). You need to tell it where to locate these resources on your classpath:

```
<parameter name="binaryDir" value="images/" />
```

You do not have to configure any of these parameters if you use HSQL and have no binary imports. However, unless you specify `datasourceJndiName` in your test configuration, you will have to call `setDatabaseJndiName()` before your test runs. If you are not using HSQL or MySQL, you need to override some methods. See the Javadoc of `DBUnitSeamTest` for more details.

37.3.4. Integration Testing Seam Mail

Attention

Warning! This feature is still under development.

It's very easy to integration test your Seam Mail:

```
public class MailTest extends SeamTest {

  @Test
  public void testSimpleMessage() throws Exception {

    new FacesRequest() {

      @Override
      protected void updateModelValues() throws Exception {
        setValue("#{person.firstname}", "Pete");
        setValue("#{person.lastname}", "Muir");
        setValue("#{person.address}", "test@example.com");
      }

      @Override
      protected void invokeApplication() throws Exception {
        MimeMessage renderedMessage = getRenderedMailMessage("/simple.xhtml");
        assert renderedMessage.getAllRecipients().length == 1;
        InternetAddress to = (InternetAddress) renderedMessage.getAllRecipients()[0];
        assert to.getAddress().equals("test@example.com");
      }

    }.run();
  }
}
```

We create a new `FacesRequest` as normal. Inside the invokeApplication hook we render the message using `getRenderedMailMessage(viewId);`, passing the viewId of the message to render. The method returns the rendered message on which you can do your tests. You can of course also use any of the standard JSF lifecycle methods.

There is no support for rendering standard JSF components so you can't test the content body of the mail message easily.

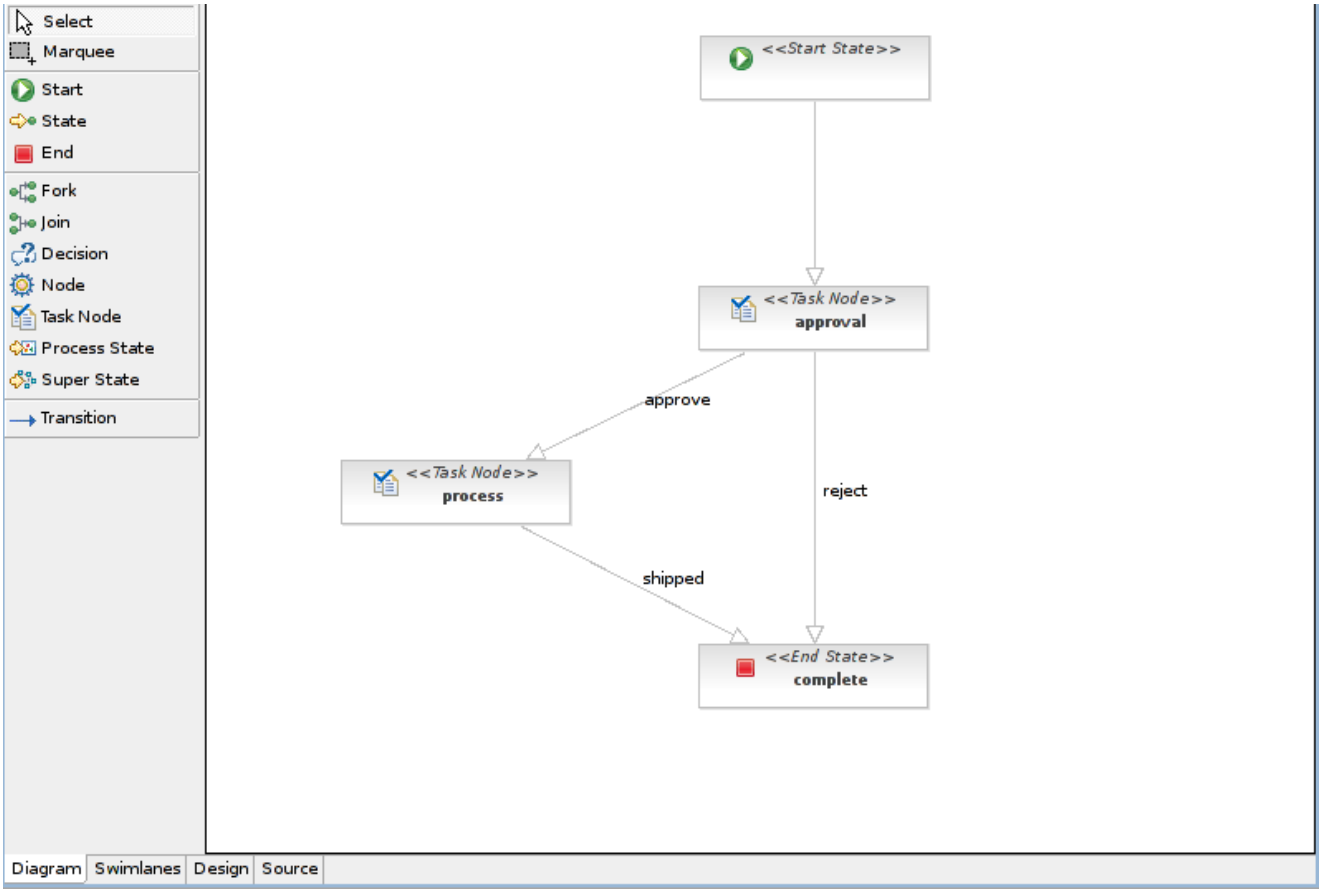
Seam tools

38.1. 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/]

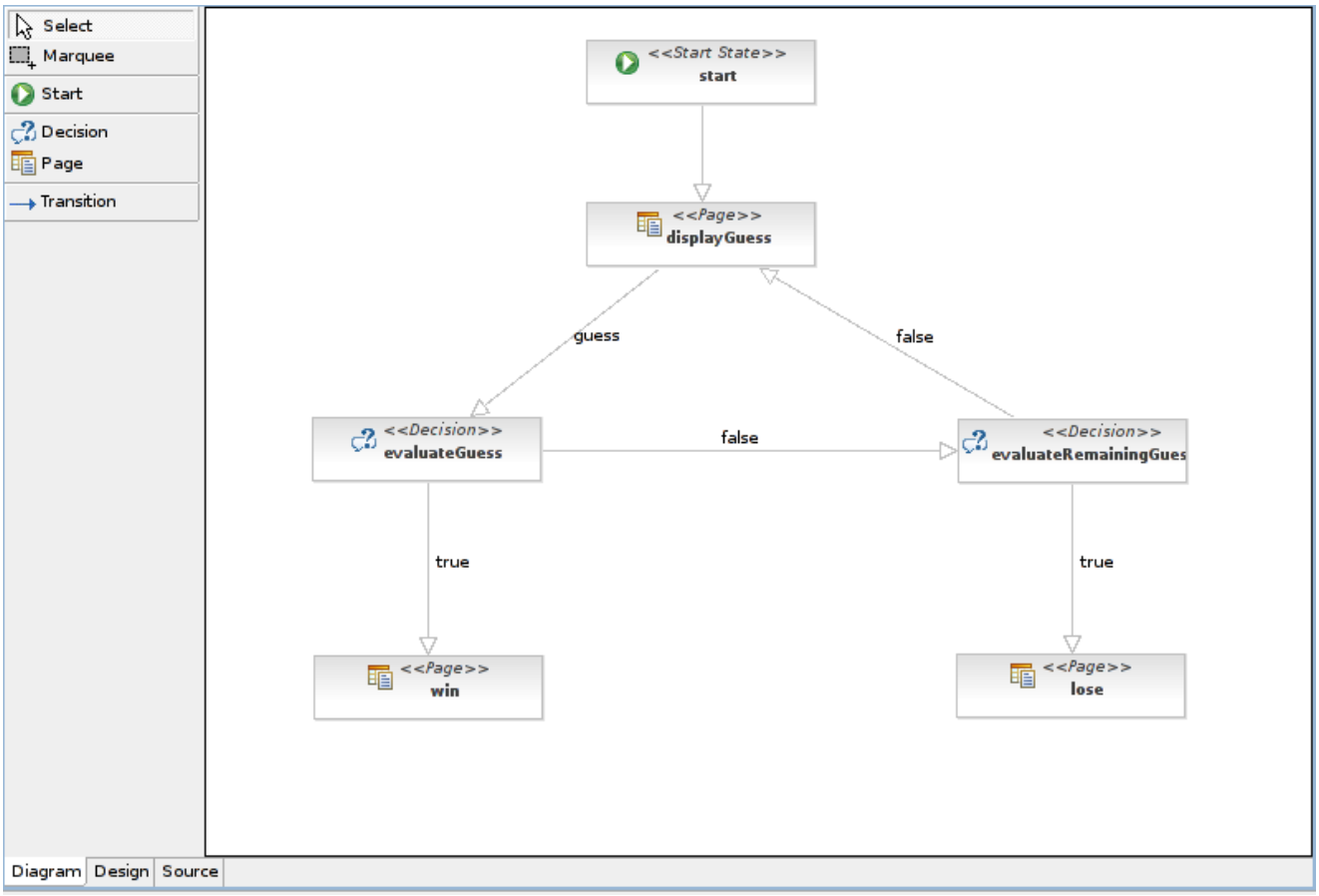
38.1.1. Business process designer

This tool lets you design your own business process in a graphical way.



38.1.2. 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.


Select
Marquee
Start
Decision
Page
Transition
<<Start State>>
start
<<Page>>
displayGuess
guess
false
<<Decision>>
evaluateGuess
false
<<Decision>>
evaluateRemainingGues
true
true
<<Page>>
win
<<Page>>
lose
Diagram
Design
Source

Seam on BEA's Weblogic

Weblogic 10.3 is BEA's latest stable JEE5 server offering. Seam applications can be deployed and developed on Weblogic servers, and this chapter will show you how. There are some known issues with the Weblogic servers that will need to be worked around, and configuration changes that are needed specific to Weblogic.

First step is to get Weblogic downloaded, installed and running. Then we'll talk about Seam's JEE5 example and the hurdles to getting it running. After that, the JPA example will be deployed to the server. Then finally we will create a `seam-gen` application and get it up and running to provide a jump start to your own application.

39.1. Installation and operation of Weblogic

First things first we need to get the server installed. There are some outstanding issues that were not addressed in 10.3, but it does solve some of the issues discussed below without the need for BEA patches. The previous release 10.0.MP1 is also available, but requires some BEA patches to function correctly.

- `Weblogic 10.0.MP1` — *Download page* [http://www.oracle.com/technology/software/products/ias/htdocs/wls_main.html]

  10.0.MP1 has some known issues with EJBs that use `varargs` in their methods (it confuses them as `transient`), as well as some others. See *Section 39.2.1, « EJB3 Issues with Weblogic »* for full details on the issues and the work around.

- `Weblogic 10.3` — *Download page* [http://www.oracle.com/technology/software/products/ias/htdocs/wls_main.html]

  This is the latest stable release of the Weblogic server, and the one that will be used with the examples below. This version has addressed some of the issues with EJBs that were in 10.0.MP1. However one of the changes did not make it into this release. See *Section 39.2.1, « EJB3 Issues with Weblogic »* for the details, but because of this we still need to use the special Weblogic seam jar discussed below.

Special `jboss-seam.jar` for Weblogic EJB Support

Starting with Seam 2.0.2.CR2 a special Weblogic specific jar has been created that does not contain the `TimerServiceDispatcher` . This is the EJB that uses `varargs` and exposes the second EJB issue. We will be using this jar for the `jee5/booking` example, as it avoids the known BEA issues.

39.1.1. Installing 10.3

Here are the quick steps to installing Weblogic 10.3. For more details or if you are having any issues please check with the BEA docs at the *Weblogic 10.3 Doc Center* [http://edocs.bea.com/wls/docs103/] . Here we install the RHEL 5 version using the graphical installer:

1. Follow the link given above for 10.3 and download the correct version for your environment. You will need to sign up for an account with Oracle in order to do this.
2. You may need to change the the `server103_XX.bin` file to be executable:

```
chmod a+x server103_XX.bin
```

3. Execute the install:

```
./server103_XX.bin
```

4. When the graphical install loads, you need to set the BEA home location. This is where all BEA applications are installed. This location will be known as `$BEA_HOME` in this document e.g.:

```
/jboss/apps/bea
```

5. Select `Complete` as the installation type. You do not need all the extras of the complete install (such as struts and beehive libraries), but it will not hurt.
6. You can leave the defaults for the component installation locations on the next page.

39.1.2. Creating your Weblogic domain

A Weblogic domain is similar to a JBoss server configuration - it is a self contained server instance. The Weblogic server you just installed has some example domains, but we are going to create one just for the seam examples. You can use the existing domains if you wish (modify the instructions as needed).

1. Start up the Weblogic configuration wizard:

```
$BEA_HOME/wlserver_10.3/common/bin/config.sh
```

2. Choose to create a new domain, configured to support `Weblogic Server`. Note that this is the default domain option.

3. Set a username and password for this domain.

4. Next choose `Development Mode` and the default JDK when given the option.

5. The next screen asks if you want to customize any setting. Select `No`.

6. Finally set the name of the domain to `seam_examples` and leave the default domain location.

39.1.3. How to Start/Stop/Access your domain

Now that the server is installed and the domain is created you need to know how to start and stop it, plus how to access its configuration console.

- Starting the domain:

  This is the easy part - go to the `$BEA_HOME/user_projects/domains/seam_examples/bin` directory and run the `./startWeblogic.sh` script.

- Accessing the configuration console:

  Launch `http://127.0.0.1:7001/console` in your web browser. It will ask for your username and password that you entered before. We won't get into this much now, but this is the starting point for a lot of the various configurations that are needed later.

- Stopping the domain:

  There are a couple of options here:

  - The recommended way is through the configuration console:

    1. Select `seam_examples` on the left hand side of the console.
    2. Choose the `Control` tab in the middle of the page.
    3. Select the check box `AdminServer` in the table.
    4. Choose `Shutdown` just above the table, and select either `When work completes` or `Force shutdown now` as appropriate.

  - Hitting `Ctrl-C` in the terminal where you started the domain.

    No negative effects have been seen, but we would not recommend doing this while in the middle of configuration changes in the console.

-

A note on Weblogic classloading

When using the `/autodeploy` directory as described in this chapter you may see `NoClassDefFound` exceptions during redeployment's. If you see this try restarting the Weblogic server. If you still see it remove the auto-deployed EAR/

WAR files, restart the server, and redeploy. We could not find a specific reason for this, but others seem to be having this issue as well.

39.1.4. Setting up Weblogic's JSF Support

These are the instructions to deploy and configure Weblogic's JSF 1.2 libraries. Out of the box Weblogic does not come with its own JSF libraries active. For complete details see *Weblogic 10.3 Configuring JSF and JSTL Libraries* [http://edocs.bea.com/wls/docs103/webapp/configurejsfandjtsl.html]

1. In the administration console navigate to the `Deployments` page using the left hand menu.
2. Then select the `Install` button at the top of the deployments table
3. Using the directory browser navigate to the `$BEA_HOME/wlserver_10.3/common/deployable-libraries` directory. Then select the `jsf-1.2.war` archive, and click the `Next` button.
4. Make sure that the `Install this deployment as a library` is selected. Click the `Next` button on the `Install Application Assistant` page.
5. Click the `Next` button on the `Optional Settings` page.
6. Make sure that the `Yes, take me to the deployment's configuration screen.` is selected. Click the `Finish` button on the `Review your choices and click Finish` page.
7. On the `Settings for jsf(1.2,1.2.3.1)` page set the `Deployment Order` to `99` so that it is deployed prior to auto deployed applications. Then click the `Save` button.

There is another step that is needed for this to work. For some reason, even with the steps above classes in the `jsf-api.jar` are not found during application deployment. The only way for this to work is to put the `javax.jsf_1.2.0.0.jar` (the jsf-api.jar) from `jsf-1.2.war` in the domains shared library. This requires a restart of the server.

39.2. The `jee5/booking` Example

Do you want to run Seam using EJB's on Weblogic? If so there are some obstacles that you will have to avoid, or some patches that are needed from BEA. This section describes those obstacles and what changes are needed to the `jee5/booking` example to get it deployed and functioning.

39.2.1. EJB3 Issues with Weblogic

For several releases of Weblogic there has been an issue with how Weblogic generates stubs and compiles EJB's that use variable arguments in their methods. This is confirmed in the Weblogic 9.X and 10.0.MP1 versions. Unfortunately the 10.3 version only partially addresses the issue as detailed below.

39.2.1.1. The `varargs` Issue

The basic explanation of the issue is that the Weblogic EJB compiler mistakes methods that use `varargs` as having the `transient` modifier. When BEA generates its own stub class from those classes during deployment it fails and the deployment does not succeed. Seam uses variable arguments in one of its internal EJB's (`TimerServiceDispatcher`). If you see exceptions like below during deployment you are running an unpatched version of 10.0.MP1.

```
java.io.IOException: Compiler failed executable.exec:
/jboss/apps/bea/wlserver_10.0/user_projects/domains/seam_examples/servers/AdminServer
/cache/EJBCompilerCache/5yo5dk9ti3yo/org/jboss/seam/async/
TimerServiceDispatcher_qzt5w2_LocalTimerServiceDispatcherImpl.java:194: modifier transient
not allowed here
  public transient javax.ejb.Timer scheduleAsynchronousEvent(java.lang.String arg0,
  java.lang.Object[] arg1)
                     ^
/jboss/apps/bea/wlserver_10.0/user_projects/domains/seam_examples/servers/AdminServer
/cache/EJBCompilerCache/5yo5dk9ti3yo/org/jboss/seam/async/
TimerServiceDispatcher_qzt5w2_LocalTimerServiceDispatcherImpl.java:275: modifier transient
not allowed here
  public transient javax.ejb.Timer scheduleTimedEvent(java.lang.String arg0,
  org.jboss.seam.async.TimerSchedule arg1, java.lang.Object[] arg2)
```

This issue has been fixed in Weblogic 10.3, and BEA has created a patch for Weblogic 10.0.MP1 (`CR327275`) for this issue that can be requested from their support.

Unfortunately a second issue has been reported and verified by BEA.

39.2.1.2. Missing EJB Methods Issue

This issue was only found once the `CR327275` patch had been applied to 10.0.MP1. This new issue has been confirmed by BEA and they created a patch for 10.0.MP1 that addresses this issue. This patch has been referred to as both `CR370259` and `CR363182`. As with the other patch this can be requested through the BEA support.

This issue causes certain EJB methods to be incorrectly left out of Weblogic's generated internal stub classes. This results in the following error messages during deployment.

```
<<Error> <EJB> <BEA-012036> <Compiling generated EJB classes produced the following Java
 compiler error message:
<Compilation Error> TimerServiceDispatcher_qzt5w2_Impl.java: The type
   TimerServiceDispatcher_qzt5w2_Impl must implement the inherited abstract method
 TimerServiceDispatcher_qzt5w2_Intf.scheduleTimedEvent(String, Schedule, Object[])
```

```
<Compilation Error> TimerServiceDispatcher_qzt5w2_LocalTimerServiceDispatcherImpl.java:
 Type mismatch: cannot convert from Object to Timer
<Compilation Error> TimerServiceDispatcher_qzt5w2_LocalTimerServiceDispatcherImpl.java:
 Type mismatch: cannot convert from Object to Timer>
<Error> <Deployer> <BEA-149265> <Failure occurred in the execution of deployment request
 with ID '1223409267344' for task '0'. Error is: 'weblogic.application.ModuleException: Exception
 preparing module: EJBModule(jboss-seam.jar)
```

It appears that when Weblogic 10.3 was released the neglected to include this fix!! This means that Weblogic 10.0.MP1 with patches will function correctly, but 10.3 will still require the special Seam jar described below. Not all users have seen this and there may be certain combinations of OS/JRE that do not see this, however is has been seen many times. Hopefully Oracle/BEA will release an updated patch for this issue on 10.3. When they do we will update this reference guide as needed.

So that Seam's users can deploy an EJB application to Weblogic a special Weblogic specific jar has been created, starting with Seam 2.0.2.CR2. It is located in the `$SEAM/lib/interop` directory and is called `jboss-seam-wls-compatible.jar` . The only difference between this jar and the `jboss-seam.jar` is that it does not contain the `TimerServiceDispatcher` EJB. To use this jar simply rename the `jboss-seam-wls-compatible.jar` to `jboss-seam.jar` and replace the original in your applications `EAR` file. The `jee5/booking` example demonstrates this. Obviously with this jar you will not be able to use the `TimerServiceDispatcher` functionality.

39.2.2. Getting the `jee5/booking` Working

In this section we will go over the steps needed to get the `jee5/booking` example to up and running.

39.2.2.1. Setting up the hsql datasource

This example uses the in memory hypersonic database, and the correct data source needs to be set up. The admin console uses a wizard like set of pages to configure it.

1. Copy `hsqldb.jar` to the Weblogic domain's shared library directory: `cp $SEAM_HOME/lib/hsqldb.jar $BEA_HOME/user_projects/domains/seam_examples/lib`
2. Start up the server and navigate to the administration console following *Section 39.1.3, « How to Start/Stop/Access your domain »*
3. On the left side tree navigate `seam_examples - Services- JDBC - Data Sources`.
4. Then select the `New` button at the top of the data source table
5. Fill in the following:
   a. Name: `seam-jee5-ds`
   b. JNDI Name: `seam-jee5-ds`

c. Database Type and Driver: other

d. Select Next button

6. Select Next button on the Transaction Options page

7. Fill in the following on the Connection Properties page:

a. Database Name: hsqldb

b. Host Name: 127.0.0.1

c. Port: 9001

d. Username: sa will empty password fields.

e. Password: leave empty.

f. Select Next button

8. Fill in the following on the Connection Properties page:

a. Driver Class Name: org.hsqldb.jdbcDriver

b. URL: jdbc:hsqldb:.

c. Username: sa

d. Password: leave empty.

e. Leave the rest of the fields as is.

f. Select Next button

9. Choose the target domain for the data source in our case the only one AdminServer. Click Next.

39.2.2.2. Configuration and Build changes

OK - now we are ready to finally begin adjusting the seam application for deployment to the Weblogic server.

resources/META-INF/persistence.xml

- Change the jta-data-source to what you entered above :

```
<jta-data-source>seam-jee5-ds</jta-data-source>
```

- Then comment out the glassfish properties.

- Then add these two properties for weblogic support.

```
<property name="hibernate.dialect"
          value="org.hibernate.dialect.HSQLDialect"/>
<property name="hibernate.transaction.manager_lookup_class"
          value="org.hibernate.transaction.WeblogicTransactionManagerLookup"/>
```

`resources/META-INF/weblogic-application.xml`

- This file needs to be created and should contain the following:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<weblogic-application>
   <library-ref>
      <library-name>jsf</library-name>
      <specification-version>1.2</specification-version>
      <implementation-version>1.2</implementation-version>
      <exact-match>false</exact-match>
   </library-ref>
   <prefer-application-packages>
      <package-name>antlr.*</package-name>
   </prefer-application-packages>
</weblogic-application>
```

- These changes do two two different things. The first element `library-ref` tells weblogic that this application will be using the deployed JSF libraries. The second element `prefer-application-packages` tells weblogic that the `antlr` jars take precedence. This avoids a conflict with hibernate.

`resources/META-INF/ejb-jar.xml`

- The changes described here work around an issue where Weblogic is only using a single instance of the `sessionBeanInterceptor` for all session beans. Seam's interceptor caches and stores some component specific attributes, so when a call comes in - the interceptor is primed for a different component and an error is seen. To solve this problem you must define a separate interceptor binding for each EJB you wish to use. When you do this Weblogic will use a separate instance for each EJB.

  Modify the `assembly-descriptor` element to look like this:

```
```

```
<assembly-descriptor>
  <interceptor-binding>
    <ejb-name>AuthenticatorAction</ejb-name>
    <interceptor-class >org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
  </interceptor-binding>
  <interceptor-binding>
    <ejb-name>BookingListAction</ejb-name>
    <interceptor-class >org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
  </interceptor-binding>
  <interceptor-binding>
    <ejb-name>RegisterAction</ejb-name>
    <interceptor-class >org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
  </interceptor-binding>
  <interceptor-binding>
    <ejb-name>ChangePasswordAction</ejb-name>
    <interceptor-class >org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
  </interceptor-binding>
  <interceptor-binding>
    <ejb-name>HotelBookingAction</ejb-name>
    <interceptor-class >org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
  </interceptor-binding>
  <interceptor-binding>
    <ejb-name>HotelSearchingAction</ejb-name>
    <interceptor-class >org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
  </interceptor-binding>
  <interceptor-binding>
    <ejb-name>EjbSynchronizations</ejb-name>
    <interceptor-class >org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
  </interceptor-binding>
</assembly-descriptor>
```

`resources/WEB-INF/weblogic.xml`

- This file needs to be created and should contain the following:

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

<weblogic-web-app>
<library-ref>
   <library-name>jsf</library-name>
   <specification-version>1.2</specification-version>
   <implementation-version>1.2</implementation-version>
   <exact-match>false</exact-match>
```

```
  </library-ref>
</weblogic-web-app>
```

- This file and the element `library-ref` tells Weblogic that this application will using the deployed JSF libraries. This is needed in both this file and the `weblogic-application.xml` file because both applications require access.

39.2.2.3. Building and Deploying the Application

There are some changes needed to the build script and the `jboss-seam.jar` then we can deploy the app.

`build.xml`

- We need to add the follow so that the `weblogic-application.xml` will be packaged.

```
<!-- Resources to go in the ear -->
<fileset id="ear.resources" dir="${resources.dir}">
   <include name="META-INF/application.xml" />
   <include name="META-INF/weblogic-application.xml" />
   <include name="META-INF/*-service.xml" />
   <include name="META-INF/*-xmbean.xml" />
   <include name="treecache.xml" />
   <include name="*.jpdl.xml" />
   <exclude name=".gpd.*" />
   <include name="*.cfg.xml" />
   <include name="*.xsd" />
</fileset>
```

`$SEAM/lib/interop/jboss-seam-wls-compatible.jar`

- This is the change discussed above in *Section 39.2.1, « EJB3 Issues with Weblogic »* . There are really two options.
- Rename this jar and replace the original `$SEAM/lib/jboss-seam.jar` file. This approach does not require any changes to the packaged `EAR` archive, but overwrites the original `jboss-seam.jar`
- The other option is the modify the packaged `EAR` archive and replace the `jboss-seam.jar` in the archive manually. This leaves the original jar alone, but requires a manual step when ever the archive is packaged.

Assuming that you choose the first option for handling the `jboss-seam-wls-compatible.jar` we can build the application by running `ant archive` at the base of the `jee5/booking` example directory.

Because we chose to create our Weblogic domain in development mode we can deploy the application by putting the EAR file in the domains autodeploy directory.

```
cp ./dist/jboss-seam-jee5.ear
            $BEA_HOME/user_projects/domains/seam_examples/autodeploy
```

Check out the application at `http://localhost:7001/seam-jee5/`

39.3. The `jpa` booking example

This is the Hotel Booking example implemented with Seam POJOs and Hibernate JPA and does not require EJB3 support to run. The example already has a breakout of configurations and build scripts for many of the common containers including Weblogic 10.X

First we'll build the example for Weblogic 10.x and do the needed steps to deploy. Then we'll talk about what is different between the Weblogic versions, and with the JBoss AS version.

Note that this example assumes that Weblogic's JSF libraries have been configured as described in *Section 39.1.4, « Setting up Weblogic's JSF Support »*.

39.3.1. Building and deploying `jpa` booking example

Step one setup the datasource, step two build the app, step three deploy.

39.3.1.1. Setting up the datasource

The Weblogic 10.X version of the example will use the in memory hsql database instead of the built in PointBase database. If you wish to use the PointBase database you must setup a PointBase datasource, and adjust the hibernate setting in `persistence.xml` to use the PointBase dialect. For reference the `jpa/weblogic92` example uses PointBase.

Configuring the datasource is very similar to the jee5 *Section 39.2.2.1, « Setting up the hsql datasource »* . Follow the steps in that section, but use the following entries where needed.

- DataSource Name: `seam-jpa-ds`
- JNDI Name: `seam-jpa-ds`

39.3.1.2. Building the example

Building it only requires running the correct ant command:

```
ant weblogic10
```

This will create a container specific distribution and exploded archive directories.

39.3.1.3. Deploying the example

When we installed Weblogic following *Section 39.1.2, « Creating your Weblogic domain »* we chose to have the domain in development mode. This means to deploy the application all we need to do is copy it into the autodeploy directory.

```
cp ./dist-weblogic10/jboss-seam-jpa.war
$BEA_HOME/user_projects/domains/seam_examples/autodeploy
```

Check out the application at the following `http://localhost:7001/jboss-seam-jpa/` .

39.3.2. What's different with Weblogic 10.x

- Between the the Weblogic 10.x and 9.2 examples there are several differences:
  - `META-INF/persistence.xml` — The 9.2 version is configured to use the `PointBase` database and a pre-installed datasource. The 10.x version uses the `hsql` database and a custom datasource.
  - `WEB-INF/weblogic.xml` — This file and its contents solve an issue with an older version of the `ANTLR` libraries that Weblogic 10.x uses internally. OC4J have the same issue as well. It also configures the application to use the shared JSF libraries that were installed above.

```
<?xml version="1.0" encoding="UTF-8"?>
<weblogic-web-app
xmlns="http://www.bea.com/ns/weblogic/90"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.bea.com/ns/weblogic/90
                    http://www.bea.com/ns/weblogic/90/weblogic-web-app.xsd">
  <library-ref>
    <library-name>jsf</library-name>
    <specification-version>1.2</specification-version>
    <implementation-version>1.2</implementation-version>
    <exact-match>false</exact-match>
  </library-ref>
  <container-descriptor>
    <prefer-web-inf-classes>true</prefer-web-inf-classes>
  </container-descriptor>
</weblogic-web-app>
```

This make Weblogic use classes and libraries in the web application before other libraries in the classpath. Without this change hibernate is required to use a older, slower query factory by setting the following property in the `META-INF/persistence.xml` file.

```
<property name="hibernate.query.factory_class"
    value="org.hibernate.hql.classic.ClassicQueryTranslatorFactory"/>
```

- `WEB-INF/components.xml` — In the Weblogic 10.x version JPA entity transactions is enabled by adding:

```
<transaction:entity-transaction entity-manager="#{em}"/>
```

- `WEB-INF/web.xml` — Because the `jsf-impl.jar` is not in the `WAR` this listener need to be configured :

```
<listener>
  <listener-class>com.sun.faces.config.ConfigureListener</listener-class>
</listener>
```

- Between the Weblogic 10.x version and the JBoss version there are more changes. Here is the rundown:
  - `META-INF/persistence.xml` — Except for datasource name the Weblogic version sets:

```
<property name="hibernate.transaction.manager_lookup_class"
  value="org.hibernate.transaction.WeblogicTransactionManagerLookup"/>
```

  - `WEB-INF/lib` — The Weblogic version requires several library packages because they are not included as they are with JBoss AS. These are primarily for hibernate, and its dependencies.
    - To use Hibernate as your JPA provider you need the following jars:
      - `hibernate.jar`

    - `hibernate-annotations.jar`
    - `hibernate-entitymanager.jar`
    - `hibernate-validator.jar`
    - `jboss-common-core.jar`
    - `commons-logging.jar`
    - `commons-collections.jar`
    - `jboss-common-core.jar`
- Various third party jars that Weblogic needs:
    - `antlr.jar`
    - `cglib.jar`
    - `asm.jar`
    - `dom4j.jar`
    - `el-ri.jar`
    - `javassist.jar`
    - `concurrent.jar`

39.4. Deploying an application created using `seam-gen` on Weblogic 10.x

`seam-gen` is a very useful tool for developers to quickly get an application up and running, and provides a foundation to add your own functionality. Out of box `seam-gen` will produce applications configured to run on JBoss AS. These instructions will show the steps needed to get it to run on Weblogic.

`seam-gen` was build for simplicity so, as you can imagine, deploying an application generated by `seam-gen` to Weblogic 10.x is not too hard. Basically it consists of updating or removing some configuration files, and adding dependent jars that Weblogic 10.x does not ship with.

This example will cover the basic `seam-gen` `WAR` deployment. This will demonstrate Seam POJO components, Hibernate JPA, Facelets, Drools security, RichFaces, and a configurable dataSource.

39.4.1. Running seam-gen setup

The first thing we need to do it tell seam-gen about the project we want to make. This is done by running ./seam setup in the base directory of the Seam distribution. Note the paths here are my own, feel free to change for you environment.

```
./seam setup
Buildfile: build.xml

init:

setup:
     [echo] Welcome to seam-gen :-)
    [input] Enter your Java project workspace (the directory that contains your
Seam projects) [C:/Projects] [C:/Projects]
/home/jbalunas/workspace
    [input] Enter your JBoss home directory [C:/Program Files/jboss-4.2.3.GA]
[C:/Program Files/jboss-4.2.3.GA]
/jboss/apps/jboss-4.2.3.GA
    [input] Enter the project name [myproject] [myproject]
weblogic-example
     [echo] Accepted project name as: weblogic_example
    [input] Select a RichFaces skin (not applicable if using ICEFaces) [blueSky]
 ([blueSky], classic, ruby, wine, deepMarine, emeraldTown, sakura, DEFAULT)

    [input] Is this project deployed as an EAR (with EJB components) or a WAR
(with no EJB support) [ear]  ([ear], war, )
war
    [input] Enter the Java package name for your session beans [org.jboss.seam.
tutorial.weblogic.action] [org.jboss.seam.tutorial.weblogic.action]
org.jboss.seam.tutorial.weblogic.action
    [input] Enter the Java package name for your entity beans [org.jboss.seam.
tutorial.weblogic.model] [org.jboss.seam.tutorial.weblogic.model]
org.jboss.seam.tutorial.weblogic.model
    [input] Enter the Java package name for your test cases [org.jboss.seam.
tutorial.weblogic.action.test] [org.jboss.seam.tutorial.weblogic.action.test]
org.jboss.seam.tutorial.weblogic.test
    [input] What kind of database are you using? [hsql]  ([hsql], mysql, oracle,
 postgres, mssql, db2, sybase, enterprisedb, h2)

    [input] Enter the Hibernate dialect for your database [org.hibernate.
dialect.HSQLDialect] [org.hibernate.dialect.HSQLDialect]

    [input] Enter the filesystem path to the JDBC driver jar [/tmp/seamlib/hsqldb.jar]
```

```
[/tmp/seam/lib/hsqldb.jar]

    [input] Enter JDBC driver class for your database [org.hsqldb.jdbcDriver]
 [org.hsqldb.jdbcDriver]

    [input] Enter the JDBC URL for your database [jdbc:hsqldb:.] [jdbc:hsqldb:.]

    [input] Enter database username [sa] [sa]

    [input] Enter database password [] []

    [input] Enter the database schema name (it is OK to leave this blank) [] []

    [input] Enter the database catalog name (it is OK to leave this blank) [] []

    [input] Are you working with tables that already exist in the database? [n]
 (y, [n], )

    [input] Do you want to drop and recreate the database tables and data in
import.sql each time you deploy? [n]  (y, [n], )

    [input] Enter your ICEfaces home directory (leave blank to omit ICEfaces) [] []

[propertyfile] Creating new property file:
/rhdev/projects/jboss-seam/cvs-head/jboss-seam/seam-gen/build.properties
     [echo] Installing JDBC driver jar to JBoss server
     [copy] Copying 1 file to /jboss/apps/jboss-4.2.3.GA/server/default/lib
     [echo] Type 'seam create-project' to create the new project

BUILD SUCCESSFUL
```

Type `./seam new-project` to create your project and `cd /home/jbalunas/workspace/weblogic_example` to see the newly created project.

39.4.2. What to change for Weblogic 10.X

First we change and delete some configuration files, then we update the libraries that are deployed with the application.

39.4.2.1. Configuration file changes

`build.xml`

- Change the default target to `archive`.

```
<project name="weblogic_example" default="archive" basedir=".">
```

`resources/META-INF/persistence-dev.xml`

- Alter the `jta-data-source` to be `seam-gen-ds` (and use this as the `jndi-name` when creating the data source in Weblogic's admin console)
- Change the transaction type to `RESOURCE_LOCAL` so that we can use JPA transactions.

```
<persistence-unit name="weblogic_example" transaction-type="RESOURCE_LOCAL">
```

- Add/modify the properties below for Weblogic support:

```
<property name="hibernate.cache.provider_class"
   value="org.hibernate.cache.HashtableCacheProvider"/>
<property name="hibernate.transaction.manager_lookup_class"
   value="org.hibernate.transaction.WeblogicTransactionManagerLookup"/>
```

- You'll need to alter `persistence-prod.xml` as well if you want to deploy to Weblogic using the prod profile.

`resource/WEB-INF/weblogic.xml`

You will need to create this file and populate it following *description of WEB-INF/weblogic.xml [628]*.

`resource/WEB-INF/components.xml`

We want to use JPA transactions so we need to add the following to let Seam know.

```
<transaction:entity-transaction entity-manager="#{entityManager}"/>
```

You will also need to add the transaction namespace and schema location to the top of the document.

```
xmlns:transaction="http://jboss.com/products/seam/transaction"
```

```
http://jboss.com/products/seam/transaction      http://jboss.com/products/seam/transaction-2.2.xsd
```

`resource/WEB-INF/web.xml`

`WEB-INF/web.xml` — Because the `jsf-impl.jar` is not in the `WAR` this listener need to be configured :

```
<listener>
  <listener-class>com.sun.faces.config.ConfigureListener</listener-class>
</listener>
```

`resources/WEB-INF/jboss-web.xml`

You can delete this file as we aren't deploying to JBoss AS (`jboss-app.xml` is used to enable classloading isolation in JBoss AS)

`resources/*-ds.xml`

You can delete these files as we aren't deploying to JBoss AS. These files define datasources in JBoss AS, in Weblogic we will use the administration console.

39.4.2.2. Library changes

The `seam-gen` application has very similar library dependencies as the `jpa` example above. See *Section 39.3.2, « What's different with Weblogic 10.x »*. Below is the changes that are needed to get them in this application.

- build.xml — Now we need to adjust the `build.xml`. Find the target `war` and add the following to the end of the target.

```
<copy todir="${war.dir}/WEB-INF/lib">
  <fileset dir="${lib.dir}">
    <!-- Misc 3rd party -->
    <include name="commons-logging.jar" />
    <include name="dom4j.jar" />
    <include name="javassist.jar" />
    <include name="cglib.jar" />
    <include name="antlr.jar" />

    <!-- Hibernate -->
    <include name="hibernate.jar" />
    <include name="hibernate-commons-annotations.jar" />
```

```
        <include name="hibernate-annotations.jar" />
        <include name="hibernate-entitymanager.jar" />
        <include name="hibernate-validator.jar" />
        <include name="jboss-common-core.jar" />
        <include name="concurrent.jar" />
      </fileset>
    </copy>
```

39.4.3. Building and Deploying your application

All that's left is deploying the application. This involves setting up a data source, building the app, and deploying it.

39.4.3.1. Setting up the data source

Configuring the datasource is very similar to the jee5 *Section 39.2.2.1, « Setting up the hsql datasource »*. Except for what is listed here follow that instruction from the link.

- DataSource Name: `seam-gen-ds`
- JNDI Name: `seam-gen-ds`

39.4.3.2. Building the application

This is as easy as typing `ant` in the projects base directory.

39.4.3.3. Deploying the example

When we installed Weblogic following *Section 39.1.2, « Creating your Weblogic domain »* we chose to have the domain in development mode. This means to deploy the application all we need to do is copy it into the autodeploy directory.

```
cp ./dist/weblogic_example.war /jboss/apps/bea/user_projects/domains/seam_examples/autodeploy
```

Check out the application at the following `http://localhost:7001/weblogic_example/`.

Seam on IBM's WebSphere AS v7

40.1. WebSphere AS environment and version recommendation

WebSphere Application Server v7 is IBM's application server offering. This release is fully Java EE 5 certified.

WebSphere AS being a commercial product, we will not discuss the details of its installation. At best, we will instruct you to follow the directions provided by your particular installation type and license.

First, we will go over some basic considerations on how to run Seam applications under WebSphere AS v7. We will go over the details of these steps using the JEE5 booking example. We will also deploy the JPA (non-EJB3) example application.

All of the examples and information in this chapter are based on WebSphere AS v7. A trial version can be downloaded here : *WebSphere Application Server V7* [http://www.ibm.com/developerworks/downloads/ws/was]

WebSphere v7.0.0.5 is the minimal version of WebSphere v7 to use with Seam. WAS v7.0.0.9 is highly recommended. Earlier versions of WebSphere have bugs in the EJB container that will cause various exceptions to occur at runtime.

Note

You may encounter two exceptions with Seam on WebSphere v7.0.0.5 :

`EJBContext may only be looked up by or injected into an EJB`
: This is a bug in WebSphere v7.0.0.5. WebSphere does not conform to the EJB 3.0 specs as it does not allow to perform a lookup on "java:comp/EJBContext" in callback methods.

  This problem is associated with APAR PK98746 at IBM and is corrected in v7.0.0.9.

`NameNotFoundException: Name "comp/UserTransaction" not found in context "java:"`
: Another bug in WebSphere v7.0.0.5. This occurs when an HTTP session expires. Seam correctly catches the exception when necessary and performs the correct actions in these cases. The problem is that even if the exception is handled by Seam, WebSphere prints the traceback of the exception in SystemOut. Those messages are harmless and can safely be ignored.

> This problem is associated with APAR PK97995 at IBM and is corrected in v7.0.0.9.

The following sections in this chapter assume that WebSphere is correctly installed and is functional, and a WebSphere "profile" has been successfully created.

This chapter explains how to compile, deploy and run some sample applications in WebSphere. These sample applications require a database. WebSphere comes by default with a set of sample applications called "Default Application". This set of sample applications use a Derby database running on the Derby instance installed within WebSphere. In order to keep this simple we'll use this Derby database created for the "Default Applications". However, to run the sample application with the Derby database "as-is", a patched Hibernate dialect must be used (The patch changes the default "auto" key generation strategy) as explained in *Chapitre 41, Seam avec le serveur d'application GlassFish*. If you want to use another database, it's just a matter of creating a connection pool in WebSphere pointing to this database, declare the correct Hibernate dialect and set the correct JNDI name in `persistence.xml`.

40.2. Configuring the WebSphere Web Container

This step is mandatory in order to have Seam applications run with WebSphere v7. Two extra properties must be added to the Web Container. Please refer to the IBM WebSphere Information Center for further explanations on those properties.

To add the extra properties:

- Open the WebSphere administration console
- Select the `Servers/Server Types/WebSphere Application Servers` in the left navigation menu
- Click on the server name (`server1`)
- On the right navigation menu, select `Web Container Settings/Web container`
- On the right navigation menu, select `custom properties` and add the following properties:
  - `prependSlashToResource = true`
  - `com.ibm.ws.webcontainer.invokefilterscompatibility = true`
- Save the configuration and restart the server

40.3. Seam and the WebSphere JNDI name space

In order to use component injection, Seam needs to know how to lookup for session beans bound to the JNDI name space. Seam provides two mechanisms to configure the way it will search for such resources:

- The global `jndi-pattern` switch on the `<core:init>`tag in `components.xml`. The switch can use a special placeholder "`#{ejbName}`" that resolves to the unqualified name of the EJB
- The `@JndiName` annotation

Section 30.1.5, « Integrating Seam with your EJB container » gives detailed explanations on how those mechanisms work.

By default, WebSphere will bind session beans in its local JNDI name space under a "short" binding name that adheres to the following pattern `ejblocal:<package.qualified.local.interface.name>`.

For a detailed description on how WebSphere v7 organizes and binds EJBs in its JNDI name spaces, please refer to the WebSphere Information Center.

As explained before, Seam needs to lookup for session bean as they appear in JNDI. Basically, there are three strategies, in order of complexity:

- Specify which JNDI name Seam must use for each session bean using the `@JndiName` annotation in the java source file,
- Override the default session bean names generated by WebSphere to conform to the `jndi-pattern` attribute,
- Use EJB references.

40.3.1. Strategy 1: Specify which JNDI name Seam must use for each Session Bean

This strategy is the simplest and fastest one regarding development. It uses the WebSphere v7 default binding mechanism. To use this strategy:

- Add a `@JndiName("ejblocal:<package.qualified.local.interface.name>)` annotation to each session bean that is a Seam component.
- In `components.xml`, add the following line:

```
<core:init jndi-name="java:comp/env/#{ejbName}" />
```

- Add a file named `WEB-INF/classes/seam-jndi.properties` in the web module with the following content:

```
com.ibm.websphere.naming.hostname.normalizer=com.ibm.ws.naming.util.DefaultHostnameNormalizer
java.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory
com.ibm.websphere.naming.name.syntax=jndi
com.ibm.websphere.naming.namespace.connection=lazy
com.ibm.ws.naming.ldap.ldapinitctxfactory=com.sun.jndi.ldap.LdapCtxFactory
```

```
com.ibm.websphere.naming.jndicache.cacheobject=populated
com.ibm.websphere.naming.namespaceroot=defaultroot
com.ibm.ws.naming.wsn.factory.initial=com.ibm.ws.naming.util.WsnInitCtxFactory
com.ibm.websphere.naming.jndicache.maxcachelife=0
com.ibm.websphere.naming.jndicache.maxentrylife=0
com.ibm.websphere.naming.jndicache.cachename=providerURL
java.naming.provider.url=corbaloc:rir:/NameServiceServerRoot
java.naming.factory.url.pkgs=com.ibm.ws.runtime:com.ibm.ws.naming
```

- At the end of `web.xml`, add the following lines:

```
<ejb-local-ref>
  <ejb-ref-name>EjbSynchronizations</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home></local-home>
  <local>org.jboss.seam.transaction.LocalEjbSynchronizations</local>
 </ejb-local-ref>
```

That's all folks! No need to update any file during the development, nor to define any EJB to EJB or web to EJB reference!

Compared to the other strategies, this strategy has the advantage to not have to manage any EJBs reference and also to not have to maintain extra files. The only drawback is one extra line in the java source code with the `@JndiName` annotation

40.3.2. Strategy 2: Override the default names generated by WebSphere

There is no simple way to globally override the default naming strategy for session beans in WebSphere. However, WebSphere provides a way to override the name of each bean.

To use this strategy:

- Add a file named `META-INF/ibm-ejb-jar-bnd.xml` in the EJB module and add an entry for each session bean like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar-bnd
  xmlns="http://websphere.ibm.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://websphere.ibm.com/xml/ns/javaee
                      http://websphere.ibm.com/xml/ns/javaee/ibm-ejb-jar-bnd_1_0.xsd"
  version="1.0">
```

```
 <session name="AuthenticatorAction" simple-binding-name="AuthenticatorAction" />
 <session name="BookingListAction" simple-binding-name="BookingListAction" />

</ejb-jar-bnd
>
```

WebSphere will then bind the `AuthenticatorAction` EJB to the `ejblocal:AuthenticatorAction` JNDI name

- In `components.xml`, add the following line:

```
<core:init jndi-name="ejblocal:#{ejbName}" />
```

- Add a file named `WEB-INF/classes/seam-jndi.properties` as described in strategy 1
- In `web.xml`, add the following lines (Note the different `ejb-ref-name` value):

```
<ejb-local-ref>
  <ejb-ref-name>ejblocal:EjbSynchronizations</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home></local-home>
  <local>org.jboss.seam.transaction.LocalEjbSynchronizations</local>
 </ejb-local-ref>
```

Compared to the first strategy, this strategy requires to maintain an extra file (`META-INF/ibm-ejb-jar-ext.xml`), where a line must be added each time a new session bean is added to the application), but still does not require to maintain EJB reference between beans.

40.3.3. Strategy 3: Use EJB references

This strategy is based on the usage of EJB references, from EJB to EJB and from the web module to EJB. To use it:

- In `components.xml`, add the following line:

```
<core:init jndi-name="java:comp/env/#{ejbName}" />
```

- Follow the instructions in *Section 30.1.5, « Integrating Seam with your EJB container »* to declare the references from web to EJB and from EJB to EJB

This is the most tedious strategy as each session bean referenced by another session bean (i.e. "injected") as to be declared in `ejb-jar.xml` file. Also, each new session bean has to be added to the list of referenced bean in `web.xml`

40.4. Configuring timeouts for Stateful Session Beans

A timeout value has to be set for each stateful session bean used in the application because stateful bean must not expire in WebSphere while Seam might still need them. At the time of writing this document, WebSphere does not provide a way to configure a global timeout at neither the cluster, server, application nor ejb-jar level. It has to be done for each stateful bean individually. By default, the default timeout is 10 minutes. This is done by adding a file named `META-INF/ibm-ejb-jar-ext.xml` in the EJB module, and declare the timeout value for each bean:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar-ext
  xmlns="http://websphere.ibm.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://websphere.ibm.com/xml/ns/javaee
                      http://websphere.ibm.com/xml/ns/javaee/ibm-ejb-jar-ext_1_0.xsd"
  version="1.0">

  <session name="BookingListAction"><time-out value="605"/></session>
  <session name="ChangePasswordAction"><time-out value="605"/></session>

</ejb-jar-ext>
```

The `time-out` is expressed in seconds and must be higher than the Seam conversation expiration timeout and a few minutes higher than the user's HTTP session timeout (The session expiration timeout can trigger a few minutes after the number of minutes declared to expire the HTTP session).

40.5. The `jee5/booking` example

The`jee5/booking`example is based on the Hotel Booking example (which runs on JBoss AS). Out of the box, it is designed to run on Glassfish, but with the following steps, it can be deployed on WebSphere. It is located in the `$SEAM_DIST/examples/jee5/booking` directory.

The example already has a breakout of configurations and build scripts for WebSphere. First thing, we are going to do is build and deploy this example. Then we'll go over some key changes that we needed.

The tailored configuration files for WebSphere use the second JNDI mapping strategy ("Override the default names generated by WebSphere") as the goal was to not change any java code to add the `@JndiName` annotation as in the first strategy.

40.5.1. Building the `jee5/booking` example

Building it only requires running the correct ant command:

```
ant -f build-websphere7.xml
```

This will create container specific distribution and exploded archive directories with the `websphere7` label.

40.5.2. Deploying the `jee5/booking` example

The steps below are for the WAS version stated above.The ports are the default values, if you changed them, you must substitute the values.

1. Log in to the administration console

```
http://localhost:9060/admin
```

   Enter your userid annd/or your password if security is enabled for the console.

2. Go to the `WebSphere enterprise applications` menu option under the `Applications --> Application Type` left side menu.
3. At the top of the `Enterprise Applications` table select `Install`. Below are installation wizard pages and what needs to done on each:
   - `Preparing for the application installation`
     - Browse to the `examples/jee5/booking/dist-websphere7/jboss-seam-jee5.ear` file using the file upload widget.
     - Select the `Next` button.
     - Select the `Fast Path` button.
     - Select the `Next` button.
   - `Select installation options`
     - Select the "`Allow EJB reference targets to resolve automatically`" check boxes at the bottom of the page. This will let WebSphere use its simplified JNDI reference mapping.
     - Select the `Next` button.
   - `Map modules to servers`
     - No changes needed here as we only have one server. Select the `Next`button.
   - `Map virtual hosts for Web modules`
     - No changes needed here as we only have one virtual host. Select the `Next`button.

- `Summary`
  - No changes needed here. Select the `Finish` button.
- `Installation`
  - Now you will see WebSphere installing and deploying your application.
  - When done, select the `Save` link and you will be returned to the `Enterprise Applications` table.
- To start the application, select the application in the list, then click on the `Start` button at the top of the table.

4. You can now access the application at `http://localhost:9080/seam-jee5-booking`

40.5.3. Deviation from the original base files

Below are the differences between the base configuration files and the WebSphere specific files held in the `resources-websphere7` directory.

- `META-INF/ejb-jar.xml` — Removed all the EJB references
- `META-INF/ibm-ejb-jar-bnd.xml` — This WebSphere specific file has been added as we use the second JNDI mapping strategy. It defines, for each session bean, the name WebSphere will use to bind it in its JNDI name space
- `META-INF/ibm-ejb-jar-ext.xml` — This WebSphere specific file defines the timeout value for each stateful bean
- `META-INF/persistence.xml` — The main changes here are for the datasource JNDI path, switching to the WebSphere transaction manager lookup class, turning off the `hibernate.transaction.flush_before_completion` toggle, and forcing the Hibernate dialect to be `GlassfishDerbyDialect` as we are using the integrated Derby database
- `WEB-INF/components.xml` — the change here is `jndi-pattern` to use `ejblocal:#{ejbname}` as using the second JNDI matching strategy
- `WEB-INF/web.xml` — Remove all the `ejb-local ref` except the one for `EjbSynchronizations` bean. Changed the ref fo this bean to `ejblocal:EjbSynchronizations`
- `import.sql` — due to the cutomized hibernate Derby dialect, the `ID` column can not be populated by this file and was removed.

Also the build procedure has been changed to include the `log4j.jar` file and exclude the `concurrent.jar` and `jboss-common-core.jar` files.

40.6. The `jpa` booking example

This is the Hotel Booking example implemented in Seam POJOs and using Hibernate JPA with JPA transactions. It does not use EJB3.

The example already has a breakout of configurations and build scripts for many of the common containers including WebSphere.

First thing, we are going to do is build and deploy that example. Then we'll go over some key changes that we needed.

40.6.1. Building the jpa example

Building it only requires running the correct ant command:

```
ant websphere7
```

This will create container specific distribution and exploded archive directories with the `websphere7` label.

40.6.2. Deploying the jpa example

Deploying `jpa` application is very similar to the `jee5/booking` example at *Section 40.5.2, « Deploying the jee5/booking example »*. The main difference is, that this time, we will deploy a war file instead of an ear file, and we'll have to manually specify the context root of the application.

Follow the same instructions as for the `jee5/booking` sample. Select the `examples/jpa/dist-websphere7/jboss-seam-jpa.war` file on the first page and on the `Map context roots for Web modules` page (after the `Map virtual host for Web module`), enter the context root you want to use for your application in the `Context Root` input field.

When started, you can now access the application at the `http://localhost:9080/<context root>`.

40.6.3. Deviation from the generic base files

Below are the configuration file differences between the base configuration files and the files customized for WebSphere held in the `resources-websphere7` directory.

- `META-INF/persistence.xml` — The main changes here are for the datasource JNDI path, switching to the WebSphere transaction manager look up class, turning off the `hibernate.transaction.flush_before_completion` toggle, and forcing the Hibernate dialect to be `GlassfishDerbyDialect` how as using the integrated Derby database
- `import.sql` — due to the customized hibernate Derby dialect, the `ID` column can not be populated by this file and was removed.

Also the build procedure have been changed to include the `log4j.jar` file and exclude the `concurrent.jar` and `jboss-common-core.jar` files.

Seam avec le serveur d'application GlassFish

GlassFish est un serveur d'application open source qui implément complètement Java EE 5. La dernière version stable est v2 UR2.

En premier, nous allons parler de l'environement FlassFish. Ensuite nous allons passer sur comment vous allez déployer l'exemple JEE5. Enfin, nous allons déployer l'application d'exemple JPA. Au final, nous allons vor comment obtenir une application générée par seam-gen pour une exécution sur GlassFish.

41.1. L'environement GlassFish et les informations de déploiement

41.1.1. Installation

Tous les exemples et les informations de ce chapitre sont basés sur la dernière version de GlassFish au moment de son écriture.

- *GlassFish v2 UR2 - page de téléchagement* [https://glassfish.dev.java.net/downloads/v2ur2-b04.html]

Après le téléchargement de GlassFish, installez le:

```
$ java -Xmx256m -jar glassfish-installer-v2ur2-b04-linux.jar
```

Après l'installation, configurez GlassFish:

```
$ cd glassfish; ant -f setup.xml
```

Le nom de domaine créé est `domain1`.

Ensuite, nous démarrons le serveur JavaDB embarqué:

```
$ bin/asadmin start-database
```

Note

JavaDB est une base de données embarqué qui est inclue dans dans GlassFish, tout comme HSQLDB est inclue dans JBoss AS.

Maintenant, démarrez le serveur GlassFish:

```
$ bin/asadmin start-domain domain1
```

La console web d'administration est disponible à l'adresse `http://localhost:4848/`. Vous pouvez accéder à la console web d'administration avec le nom d'utilisateur par défaut (`admin`) et le mot de passe (`adminadmin`). Nous allons utiliser la console d'administration pour déployer nos exemples. Vous pouvez aussi copier les fichiers EAR/WAR dans le dossier `glassfish/domains/domain1/autodeploy` pour les déployer, mais nous n'allons pas parler de cela.

Vous pouvez arrêter le serveur et la base de données en utilisant:

```
$ bin/asadmin stop-domain domain1; bin/asadmin stop-database
```

41.2. L'exemple `jee5/booking`

L'exemple `jee5/booking` est basé sur l'exemple de réservation d'hotel (qui s'exécute sur JBoss AS). Par défaut, il est aussi prévu de fonctionner sur GlassFish. Il ets localisé dans `$SEAM_DIST/examples/jee5/booking`.

41.2.1. Compilation de l'exemple `jee5/booking`

Pour construire l'exemple, exécuter simplement la cible par défaut `ant`:

```
$ ant
```

dans le dossier `examples/jee5/booking`. Ceci va créer les dossiers `dist` et `exploded-archives`.

41.2.2. Le déploiement de l'application dans GlassFish

Nous allons déployer l'application dans GlassFish en utilisant la console d'administration de GlassFish.

1. Connectez vous sur la console d'administration à l'adresse `http://localhost:4848`

2. Accédez à Enterprise Applications dans le menu option sous Applications dans le menu à gauche.
3. En haut, le tableau Enterprise Application sélectionnez Deploy. Suivez l'assistant, en utilisant ces indications:
   - La préparation de l'installation de l'application
     - Navigers vers examples/jee5/booking/dist/jboss-seam-jee5.ear.
     - Sélectionnez le bouton OK.
4. Vous pouvez maintenant accéder à l'application sur http://localhost:8081/seam-jee5/.

41.3. L'exemple de réservation jpa

Ceci est l'exemple de la réservation d'hotel en implémenté en POJOs de Seam et utilisant les transactions JPA avec d'Hibernate JPA. Il ne nécéssite par le support de EJB3 pour s'exécuter sur le serveur d'application.

Cet exemple à déjà à disposition les configurations et les scripts de compilation pour la plus part des containeurs y compris GlassFish.

41.3.1. La construction de l'exemple jpa

Pour compiler l'exemple, utilisez la cible glassfish:

```
$ ant glassfish
```

Ceci va construire pour le containeur spécifique les dossiers dist-glassfish and exploded-archives-glasfish

41.3.2. Le déploiement de l'exemple jpa

Ceci est très similaire à l'exemple jee5 à voir *Section 41.2.2, « Le déploiement de l'application dans GlassFish »* exception faire que c'est un war et non un ear.

- Connectez vous à la console d'administraiton:

```
http://localhost:4848
```

- Accédez au Web Applications dans le menu d'option sous Applications dans le menu de gauche.
  - La préparation de l'installation de l'application
    - Navigez vers examples/jpa/dist-glassfish/jboss-seam-jpa.war.

- Sélectionnez le bouton OK.

- Vous pouvez maintenant accéder à l'application sur `http://localhost:8081/jboss-seam-jpa/`.

En utilisant Derby au lieur d'Hypersonic SQL DB

Pour que l'application fonctionne immédiatement avec GlassFish, nous avons utilisé la base de données Derby (aussi connu JavaDB) dans GlassFish. Cependant, nous recommandons particulièrement d'utiliser une autre base de données (par exemple HSQL). `examples/jpa/resources-glassfish/WEB-INF/classes/GlassfishDerbyDialect.class` est un truc pour contourner un bug de Derby dans le serveur GlassFish. Vous devez l'utiliser comme dialecte Hibernate, si vous utilisez Derby avec GlassFish.

41.3.3. Ce qui est différent pour GlassFish v2 UR2

- Configuration file changes
  - `META-INF/persistence.xml` — le principal changement nécéssaire est le datasource JNDI, le basculement vers la classe de scrutation du gestionnaire de transaction de GlassFish , et en changeant le dialecte d'hibernate pour `GlassfishDerbyDialect`.
  - `WEB-INF/classes/GlassfishDerbyDialect.class` — cette classe est nécesssaire pour que le dialecte d'Hibernate soit changé en `GlassfishDerbyDialect`
  - `import.sql` — tout comme pour le dialecte la colonne `ID` de Derby DB ne peut être remplie par ce fichier et a été retiré.

41.4. Le déploiement d'une application générée par `seam-gen` sur GlassFish v2 UR2

`seam-gen` est un outil très utile pour les développeurs qui rapidement ont une application opérationnelle et s'exécutant et fourni des fondations pour ajouter vos fonctionnalités. Par défaut`seam-gen` va poduire des applications configurés pour s'exécuter sur JBoss AS. Ces instructions vont montrer les étapes nécéssaire poiyr les avoir pour GlassFish.

41.4.1. Exécution du configurateur `seam-gen`

La première étape est de configurer `seam-gen` pour construire le projet de base. Ils ya plusieurs choix à faire plus bas, particulièrement poiyr le datasource et les valeurs d'hibernate que nous allons modifier une fois que le projet a été créé.

```
$ ./seam setup
Buildfile: build.xml
```

```
init:

setup:
[echo] Welcome to seam-gen :-)
[input] Enter your Java project workspace (the directory that contains your
Seam projects) [C:/Projects] [C:/Projects]
/projects
[input] Enter your JBoss home directory [C:/Program Files/jboss-4.2.3.GA]
[C:/Program Files/jboss-4.2.3.GA]

[input] Enter the project name [myproject] [myproject]
seamgen_example
[echo] Accepted project name as: seamgen_example
[input] Do you want to use ICEfaces instead of RichFaces [n] (y, [n])

[input] skipping input as property icefaces.home.new has already
been set.
[input] Select a RichFaces skin [blueSky] ([blueSky], classic, ruby, wine,
deepMarine, emeraldTown, japanCherry, DEFAULT)

[input] Is this project deployed as an EAR (with EJB components) or a WAR
(with no EJB support) [ear] ([ear], war)

[input] Enter the Java package name for your session beans
[com.mydomain.seamgen_example] [com.mydomain.seamgen_example]
org.jboss.seam.tutorial.glassfish.action
[input] Enter the Java package name for your entity beans
[org.jboss.seam.tutorial.glassfish.action]
[org.jboss.seam.tutorial.glassfish.action]
org.jboss.seam.tutorial.glassfish.model
[input] Enter the Java package name for your test cases
[org.jboss.seam.tutorial.glassfish.action.test]
[org.jboss.seam.tutorial.glassfish.action.test]
org.jboss.seam.tutorial.glassfish.test
[input] What kind of database are you using? [hsql] ([hsql], mysql, oracle,
postgres, mssql, db2, sybase, enterprisedb, h2)

[input] Enter the Hibernate dialect for your database
[org.hibernate.dialect.HSQLDialect]
[org.hibernate.dialect.HSQLDialect]

[input] Enter the filesystem path to the JDBC driver jar
[/tmp/seam/lib/hsqldb.jar] [/tmp/seam/lib/hsqldb.jar]
```

```
[input] Enter JDBC driver class for your database [org.hsqldb.jdbcDriver]
[org.hsqldb.jdbcDriver]

[input] Enter the JDBC URL for your database [jdbc:hsqldb:.]
[jdbc:hsqldb:.]

[input] Enter database username [sa] [sa]

[input] Enter database password [] []

[input] Enter the database schema name (it is OK to leave this blank) [] []

[input] Enter the database catalog name (it is OK to leave this
blank) [] []

[input] Are you working with tables that already exist in the database? [n]
(y, [n])

[input] Do you want to drop and recreate the database tables and data in
import.sql each time you deploy? [n] (y, [n])

[propertyfile] Creating new property file:
/home/mnovotny/workspaces/jboss/jboss-seam/seam-gen/build.properties
[echo] Installing JDBC driver jar to JBoss server
[copy] Copying 1 file to
/home/mnovotny/workspaces/jboss/jboss-seam/seam-gen/C:/Program
Files/jboss-4.2.3.GA/server/default/lib
[echo] Type 'seam create-project' to create the new project

BUILD SUCCESSFUL
Total time: 4 minutes 5 seconds
```

Entrez `$ ./seam new-project` pour créer votre projet et ensuite `cd /projects/seamgen_example` pour aller dans la structure nouvellement créée.

41.4.2. Les modifications nécéssaires pour le déploiement dans GlassFish

Nous allons maintenant faire quelques modificaitions dans le projet généré.

41.4.2.1. Configuration file changes

`resources/META-INF/persistence-dev.xml`

- Modifiez le `jta-data-source` pour le mettre à `jdbc/__default`. Nous allons utiliser Derby DB intégré dans GlassFish.
- Modifiez les propriétés avec les suivantes. Les différences clefs sont brièvements décrites dans *Section 41.3.3, « Ce qui est différent pour GlassFish v2 UR2 »*:

```
<property name="hibernate.dialect" value="GlassfishDerbyDialect"/>
<property name="hibernate.hbm2ddl.auto" value="update"/>
<property name="hibernate.show_sql" value="true"/>
<property name="hibernate.format_sql" value="true"/>
<property name="hibernate.cache.provider_class"
 value="org.hibernate.cache.HashtableCacheProvider"/>
<property name="hibernate.transaction.manager_lookup_class"
 value="org.hibernate.transaction.SunONETransactionManagerLookup"/>
```

- Nous allons avoir besoin de modifier `persistence-prod.xml` aussi si vous voulez deployer dans GlassFish en utilisant le profile de prod.

`resources/GlassfishDerbyDialect.class`

Tout comme les autres exemples, nous allons avoir besoin d'inclure cette classe pour le support de la BD. Il peut être copie depuis l'exemple `jpa` dans le dossier `seamgen_example/resources`.

```
$ cp \
$SEAM_DIST/examples/jpa/resources-glassfish/WEB-INF/classes/
GlassfishDerbyDialect.class \
./resources
```

`resources/META-INF/jboss-app.xml`

Vous pouvez effacer ce fichier car nous n'allons pas déployer vers JBoss AS (`jboss-app.xml`est utilisé pour activer l'isolation du chargement des classes dans JBoss AS)

`resources/*-ds.xml`

Vous pouvez effacer ces fichiers nous n'allons pas déployer vers JBoss AS (ces fichiers définissent des sources de données dans JBoss AS, nous allons utiliser la source de données par défaut de GlassFish)

`resources/WEB-INF/components.xml`

- Active l'intégration des transactions du containeur géré - ajoutez le composant `<transaction:ejb-transaction/>` et sa déclaration d'espace de nom `xmlns:transaction="http://jboss.com/products/seam/transaction"`
- Modifiez le `jndi-pattern` à `java:comp/env/seamgen_example/#{ejbName}`

`resources/WEB-INF/web.xml`

Avez l'exemple `jee5/booking`, nous allons avoir besoin d'ajouter les références EJB references à web.xml. Techniquement, le type de référence n'est pas requi, mais nous l'ajoutons ici pour faire bonne figure. Notez que ces références nécéssitent la présence d'un élément vide `local-home` pour conserver la compatibilité avec un déploiement JBoss AS 4.x.

```
<ejb-local-ref
>
   <ejb-ref-name
>seamgen_example/AuthenticatorAction</ejb-ref-name
>
   <ejb-ref-type
>Session</ejb-ref-type
>
   <local-home/>
   <local
>org.jboss.seam.tutorial.glassfish.action.Authenticator</local
>
 </ejb-local-ref>

 <ejb-local-ref>
   <ejb-ref-name
>seamgen_example/EjbSynchronizations</ejb-ref-name
>
   <ejb-ref-type
>Session</ejb-ref-type>
   <local-home/>
   <local
>org.jboss.seam.transaction.LocalEjbSynchronizations</local>
 </ejb-local-ref
>
```

Gardez à l'esprit que si vous déployer dans JBoss AS 4.x, et que vous avez défini les références EJB comme montrées ci-dessus dans votre web.xml, vous allez avoir aussi besoin de définir les noms JNDI locaux de chacun d'entre eux dans jboss-web.xml, comme montré ci-dessous. Cette étape n'est pas nécéssaire avec le déploiement dans GlassFish, mais elle

est mentionnée ici dans le cas où vous voudriez aussi déployer lapplication dans JBoss AS 4.x (pas nécéssaire pour JBoss AS 5).

```
<ejb-local-ref
>
   <ejb-ref-name
>seamgen_example/AuthenticatorAction</ejb-ref-name
>
   <local-jndi-name
>AuthenticatorAction</local-jndi-name
>
 </ejb-local-ref>

 <ejb-local-ref>
   <ejb-ref-name
>seamgen_example/EjbSynchronizations</ejb-ref-name
>
   <local-jndi-name
>EjbSynchronizations</local-jndi-name>
 </ejb-local-ref
>
```

41.4.2.2. La création de l'EJB `AuthenticatorAction`

Nous voulons prendre le composant POJO de Seam existant `Authenticator` et créer un EJB à l'extérieur de lui.

1. • Renommer la classe en `AuthenticatorAction`
   - Ajoutez l'annotation `@Stateless` à la nouvelle classe `AuthenticatorAction`.
   - Créer un interface appelé `Authenticator` qui implémente `AuthenticatorAction` (EJB3 nécéssite pour les beans de session d'avoir un interface local). Annotez l'interface avec `@Local`, et ajoutez une seule méthode avec la même signature que `authenticate` dans `AuthenticatorAction`.

```
@Name("authenticator")
@Stateless
public class AuthenticatorAction implements Authenticator {
```

```
@Local
public interface Authenticator {
```

```
    public boolean authenticate();
}
```

2. Vous avons déjà ajouté sa référence au fichier `web.xml` donc nous pouvons continuer.

41.4.2.3. Les dépendances jar additionnelles et les autres modification au `build.xml`

Cette application a des besoins similaires à l'exemple `jee5/booking`.

- Modifiez la cible par défaut à `archive` (nous n'allons pas couvrir le déploiement automatique vers GlassFish).

```
<project name="seamgen_example" default="archive" basedir="."
>
```

- Nous alllons avoir besoin de `GlassfishDerbyDialect.class` dans notre jar d'application. Pour faire cela, trouvez la tâche `jar` et ajoutez la ligne `GlassfishDerbyDialect.class` comme montré ci-dessous:

```
<target name="jar" depends="compile,copyclasses" description="Build the distribution .jar
 file">
  <copy todir="${jar.dir}">
    <fileset dir="${basedir}/resources">
      <include name="seam.properties" />
      <include name="*.drl" />
      <include name="GlassfishDerbyDialect.class" />
    </fileset
>
  </copy>
  ...
```

- Maintenant nous allons avoir besoin d'ajouter les jars additionnel dans le fichier `ear`. Regarez dans la section `<copy todir="${ear.dir}/lib" >` de la cible `ear`. Ajoutez ce qui suit à l'enfant de l'élément `<fileset dir="${lib.dir}" >`.

- • Ajoutez les dépendances Hibernate

```
<!-- Hibernate and deps -->
<include name="hibernate.jar"/>
<include name="hibernate-commons-annotations.jar"/>
```

```
  <include name="hibernate-annotations.jar"/>
  <include name="hibernate-entitymanager.jar"/>
  <include name="hibernate-validator.jar"/>
  <include name="jboss-common-core.jar"/>
```

- Ajoutez les dépendances des tierces parties.

```
  <!-- 3rd party and supporting jars -->
  <include name="javassist.jar"/>
  <include name="dom4j.jar"/>
  <include name="concurrent.jar" />
  <include name="cglib.jar"/>
  <include name="asm.jar"/>
  <include name="antlr.jar" />
  <include name="commons-logging.jar" />
  <include name="commons-collections.jar" />
```

Vous pourrions finir avec quelquechose comme:

```
<fileset dir="${lib.dir}">
  <includesfile name="deployed-jars-ear.list" />

  <!-- Hibernate and deps -->
  <include name="hibernate.jar"/>
  <include name="hibernate-commons-annotations.jar"/>
  <include name="hibernate-annotations.jar"/>
  <include name="hibernate-entitymanager.jar"/>
  <include name="hibernate-validator.jar"/>
  <include name="jboss-common-core.jar" />

  <!-- 3rd party and supporting jars -->
  <include name="javassist.jar" />
  <include name="dom4j.jar" />
  <include name="concurrent.jar" />
  <include name="cglib.jar" />
  <include name="asm.jar" />
  <include name="antlr.jar" />
  <include name="commons-logging.jar" />
  <include name="commons-collections.jar" />
</fileset
>
```

41.4.2.4. Compilez et déployez l'application en seam-gen vers GlassFish

- Compilez votre application en appelant `ant` dans le dossier de base de votre projet (par exemple `/projects/seamgen-example`). La cible de cette compilation sera `dist/seamgen-example.ear`.
- Pour déployer l'application suivre les instructions ici *Section 41.2.2, « Le déploiement de l'application dans GlassFish »* mais utiliser les références pour ce projet `seamgen-example` au lieu de `jboss-seam-jee5`.
- Vérifiez l'application sur `http://localhost:8081/seamgen_example/`

Les dépendances

42.1. Les dépendances du JDK

Seam ne fonctionne pas avec le JDK 1.4 et a besoin du JDK 5 ou supérieur car il utilise les annotations et d'autres fonctionnalités du JDK 5.0. Seam a été complètement testé en utilisant les JDK de Sen. Cependant il n'y a pas de bugs connus spécifique à Seam avec d'autres JDKs.

42.1.1. Les considérations sur le JDK6 de Sun

Les premières versions du JDK 6 de Sun contenait une version incompatible de JAXB et nécéssitait son remplacement en utilisant le dossier "endorsed". Le JDK 6 de Sun Update 4 fourni une version à jours de JAXB 2.1 et remplace cette contrainte. Pour la compilation, le test ou l'exécution soyez sur d'utiliser cette version ou une suppérieure.

Seam utilise JBoss Embedded dans ses tests unitaires et d'intégration. Ceci est un prérequis addictionnel avec l'utilisation de JDK 6. Pour exécuter le JBoss Embedded avec le JDK 6, vous allez avoir besoin de définir les arguments de la JVM suivant

```
-Dsun.lang.ClassLoader.allowArraySyntax=true
```

le système de compilation interne de Seam le défini par défaut quand il exécute la suite de tests de Seam. Cependant, si vous utilisez aussi JBoss Embedded pour vos tests vous allez avoir besoin de définir cette valeur.

42.2. Les dépendans de projet

Cette section liste à al fois les dépendances au moment de la compilation et de l'exécution de Seam. Quand le type est listé comme `ear`, la bibliothèque devrait être incluse dans le dossier /lib de votre fichier ear de votre application. Quand le type est listé comme `war`, la bibliothèque devrait être placée dans le dossier `/WEB-INF/lib` de votre fichier war de votre application. L'étendue de la dépendance est soit tout, à l'éxécution, ou fourni (par JBoss AS 4.2 ou 5.0).

L'information de version à jours et l'information complète des dépendances n'est pas inclus dans la documentation, mais c'est fourni dans le `/dependency-report.txt` qui est généré depuis les POMs de Maven stockés dans `/build`. Vous pouvez générer ce fichier en exécutant `ant dependencyReport`.

42.2.1. Noyau

Tableau 42.1.

Nom	Etendue	Type	Notes
`jboss-seam.jar`	all	ear	

Nom	Etendue	Type	Notes
			La bibliothèque du noyau de Seam, toujours requise.
`jboss-seam-debug.jar`	exécution	war	Inclue pendant le développement avec l'activation de la fonctionnalité de débogage de Seam
`jboss-seam-ioc.jar`	exécution	war	Requise avec l'utilisation de Seam avec Spring
`jboss-seam-pdf.jar`	exécution	war	Requise avec la fonctionnalité PDF de Seam
`jboss-seam-excel.jar`	exécution	war	Requise avec l'utilisation de la fonctionnalité Microsoft® Excel® de Seam
`jboss-seam-rss.jar`	exécution	war	Requise avec l'utilisation de la fonctionnalité de génération RSS de Seam
`jboss-seam-remoting.jar`	exécution	war	Requise avec l'utilisation de Seam Remoting
`jboss-seam-ui.jar`	exécution	war	Requise avec l'utilisation des controles JSF de Seam
`jsf-api.jar`	fournie		Les API de JSF
`jsf-impl.jar`	fournie		L'implémentation de référence de JSF
`jsf-facelets.jar`	exécution	war	Les facelets
`urlrewrite.jar`	exécution	war	la bibliothèque de réécriture des URLs
`quartz.jar`	exécution	ear	Requise quand vous voules utiliser Quartz avec la fonctionnalité assynchrone de Seam

42.2.2. Les RichFaces

Tableau 42.2. Les dépendances de RichFaces

Nom	Etendue	Type	Notes
`richfaces-api.jar`	all	ear	Requise pour utiliser les RichFaces. Fourni les classes API que vous pouvez vouloir

Nom	Etendue	Type	Notes
			utiliser depuis votre application, par exemple, pour créer un arbre
`richfaces-impl.jar`	exécution	war	Requise pour utiliser les RichFaces.
`richfaces-ui.jar`	exécution	war	Requise pour utiliser les RichFaces. Fourni tous les composants UI.

42.2.3. Seam Mail

Tableau 42.3. Les dépendances de Seam Mail

Nom	Etendue	Type	Notes
`activation.jar`	exécution	ear	Requise pour le support des pièces jointes
`mail.jar`	exécution	ear	Requise pour le support de l'émission des emails
`mail-ra.jar`	seulement à la compilation		Requise pour le support de la réception des emails mail-ra.rar devrait être déployé sur le serveur d'application à l'exécution
`jboss-seam-mail.jar`	exécution	war	Seam Mail

42.2.4. Seam PDF

Tableau 42.4. Les dépendances de Seam PDF

Nom	Type	Etendue	Notes
`itext.jar`	exécution	war	La bibliothèque de PDF
`jfreechart.jar`	exécution	war	La bibliothèque de diagramme
`jcommon.jar`	exécution	war	Requise par JFreeChart
`jboss-seam-pdf.jar`	exécution	war	La bilbiothèque du noyau de Seam PDF

42.2.5. Seam Microsoft® Excel®

Tableau 42.5. Seam Microsoft® Excel® Dependencies

Nom	Type	Etendue	Notes
`jxl.jar`	exécution	war	La bibliothèque de JExcelAPI

Nom	Type	Etendue	Notes
jboss-seam-excel.jar	exécution	war	Seam Microsoft® Excel® core library

42.2.6. Le support des RSS de Seam

Tableau 42.6. Les dépendances des RSS de Seam

Nom	Type	Etendue	Notes
yarfraw.jar	exécution	war	La bibliothèque de YARFRAW RSS
JAXB	exécution	war	Les bibliothèques d'analyse JAXB XML
http-client.jar	exécution	war	Les bibliothèques de Apache HTTP Client
commons-io	exécution	war	La bibliothèque de Apache commons IO
commons-lang	exécution	war	La bibliothèque de Apache commons lang
commons-codec	exécution	war	La bibliothèque de Apache commons codec
commons-collections	exécution	war	La bibliothèque de Apache commons collections
jboss-seam-rss.jar	exécution	war	La bibliothèque du noyau de RSS de Seam RSS

42.2.7. JBoss Rules

Les bibliothèques de JBoss Rules peuvent être trouvées dans le dossier `drools/lib` dans Seam.

Tableau 42.7. Les dépendances de JBoss Rules

Nom	Etendue	Type	Notes
antlr-runtime.jar	exécution	ear	ANTLR Runtime Library
core.jar	exécution	ear	Eclipse JDT
drools-api.jar	exécution	ear	
drools-compiler.jar	exécution	ear	
drools-core.jar	exécution	ear	
drools-decisiontables.jar	exécution	ear	
drools-templates.jar	exécution	ear	

Nom	Etendue	Type	Notes
janino.jar	exécution	ear	
mvel2.jar	exécution	ear	

42.2.8. JBPM

Tableau 42.8. JBPM dependencies

Nom	Etendue	Type	Notes
jbpm-jpdl.jar	exécution	ear	

42.2.9. GWT

Ces bibliothèques sont requises si vous voulez utiliser le Google Web Toolkit (GWT) avec votre application Seam.

Tableau 42.9. Les dépendances de GWT

Nom	Etendue	Type	Notes
gwt-servlet.jar	exécution	war	Les bibliothèques Servlet de GWT

42.2.10. Spring

Ces bibliothèques sont requise si vous voulez utiliser le Framework de Spring avec votre application Seam.

Tableau 42.10. Les dépendances du Framework de Spring

Nom	Etendue	Type	Notes
spring.jar	exécution	ear	La bibliothèque du Framework de Spring

42.2.11. Groovy

Ces bibliothèques sont requises si vous voulez utiliser Groovy avec votre application Seam.

Tableau 42.11. Les dépendances de Groovy

Nom	Etendue	Type	Notes
groovy-all.jar	exécution	ear	Les bibliothèques de Groovy

42.3. La gestion des dépendances en utilisant Maven

Maven offre le support de la gestion des dépendances de manière transitive et peut être utilisé pour gérer les dépendances de votre projet Seam. Vous pouvez utiliser les tâches Maven Ant

pour intégrer Maven dans vos compilations ANt ou vous pouvez utiliser Maven pour compiler et deployer votre projet.

Nous n'allons pas maintenant discuter de comment utiliser Maven ici, mais parcourir juste quelques POMs simples que vous pouvez utiliser.

Les versions d'étapes de Seam sont disponibles sur *http://repository.jboss.org/maven2* [http://repository.jboss.org/maven2] et les instannées courrantes sont disponibles dans *http://snapshots.jboss.org/maven2* [http://snapshots.jboss.org/maven2].

Tous les artéfactes de Seam sont disponibles dans Maven:

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

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

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

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

```
>jboss-seam-remoting</artifactId>
</dependency
>
```

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

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

Ce POM d'exemple va vous donner Seam, JPA (fourni par Hibernate), Hibernate Validator et Hibernate Search:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  http://maven.apache.org/maven-
v4_0_0.xsd">
  <modelVersion
>4.0.0</modelVersion>
  <groupId
>org.jboss.seam.example/groupId>
  <artifactId
>my-project</artifactId>
  <version
>1.0</version>
  <name
>My Seam Project</name>
  <packaging
>jar</packaging>
```

```
 <repositories>
  <repository>
   <id
>repository.jboss.org</id>
   <name
>JBoss Repository</name>
   <url
>http://repository.jboss.org/maven2</url>
  </repository>
 </repositories>

 <dependencies>

  <dependency>
   <groupId
>org.hibernate</groupId>
   <artifactId
>hibernate-validator</artifactId>
   <version
>3.1.0.GA</version>
  </dependency>

  <dependency>
   <groupId
>org.hibernate</groupId>
   <artifactId
>hibernate-annotations</artifactId>
   <version
>3.4.0.GA</version>
  </dependency>

  <dependency>
   <groupId
>org.hibernate</groupId>
   <artifactId
>hibernate-entitymanager</artifactId>
   <version
>3.4.0.GA</version>
  </dependency>

  <dependency>
    <groupId
>org.hibernate</groupId>
    <artifactId
```

```
>hibernate-search</artifactId>
      <version
>3.1.1.GA</version>
   </dependency>

   <dependency>
     <groupId
>org.jboss.seam</groupId>
     <artifactId
>jboss-seam</artifactId>
     <version
>2.2.0.GA</version>
   </dependency>

 </dependencies>

</project
>
```