13

Configuration EJB3 avancée avec Seam

Au chapitre précédent, vous avez bâti votre application sur EJB3 et JSF et mis en œuvre les innovations apportées par la nouvelle spécification.

Dans cet ultime chapitre, vous reprendrez l'application exemple webstock en partie générée à l'aide de l'atelier EclipseUML d'Omondo. Vous utiliserez pour ce faire le framework Seam, présenté au chapitre 8, mais cette fois appliqué au développement EJB3.

Vous découvrirez les aspects avancés de ce framework, en particulier pour la gestion des contextes, un des points forts de Seam.

Les contextes Seam

Comme expliqué au chapitre 8, JBoss Seam simplifie les échanges entre les différentes couches des applications JEE, en particulier entre la partie présentation, représentée par la technologie JSF, et la partie EJB3, comme illustré à la figure 13.1.

Figure 13.1

Échanges Seam avec les couches JSF et EJB3

La possibilité de gérer plusieurs conversations simultanées est une fonctionnalité remarquable de Seam permise par le support du contexte conversationnel, qui permet de simuler virtuellement des sessions multiples sans avoir à les créer physiquement.

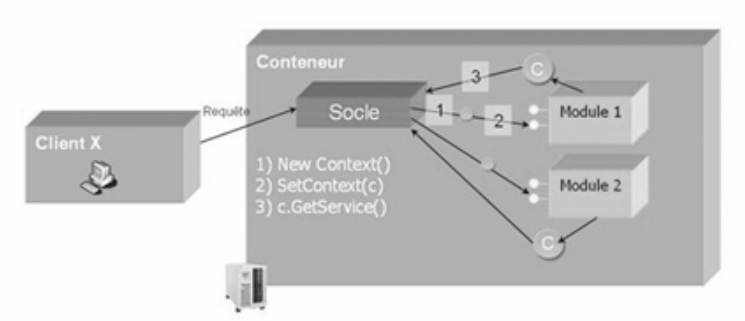
Pour Seam, tous les composants d'une application en cours d'exécution sont des objets avec état (stateful). Ces objets sont souvent des EJB, comme dans notre exemple.

Chaque instance d'un composant est associée à un contexte défini et possède un nom permettant de la référencer.

Par définition, un contexte est une vue sur les composants d'une application à base de conteneur en cours d'exécution, comme l'illustre la figure 13.2.

Figure 13.2

Représentation d'un contexte Seam

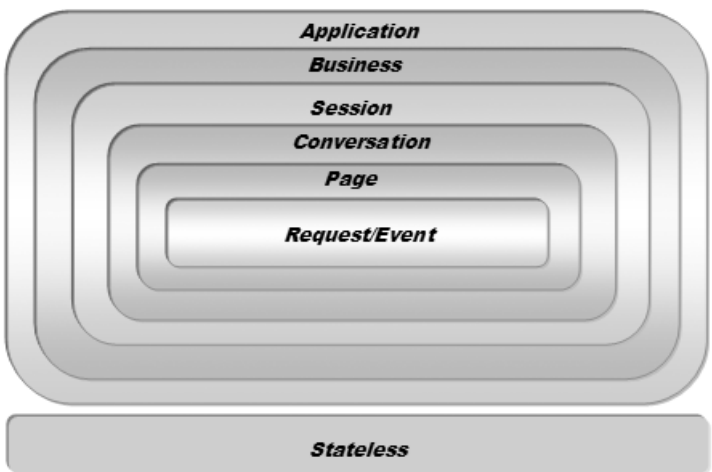


Une application Web classique comporte plusieurs contextes traditionnels, comme Session, Requête et Application. Seam ajoute quatre nouveaux contextes au modèle existant (voir figure 13.3) :

- Stateless
- Conversation
- Page
- Processus métier (Business)

Figure13.3

Modèle contextuel de Seam



Les contextes Seam sont créés et détruits par le conteneur JBoss. Ils ne sont plus contrôlés avec une API Java spécifique, mais gérés d'une manière explicite, déclarée à l'aide des annotation JDK5.

Chaque type de composant Seam possède un contexte par défaut, que le développeur peut modifier à l'aide de l'annotation @Scope.

Le tableau 13.1 récapitule les contextes par défaut de Seam.

Tableau 13.1 Contextes par défaut de Seam

objet	contexte seam par defaut
Beans stateless	STATELESS
Beans entité	CONVERSATION
Beans session stateful	CONVERSATION
JavaBeans (POJO)	EVENT

Rappelons brièvement les rôles des contextes Seam :

- **Stateless.** Les objets sans état vivent dans le contexte Stateless traditionnel d'une application Web. Tout JavaBean peut être désigné comme étant un objet Seam sans état. Ce contexte est utilisé lorsqu'il n'est pas nécessaire de persister l'état du contexte durant plusieurs appels de la classe.
- Voici un exemple de JavaBean helloworld stateless :

```
@Name ("statelessHelloWorld")
@Scope (ScopeType.STATELESS)
public class StatelessHelloWord {
  @RequestParameter ("nom")
  String nom;

  public String getText() {
      return "Hello World ! " + nom ;
   }
}
```

- **Event.** C'est le contexte le plus simple et le plus utilisé dans les applications Web. Les objets enregistrés dans le cadre de ce contexte sont détruits à la fin de la requête. Les JavaBeans sont stockés par défaut dans ce contexte événementiel.
- **Page.** Les composants de ce contexte sont attachés à une page spécifique. Vous avez accès à ces composants pour tout événement émis à partir de cette page.
- **Conversation.** Ce contexte est détaillé ultérieurement dans ce chapitre.
- **Session.** Gère l'état associé à la session utilisateur afin de garder trace de certaines informations relatives à cet utilisateur et des données susceptibles d'être partagées entre différentes conversations.
- **Business.** C'est le contexte associé aux processus métier de longue durée. L'état de ce contexte est géré et persisté par le moteur de workflow JBoss jBPM (JBoss Business Process Management). Rappelons qu'un processus métier est un ensemble d'actions réparties sur différents types d'utilisateurs. L'état de ce contexte est partagé entre ces différents utilisateurs selon une réglementation définie.
- **Application.** Utilisé pour garder les informations statiques relatives à une application donnée, comme les métamodèles ou les objets de référence. Par exemple, Seam persiste ses configurations et métamodèles dans le contexte de l'application.

Gestion des états de composants Seam

Les instances de composants Seam sont associées à une variable de contexte. Le nom du composant est attribué *via* l'annotation @Name et le contexte auquel il sera associé à l'aide de l'annotation @Scope.

Contexte conversationnel

La conversation est un concept bien connu du développement J2EE. JBoss Seam est toutefois le premier framework à l'implémenter complètement et à l'enrichir avec le scope de conversation.

Une conversation peut être vue comme un genre de pas-à-pas. C'est une action qui s'étend sur plusieurs petites actions et qui permet a l'utilisateur de se promener aisément d'avant en arrière, sans se soucier de la perte de ses données, mais sans pour autant les persister dans la base.

Toutes les informations utilisées pendant une conversation sont stockées dans le scope de conversation. La notion de conversation est introduite par Seam à travers le support de ses nouveaux contextes.

Pour Seam, une conversation est une requête utilisateur qui s'étend sur plusieurs pages avant d'être close. Le panier d'achats en ligne en est un exemple. Chaque cycle requête/réponse est aussi une conversation élémentaire entre la page qui expose le formulaire de la requête et la page de réponse. En d'autres termes, une conversation est une unité de travail élémentaire : ce qu'on fait à un moment donné sur une page donnée.

Par défaut, les objets Seam avec état (stateful) ont une conversation de type Scope. Le contexte conversationnel par défaut se résume ainsi : le composant est instancié quand la première page est émise puis est détruit quand la page de réponse est définitivement affichée.

Les composants utilisent des conversations temporaires, qui ont lieu tout au long de la requête avant d'être détruites à l'affichage de la page de réponse.

POJO

Un POJO Seam qui ne porte que l'annotation @Name est un objet avec état et est enregistré dans le contexte conversationnel du conteneur du serveur d'applications.

Mise en œuvre de l'étude de cas avec Seam

Dans cette section, vous allez « refactorer » le code de l'application webstock afin de bien cerner l'intégration de Seam dans l'architecture MVC et la valeur ajoutée de cette plate-forme. Vous pourrez ainsi mesurer à quel point elle allège le contexte de l'application suivant la bonne pratique dite DRY.

DRY (Don't Repeat Yourself)

Le concept DRY vise à éviter les redondances et les duplications de code d'une application afin d'en augmenter la cohérence ainsi que l'évolutivité et la maintenance.

Intégration de Seam dans l'architecture MVC

Le diagramme de séquences illustré à la figure 13.4 illustre un cycle JSF complet intégrant Seam.

Seam utilise différents intercepteurs pour contrôler ses objets à travers les différentes couches de l'application. Le but est de ne pas encombrer l'architecture de l'application en ajoutant des servlets spécifiques, comme le font les autres plates-formes, et de ne pas perturber le cycle JSF.

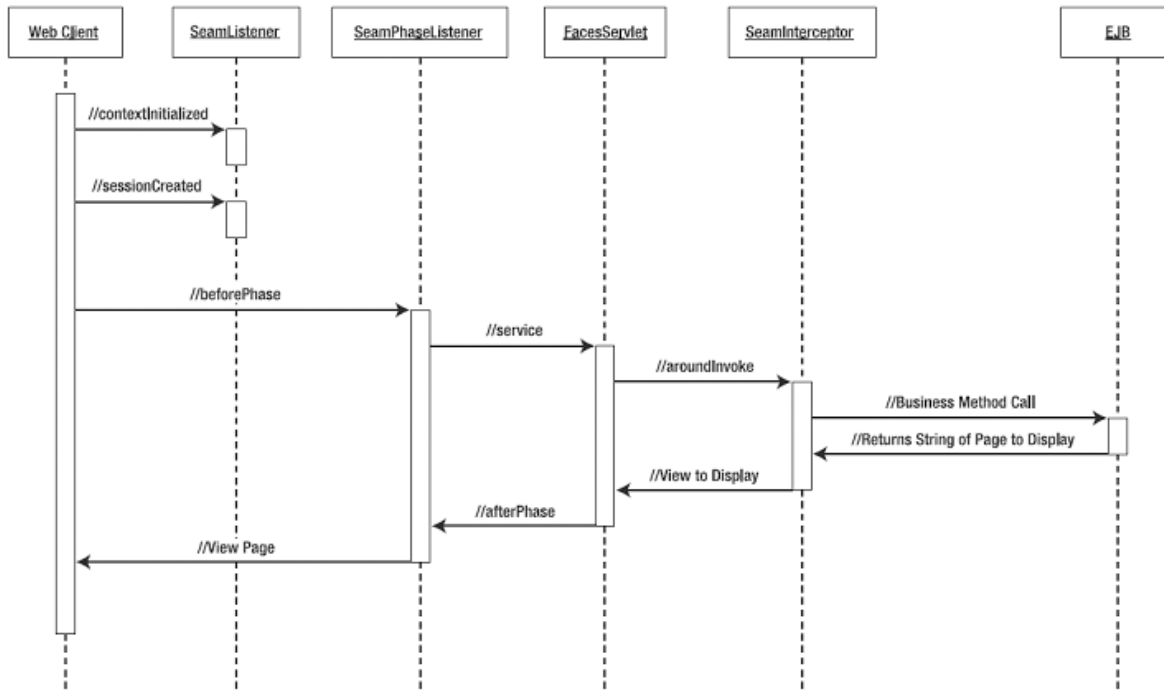


Figure 13.4

Diagramme de séquences du cycle JSF intégrant Seam

Le premier listener appelé est SeamListener (org.jboss.seam.servlet.SeamListener), que vous devez déclarer dans le fichier descripteur de l'application web.xml (voir listing suivant). Ce listener est appelé à chaque instanciation d'une nouvelle session.

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

<!—Faces Servlet Maping -- >
<servlet-mapping>
    <servlet-name>faces Servlet</servlte-name>
    <url-pattern>*.seam</url-pattern>
</servlet-mapping>
```

C'est à travers cet objet que vous invoquerez la servlet FacesServlet. Ce listener fait partie du cycle de vie de la servlet FacesServlet, comme décrit dans l'extrait suivant (notez le package org.andromda.presentation utilisé par AndroMDA pour la génération JSF avec EclipseUML) :

```
<lifecycle>
  <phase-listener> org.andromda.presentation.jsf.MessagePhaseListener
  </phase-listener> org.jboss.seam.jsf.SeamPhaseListener</phase-listener>
</lifecycle>
```

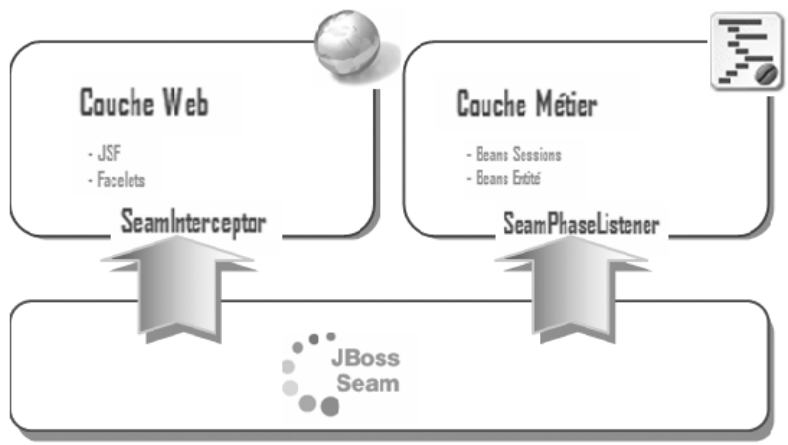
Pour intégrer Seam à la couche métier, vous devrez utiliser SeamInterceptor afin d'intercepter les appels aux EJB et composants métier :

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

La figure 13.5 illustre ce principe d'intégration « d'intercepteurs » à travers les couches Web et métier des applications EJB3/JSF.

Figure 13.5

Intercepteurs Seam



Configuration de Seam

L'installation du framework Seam est décrite en annexe de l'ouvrage. Dans le cadre de ce chapitre, vous aurez besoin des JAR jboss-seam.jar et jboss-seam-ui-jar que vous copiez sous la racine de votre EAR, sans oublier d'ajouter ces dépendances au fichier application.xml :

```
<module>
<java>jboss-seam.jar</java>
</module>
```

Vous disposez à présent d'une structure projet prête à fonctionner avec Seam.

La section suivante reprend certains cas d'utilisation de l'application webstock afin de les adapter à cette plate-forme.

Cas d'utilisation « Ajout d'un nouveau client »

Vous allez reprendre le cas d'utilisation « Ajout d'un nouveau client » de l'application webstock pour l'adapter au contexte Seam (voir la figure 12.11 du chapitre 12 décrivant le diagramme d'état associé à ce cas d'utilisation).

Rappelons que le cycle JSF sans Seam de ce use case se résume ainsi :

1. Une page xHTML présente le formulaire de saisie des paramètres du nouveau client, qui invoque la servlet AjoutNouveauClient.java.
2. La servlet fait appel au contrôleur AjoutClientController.java pour enregistrer les paramètres dans un bean managé sous forme de POJO, qui sert à contenir les différentes valeurs des paramètres saisis sur le formulaire d'ajout.
3. Le contrôleur se charge d'invoquer la méthode métier d'ajout du nouveau client sur le bean session.

La figure 13.6 illustre le processus d'une requête JSF dans le cadre de l'application webstock avant l'intégration de Seam.

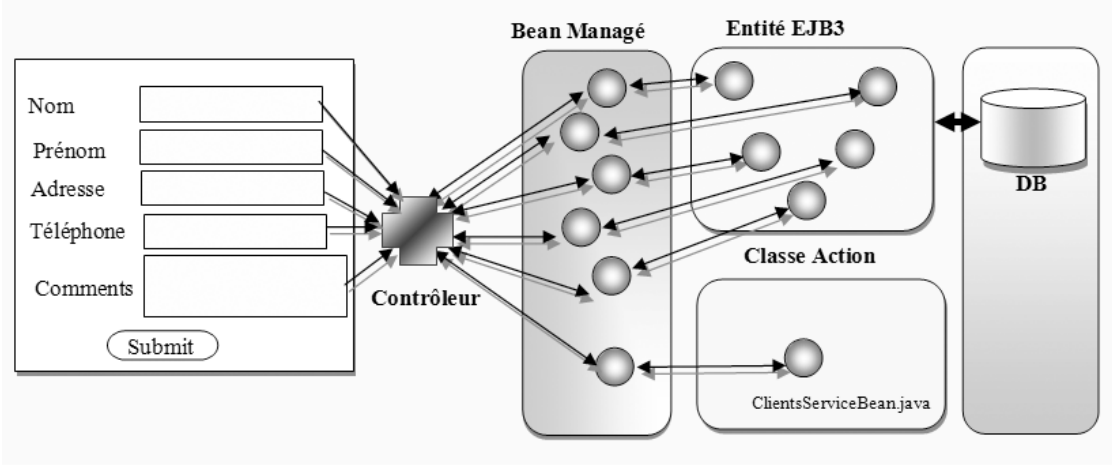


Figure 13.6

Use case « Ajout d'un nouveau client » avant l'intégration de Seam

Vous allez reprendre cette chaîne d'invocation des objets EJB3 afin de la refactorer selon le principe Seam.

La méthodologie Seam stipule qu'il faut supprimer tout bean managé pour « laisser la main » à la plate-forme. Il vous est simplement demandé d'ajouter les annotations nécessaires.

Voici les listing des différentes classes invoquées lors du déroulement du cas d'utilisation après leurs adaptation pour une intégration Seam.

DataModel `Client` :

```
@Name("Client")
@javax.persistence.Entity
@javax.persistence.Table(name = "CLIENT")
@javax.persistence.NamedQuery(name = "Client.findAll", query = "select client from
➥Client AS client")
public class Client
    implements java.io.Serializable, Comparable<Client>
{

    // ----------- Attribute Definitions ------------

    private java.lang.String comments;
    private java.lang.String telephone;
    private java.lang.String adresse;
    private java.lang.String prenom;
    private java.lang.String nom;
    private java.lang.Long id;


    // --------- Relationship Definitions -----------

    private java.util.Set<unnamed.Commande> commande = new java.util.TreeSet
    ➥<unnamed.Commande>();

@NotNull
@Range(min=3, max=100,
message="Prenom must be between 3 and 100")

@javax.persistence.Column(name = "PRENOM", nullable = false, insertable = true,
➥updatable = true)
    public java.lang.String getPrenom()
    {
        return prenom;
    }

    @NotNull
@Pattern(regex="^[a-zA-Z.-]+ [a-zA-Z.-]+",
message="Vous devez un commentaire")


    @javax.persistence.Column(name = "COMMENTS", nullable = false, insertable
    ➥= true, updatable = true)
    public java.lang.String getComments()
    {
        return comments;
    }

    @javax.persistence.Column(name = "TELEPHONE", nullable = false, insertable
    ➥= true, updatable = true)
    public java.lang.String getTelephone()
    {
        return telephone;
    }

@Length(max=250)
```

```
@javax.persistence.Column(name = "ADRESSE", nullable = false, insertable = true,
➥updatable = true)
    public java.lang.String getAdresse()
    {
        return adresse;
    }
```

La classe Client.java porte l'annotation @Name lui permettant d'avoir un nom dans le contexte Seam. Aucun tag @Scope n'étant déclaré, le contexte par défaut est utilisé. Dans ce cas, l'entité `client` est enregistrée comme étant un objet nommé `Client` dans le contexte conversationnel Seam.

Certaines annotations abordées au chapitre 8 sont utilisées ici pour la validation. Elles permettront de déclencher la validation JSF des attributs qu'elles annotent. Le formulaire « Ajout d'un nouveau client » en est un exemple.

La seconde classe à modifier est le bean session `ClientsServiceBean` qui déclare toutes les méthodes CRUD relatives au bean `Client` :

```
@Name("ClientsServiceBean")
@Scope(ScopeType.CONVERSATION)
public class ClientsServiceBean
    extends book.webstock.services.ClientsServiceBase
{
@In(required=true,create=true) @Out
    Client client;

public Client getClient() {
    return client;
}

public void setClient(Client client) {
    this.client = client;
}

    public void ajouterClient(){
    try {
        getClientDao().create(client);
    } catch (Exception e) {
        e.printStackTrace();
    }
    }
    // --------------- Constructors ---------------

    public ClientsServiceBean()
    {
        super();
    }


…
}
```

Le service "ClientsServiceBean" est enregistré comme étant un objet Seam appelé `ClientsServiceBean` vivant dans le scope conversationnel. Remarquez bien l'injection du paramètre « client » dans le bean session.

Maintenant que vous avez enregistré vos deux composants dans le contexte conversationnel, vous pouvez les invoquer directement sans passer par les beans managés habituels de JSF. Pour ce faire, reprenez votre facelet ajoutClient.xhtml pour une refactorisation Seam.

Voici le code du facelet avant refactoring :

```
<html 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:c="http://java.sun.com/jstl/core"
      xmlns:t="http://myfaces.apache.org/tomahawk"
      xmlns:a="http://www.andromda.org/cartridges/jsf"
      xmlns:af="http://xmlns.oracle.com/adf/faces">
<ui:composition>
<h:form id="ajoutDUnNouveauClientFormulaireAjoutClientAjoutClientEffectForm"
➥enctype="multipart/form-data">
    <af:panelForm>
        <af:inputText id="nom" value="#{ajoutDUnNouveauClient
        ➥FormulaireAjoutClientAjoutClientEffectForm.nom}" label
        ➥="#{messages['nom']}:" required="false" readOnly="false">
        </af:inputText>
        <af:inputText id="prenom" value="#{ajoutDUnNouveauClient
        ➥FormulaireAjoutClientAjoutClientEffectForm.prenom}" label
        ➥="#{messages['prenom']}:" required="false" readOnly="false">
        </af:inputText>
        <af:inputText id="adresse" value="#{ajoutDUnNouveauClient
        ➥FormulaireAjoutClientAjoutClientEffectForm.adresse}" label
        ➥="#{messages['adresse']}:" required="false" readOnly="false">
        </af:inputText>
        <af:inputText id="telephone" value="#{ajoutDUnNouveauClient
        ➥FormulaireAjoutClientAjoutClientEffectForm.telephone}" label
        ➥="#{messages['telephone']}:" required="false" readOnly="false">
        </af:inputText>
        <af:inputText id="comments" value="#{ajoutDUnNouveauClient
        ➥FormulaireAjoutClientAjoutClientEffectForm.comments}" label
        ➥="#{messages['comments']}:" required="false" readOnly="false" rows="3"
        ➥columns="40">
        </af:inputText>
        <f:facet name="footer">
            <af:panelButtonBar>
                <af:commandButton text="#{messages['ajout.client.effect']}"
                ➥action="#{ajoutClientController.formulaireAjoutClient
                ➥AjoutClientEffect}"/>
            </af:panelButtonBar>
        </f:facet>
    </af:panelForm>
    <a:validator client="false"/>
</h:form>
</ui:composition>
</html>
```

Le code du facelet après sa refactorisation est le suivant (remarquez au passage que le code est plus compact) :

```
<html 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:c="http://java.sun.com/jstl/core"
      xmlns:t="http://myfaces.apache.org/tomahawk"
      xmlns:a="http://www.andromda.org/cartridges/jsf"
      xmlns:af="http://xmlns.oracle.com/adf/faces">
<ui:composition>
<h:form >
    <af:panelForm>
        <af:inputText id="nom" value="#{Client.nom}" label="#{messages['nom']}:"
        ➥required="false" readOnly="false">
        </af:inputText>
        <af:inputText id="prenom" value="#{Client.prenom}"
        ➥label="#{messages['prenom']}:" required="false" readOnly="false">
        </af:inputText>
        <af:inputText id="adresse" value="#{Client.adresse}"
        ➥label="#{messages['adresse']}:" required="false" readOnly="false">
        </af:inputText>
        <af:inputText id="telephone" value="#{Client.telephone}"
        ➥label="#{messages['telephone']}:" required="false" readOnly="false">
        </af:inputText>
        <af:inputText id="comments" value="#{Client.comments}"
        ➥label="#{messages['comments']}:" required="false" readOnly="false"
        ➥rows="3" columns="40">
        </af:inputText>
        <f:facet name="footer">
            <af:panelButtonBar>
                <af:commandButton text="#{messages['ajout.client.effect']}"
                ➥action="#{ClientsServiceBean.ajouterClient}"/>
            </af:panelButtonBar>
       </f:facet>
    </af:panelForm>
    <a:validator client="false"/>
</h:form>
</ui:composition>
</html>
```

Maintenant que tout est prêt, il ne vous reste qu'à ajouter le fichier components.xml pour JBoss Seam, comme dans le listing suivant :

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

<core:ejb installed="false"/>
<component
        name="client"
        class="unnamed.Client"
        scope="CONVERSATION"/>
<component
        name="ClientsServiceBean"
```

```
        class="book.webstock.services.ClientsServiceBean"
        scope="CONVERSATION"
    jndi-name="testNG-1.0.0/ClientsServiceBean/remote"/>
      <!-- seam-components merge-point -->
</components>
```

Ce fichier doit se trouver sous le répertoire META-INF du projet EJB. Il utilise ici une description minimale pour déclarer les objets Seam (noms, classes et contexte correspondant). Le code complet est disponible sur la page Web dédiée à l'ouvrage.

Tout est prêt pour exécuter l'application webstock avec son « revêtement » Seam. Vous n'avez plus besoin de passer par les beans managés ni par les contrôleurs, évitant ainsi les redondances et laissant Seam gérer en toute transparence les synchronisations entre les couches métier et Web.

La figure 13.7 illustre votre use case après l'intégration Seam.

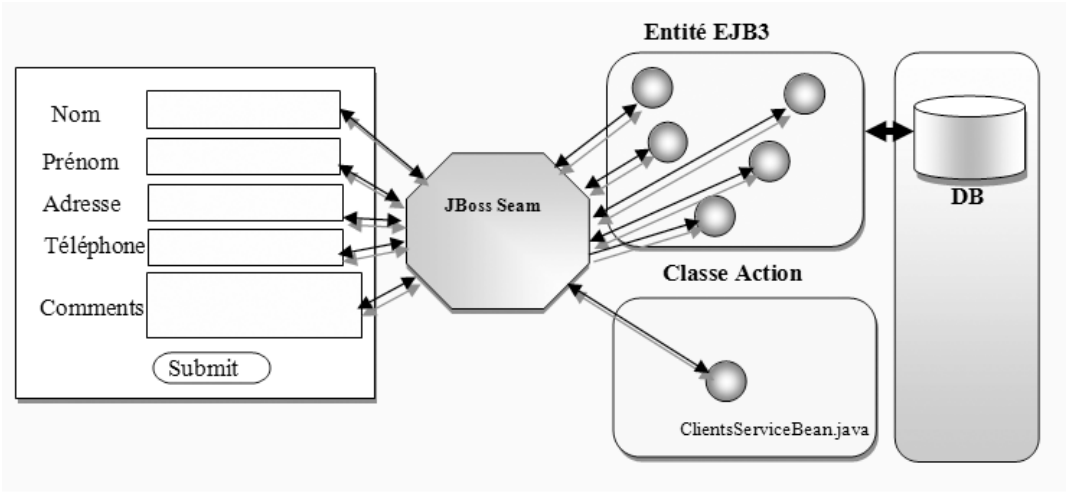


Figure 13.7

Use case « Ajout d'un nouveau client » après l'intégration Seam

Conversation Seam

Vous allez à présent appliquer une conversation Seam à l'application WebStock.

Vous prendrez cette fois le cas classique du Login, ou identification à l'application.

Voici le listing du bean session WebStockAccessServiceBean :

```
@javax.ejb.TransactionAttribute(javax.ejb.TransactionAttributeType.REQUIRED)
@javax.ejb.Remote({book.webstock.services.WebStockAccessServiceRemote.class})

@Scope(ScopeType.CONVERSATION)
@Name("WebStockAcess")

public class WebStockAccessServiceBean
    extends book.webstock.services.WebStockAccessServiceBase
{
```

```
// ------ Session Context Injection ------

@javax.annotation.Resource
protected javax.ejb.SessionContext context;

@In @Out
WebstockAccess user;

@In
FacesMessages facesMessages;


@RequestParameter("login")
String login;

@RequestParameter ("password")
String password;

// ------ Persistence Context Definitions --------

/**
 * Inject persistence context EJB3_JSF_SEAM_Webstock
 */
@javax.persistence.PersistenceContext(unitName = "EJB3_JSF_SEAM_Webstock")
protected javax.persistence.EntityManager emanager;

// ------ DAO Injection Definitions --------

/**
 * Inject DAO WebstockAccessDao
 */
@javax.ejb.EJB
private WebstockAccessDao webstockAccessDao;

// --------------- Constructors ---------------

public WebStockAccessServiceBase()
protected WebstockAccessDao getWebstockAccessDao()
{
    return this.webstockAccessDao;
}

// -------- Business Methods  --------------

/**
 *
 */
@Begin
@javax.ejb.TransactionAttribute(javax.ejb.TransactionAttributeType.REQUIRED)
public boolean login(java.lang.String login, boolean password)
{
    try
    {
        return this.handleLogin(login, password);
    }
    catch (Throwable th)
    {
```

```
            throw new book.webstock.services.WebStockAccessServiceException(
                "Error performing 'book.webstock.services.WebStockAccessService
                ➥.login(java.lang.String login, boolean password)' --> " + th,
                th);
        }
    }

    /**
     * Rechercher le user ayant ce login et ce password pour voir   si ça existe
     */

    Public WebStockAccess getUser(login, password){
        return getWebstockAccessDao.findUser(login, password);
    }

    @End
    public void displayLoginResult(){
    if (user==null)
        facesMessages.add("Error login action");
    else{facesMessages.add("you're logged in");}
    }

    /**
     * Performs the core logic for {@link #login(java.lang.String, boolean)}
     */
    protected abstract boolean handleLogin(java.lang.String login, boolean password)
        throws java.lang.Exception;

    public void logOut(WebstockAccess user){
    try
        {
            handlelogOut(user);
        }
        catch (Throwable th)
        {
            throw new book.webstock.services.WebStockAccessServiceException(
                "Error performing 'book.webstock.services.WebStockAccessService
                ➥.logout(user)' --> " + th,
                th);
        }
    }

    protected abstract boolean handlelogOut(WebstockAccess user)
    throws java.lang.Exception;


    /**
     * Remove lifecycle method
     */
    @Remove @Destroy
    public void destroy() {
        handleDestroy();
    }

    /**
     * Performs the core logic for {@link #destroy())
     */
    protected abstract void handleDestroy()}}
```

Le service WebStockAccessServiceBean est un bean session avec état qui implémente la logique métier de connexion à l'application webstock *via* un nom d'utilisateur et un mot de passe. La requête de login est élue pour être conversationnelle selon les termes Seam.

Le tableau 13.2 récapitule les annotations utilisées jusqu'ici dans la gestion du contexte Seam (voir en annexe pour plus de détails).

Tableau 13.2 Annotations de gestion des contextes Seam

Annotation Seam	Description
@Scope(ScopeType.CONVERSATION)	Permet au bean d'être enregistré dans le contexte conversationnel Seam.
@Begin	Permet de débuter la conversation Seam si la méthode qu'elle marque est invoquée. Dans le cas du Bean WebStockAccessBean, c'est la méthode `login()`.
@End	Permet d'arrêter la conversation si la méthode qu'elle annote est invoquée. Dans ce cas, la méthode `displayLoginResult()` arrête la conversation.
@Destroy	Utilisée pour tout clean-up effectué par Seam
@Remove	Informe la plate-forme que le bean session doit être détruit après l'invocation de la méthode qu'elle marque.

@Destroy et @Remove

Dans Seam, tous les beans session avec état doivent déclarer une méthode marquée @Destroy et @Remove. C'est la méthode EJB Remove() qui est invoquée quand Seam détruit le contexte Session.

En résumé

Tout au long de cette partie consacrée au développement EJB3 avec la plate-forme JBoss, vous avez pu apprécier toute la souplesse et la simplicité de la nouvelle norme EJB3, complétée par la richesse de l'API JPA et des sous-projets associés (Dali en particulier).

Avec l'apport du framework Seam, conjugué à une démarche de développement centrée sur le modèle avec MDA voir SOA, la spécification EJB3 devrait atteindre toute sa puissance pour le développement d'applications Web 2.0 complexes d'aujourd'hui et de demain.