9

Les fondamentaux EJB3

Ce chapitre rappelle la genèse de la nouvelle spécification EJB3 et présente les nombreux avantages de cette norme, voulue comme une simplification radicale de l'ancienne norme EJB2, jugée trop complexe par la communauté des développeurs Java pour les développements courants.

La spécification EJB3

Apparue sous sa version finale en 2006, la spécification EJB3 dispose de nombreux atouts pour redorer un tant soit peu le blason des EJB, terni en grande partie par la complexité de l'ancienne norme EJB2.

Comme nous allons le voir, cette nouvelle mouture a bénéficié des apports de frameworks tels que Spring et Hibernate et des annotations emmenés par le JDK5 pour permettre à la communauté des développeurs J2EE/JEE de s'en emparer et de l'adapter à leurs projets.

D'EJB1 à EJB3

Attendue avec impatience par la communauté Java, EJB3, dont les premières « épreuves » ont paru en 2003, est une étape importante pour la spécification EJB, qui compte déjà plus de huit ans d'évolutions. Répondant aux souhaits des développeurs et aux best practices de développement, cette nouvelle spécification a pour premier avantage décomplexifier fortement la précédente.

Rappelons les dates clés qui ont jalonné l'évolution de la spécification :

- Mars 1998 : EJB 1.0
- Novembre 1999 : EJB 1.1
- Août 2001 : EJB 2.0
- Novembre 2003 : EJB 2.1

- 2005 : EJB 3.0
- 2006 : phase finale de la spécification EJB 3.0

Définie par la JSR-220, la spécification EJB3 comporte trois documents distincts :

- EJB3 simplified API, qui fournit une description globale du nouveau modèle de développement EJB3.
- EJB3 Core Contracts and Requirements, qui se concentre sur les beans de types session et orientés message (MDB).
- Java Persistence API, qui s'applique aux beans entité et aux frameworks de persistance.

Le nouveau modèle de composants EJB3 définit toujours les mêmes types d'objets :

- Les beans session, qui exécutent les services et opérations métier et orchestrent les transactions. Il existe deux types de bean session :
  - Les beans session stateless (sans état), qui ont une durée de vie limité et qui se limitent à l'invocation de la méthode et à la récupération de son résultat (exemple : une classe MailSender chargée de l'envoi de messages).
  - Les beans session stateful (avec état), qui permettent la conservation de l'état transactionnel (exemple : caddy virtuel).
- Les beans MDB, qui sont invoqués de manière asynchrone et répondent aux événements extérieurs par le biais d'une file de messages (exemple : un bean chargé de la commande d'articles à un centre de gestion des commandes).
- Les beans entité, qui disposent d'un unique identifiant et représentent les données persistantes du modèle métier. Ils peuvent être utilisés pour mapper une entrée dans la table de la base avec une classe Java (mapping objet-relationnel), le serveur d'applications fournissant les fonctionnalités pour charger, mettre à jour et supprimer les valeurs de l'instance de classe dans la base.

La différence fondamentale entre EJB2 et EJB3 tient à ce que les beans entité sont désormais gérés par un fournisseur de persistance particulier (TopLink, Hibernate, etc.), et non plus par le conteneur. Ils ne sont donc plus considérés comme de « vrais » beans d'entreprise. Cette volonté de séparer les services de persistance du conteneur est la réponse au reproche fait à EJB2 de ne rendre utilisables les couches implémentées avec des EJB qu'au sein de conteneurs EJB, rendant problématiques les tests unitaires. Concernant enfin la persistance, EJB3 propose un nouveau modèle de persistance, Java Persistence API, totalement inspiré d'Hibernate.

La spécification JEE5, qui intègre EJB3, prend la suite de J2EE 1.4 (notez l'abandon du « 2 » de J2EE, qui faisait référence au JDK 1.2). La version qui succède à J2SE 5.0 se nomme Java SE 6, ou JSE6. L'objectif de JEE5 est avant tout de simplifier le développement des applications Java d'entreprise. Elle intègre bon nombre de fonctionnalités EJB3, telles que les annotations, la programmation de POJO (Plain Old Java Objects), l'injection de dépendances, de nouvelles API, de nouveaux frameworks, etc.

Les points forts de cette « révolution » du modèle EJB sont détaillés à la section suivante, ainsi que dans les différents chapitres suivants.

Principales nouveautés d'EJB3

Nous ne présentons ici que les fonctionnalités marquantes de la spécification EJB3, lesquelles seront mises en œuvre tout au long des chapitres de cette partie.

Modèle simplifié grâce aux annotations

EJB3 apporte une simplification des développements grâce aux annotations, une nouvelle fonctionnalité du langage Java 5.0 en lieu et place des descripteurs de déploiement. Les annotations sont des métadonnées qui permettent à certains outils de générer des constructions additionnelles à la compilation ou à l'exécution ou encore de renforcer un comportement voulu au moment de l'exécution (comme la nature « sans état » d'un composant EJB).

Les annotations simplifient considérablement l'écriture des programmes, comme nous l'avons vu au chapitre précédent avec Seam. Les annotations permettent « d'attacher » des informations additionnelles (couramment appelées attributs) aux classes Java et aux interfaces, ainsi qu'aux méthodes et aux variables qui les composent.

Ces informations additionnelles apportées par les annotations peuvent être assimilées à des interfaces. Elles peuvent être utilisées dans un environnement de développement tel qu'Eclipse ou WebTools, voire Dali, comme nous le verrons au chapitre 11, et par les différents assistants de déploiement du conteneur JEE. Elles rendent ainsi caduques les descripteurs de déploiement de la norme EJB2 tels que ejb-jar.xml.

La figure 9.1 illustre la complémentarité entre POJO et annotations.

Figure 9.1

Transformation d'un objet POJO avec les annotations en EJB3

Descripteur de la persistance des données simplifié

Le fichier persistence.xml fournit un mécanisme puissant et simple pour la persistance des beans entité, appelée unité de persistance.

En voici un exemple simple :

```
<persistence>
<persistence-unit name="WebStockDB">
<provider>org.hibernate.ejb.HibernatePersistence</provider>
<jta-data-source>java:/WebStockDBDS</jta-data-source>
<properties>
<property name="hibernate.hbm2ddl.auto" value="update"/>
</properties>
</persistence-unit>
</persistence>
```

Ce fichier définit le fournisseur de la source de données (ici le fournisseur Hibernate, adapté au conteneur JBoss). Nous décrivons également dans ce fichier le nom de la

source de données définie par JBoss, ou nom JNDI, qui va être utilisée pour l'invoquer à travers le client à l'aide de la propriété jta-data-source.

La balise <properties> comporte des propriétés additionnelles, comme le fait de générer automatiquement le schéma de la base en fonction du bean entité fourni. La valeur "update" de la propriété "hibernate.hbm2ddl.auto" indique que nous souhaitons qu'Hibernate crée la structure de données automatiquement et la mette à jour si nécessaire (voir la documentation en ligne *http://www.hibernate.org/hib_docs/reference/en/html/tutorial.html).*

Puissant mécanisme d'injection des dépendances (DI)

Tout composant d'une application utilise un tant soit peu les services d'un autre composant pour répondre à ses besoins. Le principal objectif du concept d'injection de dépendances, ou DI (Dependency Injection), est de rendre les dépendances entre composants les plus ténus possibles.

Cela signifie que si un composant a besoin des fonctions d'un autre composant ou d'un gestionnaire de ressources, il doit pouvoir le faire à travers une interface et en utilisant un fichier de configuration adapté, au lieu d'écrire du code. Cela favorise évidemment la flexibilité de l'ensemble de l'application.

Ce concept, popularisé par le framework Spring et incarné par le principe d'Hollywood « ne nous appelez pas, nous vous appellerons », permet de décharger le développeur Java de l'instanciation des objets de l'application et de la résolution des dépendances entre eux.

La figure 9.2 illustre le schéma de principe de l'injection de dépendances. La responsabilité du conteneur est d'injecter l'objet en se fondant sur sa déclaration, à l'inverse, par exemple, d'un lookup JNDI classique, qui nécessite que le bean fasse le travail de recherche des ressources et des beans nécessaires.

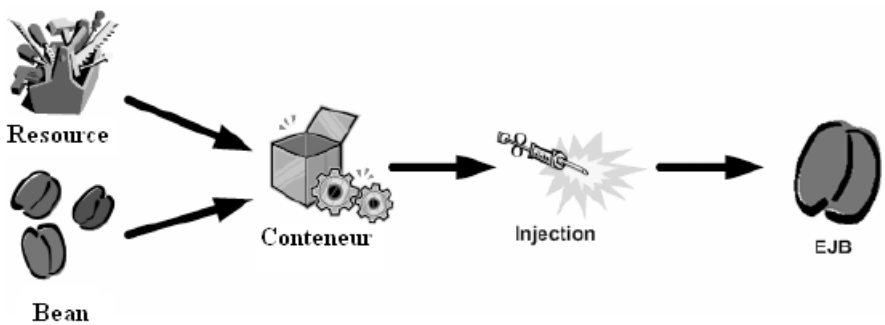


Figure 9.2

Principe du mécanisme d'injection de dépendances

L'exemple suivant illustre l'utilisation de ce principe par le biais de l'annotation javax.annotation.Resource qui s'applique aux ressources de type source de donnée, objets JMS, etc. Le bean délègue au conteneur l'instanciation du contexte monContexte

ainsi que de la datasource maBD, évitant les fastidieux lookups JNDI bien connus des développeurs EJB2 :

```
@Resource(name="maBD", type=javax.sql.DataSource)
@Stateful public class ShoppingCartBean implements ShoppingCart
{
    @Resource SessionContext monContexte;

    public Collection startToShop (String prodNom) {
        ...
        DataSource prodNom = (DataSource)monContexte.lookup("maBD");
      Connection conn = maBD.getConnection():
      ...
  }
  ...
}
```

Nouvelle API de persistance

Désormais, dans JEE5, les entités persistantes sont de simples JavaBeans, les fameux POJO, déclarés à l'aide d'une annotation @Entity. Développée sous les auspices du JSR-220, la nouvelle spécification EJB 3.0 se veut moins académique que l'ancienne et s'inspire des meilleures idées qui ont émergé dans différents contextes : le framework Open Source Hibernate 3.0, les spécifications JDO 2.0 de la JSR-243 ou des solutions commerciales comme TopLink.

En contraste avec les EJB 2.x, la nouvelle API de persistance est utilisable aussi bien dans un contexte dit « managé », c'est-à-dire au sein d'un serveur d'applications, que dans une application J2SE autonome.

Support des intercepteurs

EJB 3.0 intègre une approche POA (programmation orientée aspect) en introduisant la notion d'intercepteur : c'est une méthode qui intercepte l'invocation d'une méthode métier sur un EJB session ou MDB. EJB 3.0 propose même d'associer à un EJB session une liste de classes Interceptors dont les méthodes intercepteront les invocations aux méthodes métier de cet EJB.

Synthèse des frameworks à succès

Toutes ces nouveautés, qui vont vers une plus grande simplicité du modèle EJB, ont déjà été anticipées par des frameworks tels que Spring et Hibernate. C'est en ce sens que nous pouvons affirmer qu'un projet démarré avec la technologie EJB3 bénéficie d'assises et de ressources solides.

À l'heure d'écrire ces lignes, quelques serveurs Open Source et commerciaux supportent déjà la norme JEE5 :

- JBoss 5 (version bêta ; la version stable 4.2 supporte déjà EJB3), qui sert de cadre à cet ouvrage *(www.jboss.com)*.
- Geronimo 2.0.1, sorti en août 2007, de la fondation Apache *(http://geronimo.apache.org)*.

- Glassfish, implémentation de référence de Sun des spécifications JEE5, qui supporte les derniers standards technologiques : JSP 2.1, JSF 1 .2, Servlet 2.5EJB 3.0, JAX-WS 2.0, JAXB 2.0, Web Services Metadata for the Java Platform 1.0, … *(https://glassfish.dev.java.net/)*.
- BEA WebLogic 10, premier serveur commercial supportant JEE5.

Curieusement, IBM ne semble pas pressé de sortir une version concurrente supportant EJB3. Attendons les prochaines annonces pour en savoir plus, notamment avec la version 7 du serveur WebSphere, attendue avec impatience à l'heure ou nous mettons sous presse.

Introduction aux beans session

Quel que soit le niveau de la norme EJB utilisée, les beans session sont des composants qui s'exécutent au sein d'un conteneur EJB standalone ou d'un serveur d'applications.

Ces composants sont utilisés pour représenter des cas d'utilisation ou des traitements spécifiques du client. Ils gèrent les opérations sur les données persistantes, mais non les données persistantes elles-mêmes.

Il existe deux types de beans session : les beans non persistants (stateless) et les beans persistants (stateful).

Quand utiliser des beans session ?

Cette question se pose de manière identique avec les EJB3 et les EJB2. Un bean session est utilisé pour modéliser un processus métier nécessitant le maintien d'un état conversationnel pour le client. Il permet en outre de modéliser les traitements associés, souvent avec l'aide d'un bean entité pour les opérations transactionnelles associées et la persistance.

Exemples de situation

Typiquement, un bean session est utilisé pour :

- Modéliser un traitement dont l'état ne sera pas rendu persistant et pour une durée déterminée.
- Modéliser un traitement de gestion des employés d'une société en leur affectant un service et un profil d'identification dans le système cible en liaison avec un annuaire.
- Créer un bordereau de commande pour un client du système.
- Gérer et orchestrer les méthodes dites CRUD (création, lecture, mise à jour, suppression) sur une entité du modèle métier.

Avantages

Voici quelques-uns des avantages offerts par les beans session :

- Amélioration des services du serveur d'applications en apportant des fonctionnalités évoluées de gestion transactionnelle.
- Aide aux besoins de déploiement du client lorsque que l'application cliente n'est pas localisée dans le même serveur.

- Amélioration de la sécurité fournie par le conteneur au niveau du composant et de la méthode.

Performances

Pour des raisons de performances, il est important de distinguer les accès locaux (composants session EJB ou Web situés dans une même application) et les accès distants (composants session entre plusieurs applications).

Beans session sans état

Les beans session non persistants sont les types d'EJB les plus simples à implémenter. Ils ne conservent aucun état de leur conversation avec les clients entre les invocations de méthodes et sont donc facilement réutilisables dans la partie serveur. Comme ils peuvent être mis en cache, ils supportent bien les variations de la demande.

Lors de l'utilisation de beans session non persistants, tous les états doivent être stockés à l'extérieur de l'EJB.

Création de beans session sans état

Pour créer un bean session sans état en utilisant la spécification EJB3, il suffit d'utiliser l'annotation @Stateless appliquée à la classe, comme dans l'exemple suivant :

```
package com.eyrolles.chap09.exempleSession ;

public interface CalculSalaire {
  public final static double tauxHoraire = 7 ;
  public double getSalaire(int nbreHeures) ;
}

package com.eyrolles.chap09.exempleSession ;
import javax.ejb.Local ;

@Local
public interface CalculSalaireLocal extends CalculSalaire {}

import javax.ejb.Stateless;

@Stateless

public class CalculSalaireBean implements CalculSalaireLocal {

  public CalculSalaireBean() {}

  public double getSalaire(int nbreHeures) {
    return nbreHeures*tauxHoraire ;
  }

}
```

La classe CalculSalaireBean (remarquez l'extension Bean après le nom de la classe, qui est une convention de nommage bien pratique) n'utilise pas d'interface particulière, comme javax.ejb.SessionBean, ni d'interface EJBHome ou EJBObject, si familières de générations de développeurs EJB2.

Nous utilisons ici une interface locale spécifiée par l'annotation @Local, CalculSalaire-Local, et le bean session sans état CalculSalaireBean qui l'implémente.

Rappelons que l'utilisation d'une interface locale précisée avec l'annotation @Local (optionnelle, mais utile) spécifie que les méthodes qui sont définies ici sont disponibles uniquement pour les clients qui ont invoqué cet EJB au sein du même conteneur JEE, cette interface locale étant équivalente à l'interface locale utilisée avec EJB2.

À l'inverse, il existe, comme en EJB2, une interface distante spécifiée par l'annotation @Remote (annotation obligatoire). Cette interface est toutefois à utiliser avec modération pour des raisons de performance, du fait du passage des paramètres par valeur et non par référence *via* RMI/IIOP entre les différentes machines virtuelles.

Voici le code correspondant à la définition d'une interface distante (Remote) qu'une simple annotation @Remote suffit à utiliser :

```
package com.eyrolles.chap09.exempleSession ;

import javax.ejb.Remote ;


@Remote
public interface CalculSalaireRemote extends CalculSalaire {}

package com.eyrolles.chap09exempleSession ;
import javax.ejb.Stateless ;

@Stateless

public class CalculSalaireBean implements CalculSalaireRemote {
  public CalculSalaireBean() {}
  public double getSalaire(int nbreHeures) {
    return nbreHeures*tauxHoraire ;
  }
}
```

Il est possible de rendre le bean CalculeSalaireBean accessible à distance et localement avec les mêmes fonctions et *via* la même interface en modifiant la signature du bean de la manière suivante :

```
@Stateless

public class CalculSalaireBean implements
CalculSalaireRemote,CalculSalaireLocal {
  public CalculSalaireBean() {}
  public double getSalaire(int nbreHeures) {
    return nbreHeures*tauxHoraire ;
  }
}
```

Si le bean n'implémente aucune interface (déconseillé), une interface métier locale est générée automatiquement par le conteneur. Elle prend le nom de la classe moins le suffixe Bean ou Impl en fonction de la convention de nommage utilisée.

L'exception levée par le conteneur RemoteException n'est plus nécessaire ici, l'accès distant étant complètement encapsulé. C'est encore un des aspects de la simplification de la norme EJB3.

Interfaces locales et distantes

Une application cliente qui accède aux beans session hébergés dans un conteneur d'applications peut être de trois types :

- Distante : les clients de l'application s'exécutent dans une JVM séparée du bean session auquel ils accèdent (voir figure 9.3). Un client distant accède au bean session à travers une interface métier distante. Un client distant peut être un autre bean, un programme Java client ou simplement une servlet. Le principal avantage de l'utilisation d'une interface distante est l'indépendance de la localisation des traitements. Cela se paye en contrepartie de moins bonnes performances en termes d'invocation de méthodes distantes et de la nécessité de sérialiser les objets et de les passer par valeur.
- Locale : les clients locaux s'exécutent dans la même JVM (voir figure 9.4). Un client local peut être un autre bean ou une application Web utilisant des servlets et des JSP/JSF. Ce type d'interface est dite *location dependant*.
- Web Service : vous pouvez publier un bean session sous forme de service Web. Ce dernier peut être invoqué à partir d'un client de service Web.

Figure 9.3

Client riche utilisant les interfaces distantes d'un bean session

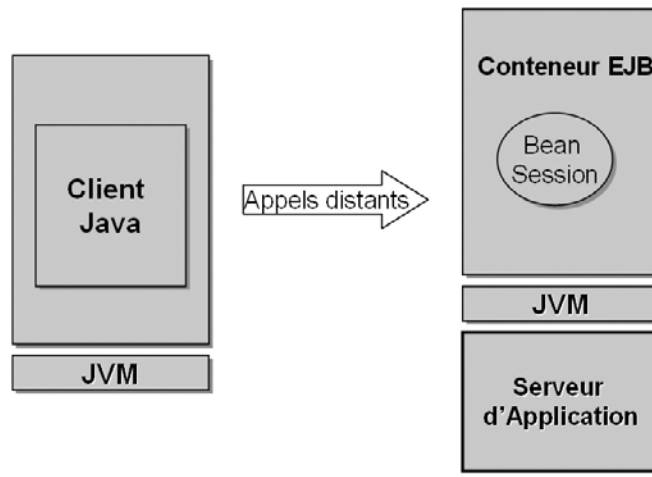
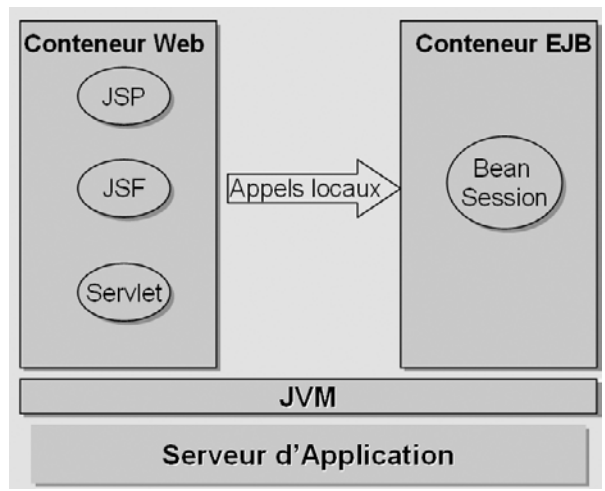


Figure 9.4

Client Web accédant aux méthodes d'un bean session via *une interface locale*



Le choix d'un type d'interface est déterminé par les contraintes de l'application. En règle générale, une application de type client Web qui s'exécute au sein de la même JVM et qui invoque les services d'un bean utilise une interface locale.

À l'inverse, un client Java riche, de type Eclipse RCP par exemple, utilise une interface distante pour invoquer ses méthodes distantes.

Selon les cas, il est conseillé d'utiliser des interfaces métier à la fois locale et distante afin que celles-ci puissent supporter les différents types de clients d'application.

Un client peut obtenir une interface métier du bean session en utilisant l'injection de dépendances ou par un lookup JNDI classique. En utilisant l'injection de dépendances avec l'annotation @EJB, vous pouvez, par exemple, obtenir une interface métier sur le bean session SearchFacade en utilisant le code suivant :

```
public class SearchFacadeClientIOC {

@EJB
SearchFacade searchFacade;

public static void main(String[] args) {

    SearchFacadeClientIOC SearchFacadeClientIOC
        ➥= new SearchFacadeClientIOC(args);
    SearchFacadeClientIOC.doTest();

}

public SearchFacadeClientIOC(String[] args) {

}

  public void doTest() {
    InitialContext ic;

    try {

      ic = new InitialContext();
      List articlesList = searchFacade.articleSearch("Disque Dur");

      for (String article: (List<String>)articlesList) {
        System.out.println(article);
      }
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }


}
```

Cycle de vie des beans et méthodes de callback

Le serveur d'applications contrôle entièrement le cycle de vie des composants EJB déployés en gardant au besoin un pool de beans sans état disponible, l'application n'ayant aucun contrôle sur l'initialisation du bean.

La méthode `create()`, bien connue des développeurs EJB2 — qui est, rappelons-le, une méthode de fabrique utilisée par les clients pour obtenir une référence à un objet EJB —, n'existe plus en EJB3. Elle est remplacée par les annotations, comme le montre l'extrait suivant :

```
import javax.ejb.Stateless;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

@Stateless
public class  CalculSalaireBean implements CalculSalaireLocal {

@PostConstruct
public void init() {
// invoque toutes methodes ou composants necessaires lors de la creation du bean
}

@PreDestroy
public void recycle() {
// invoque toutes methodes ou composants necessaires lors de la destruction du bean
}

}
```

La nouveauté dans ce mécanisme est son caractère optionnel. En EJB2, le développeur était forcé d'écrire ces méthodes, même si elles n'étaient pas implémentées par la suite. La spécification EJB3 se sert des annotations pour définir des méthodes de callback.

Une méthode de callback peut être n'importe quelle méthode définie dans un bean session qui possède une annotation de callback. Le conteneur EJB appelle cette méthode à des moments déterminés du cycle de vie du bean. Par exemple, la méthode de callback `init()` associée à l'annotation @PostConstruct intervient après toutes les injections de dépendances effectuées par le conteneur et avant le premier appel de la méthode métier. La méthode de callback associée à l'annotation @PreDestroy est appelée au moment où l'instance du bean est détruite.

Intercepteurs

La spécification EJB3 fournit des annotations particulières, appelées intercepteurs (à opposer aux méthodes de callback, même si le principe est assez proche), qui permettent d'intercepter l'invocation de méthodes métier. Une méthode intercepteur peut être définie pour un bean session ou un bean orienté message (MDB). Son utilisation peut se justifier dans des situations très particulières, dans lesquelles il est nécessaire d'effectuer certains contrôles, comme par exemple des contrôles de sécurité préliminaires avant une opération de transfert sur un compte.

La mise en œuvre d'un intercepteur s'effectue à l'aide de l'annotation @AroundInvoke appliquée à une méthode particulière. Vous pouvez également définir une classe intercepteur dont les méthodes sont appelées avant qu'une méthode métier soit invoquée sur la classe du bean.

Une classe intercepteur est définie par l'annotation @Interceptor sur la classe du bean à laquelle elle est associée. Cette annotation s'utilise lorsque plusieurs classes intercepteur sont définies.

Les méthodes intercepteur associées à l'annotation @AroundInvoke doivent posséder la signature suivante :

```
public Object <NOM_METHODE>( javax.ejb.InvocationContext ctx) throws
➥java.lang.Exception
```

Ajoutons que vous pouvez définir une méthode intercepteur au sein du bean lui-même, ou dans une classe séparée, sachant qu'il ne peut exister qu'une méthode intercepteur par classe. L'exemple qui suit montre une méthode intercepteur définie dans la classe du bean EmailSystemBean et au niveau de la classe grâce à l'annotation @Interceptors :

```
@Stateless
@Interceptors ({TracingIntercepteur.class})
@Remote(EmailSystem.class)
public class EmailSystemBean implements EmailSystem
{

 @Interceptors({AccountsConfirmIntercepteur.class})
 public void sendBookingConfirmationMessage(long orderId)
   {
      System.out.println("<Dans EmailSystemBean.sendBookingConfirmationMessage
      ➥methode metier");
      String email = "xyz@blabla.com";
      sendMessage(email, "Enregistrement Confirme", "Votre commande " + orderId
      ➥+ "est confirmee !");
      System.out.println("Sortie methode metier EmailSystemBean
      ➥.sendBookingConfirmationMessage >");
   }

@AroundInvoke
   public Object monBeanIntercepteur(InvocationContext ctx) throws Exception
   {
      if (ctx.getMethod().getName().equals("emailLostPassword"))
      {
         System.out.println("*** EmailSystemBean.monBeanIntercepteur - Nom :
         ➥" + ctx.getParameters()[0]);
      }

      return ctx.proceed();
   }

// . . .

}


public class AccountsConfirmInterceptor extends AccountsInterceptor
{
   @Resource(mappedName="java:ConnectionFactory")
   QueueConnectionFactory cf;

   @Resource(mappedName="queue/tutorial/accounts")
   Queue queue;

   @PersistenceContext
   EntityManager em;

   QueueConnection conn;
```

```
    public Object intercept(InvocationContext ctx) throws Exception
    {
       return null;
    }

// . . .

}
```

Support des transactions

Il peut exister dans une application un scénario dans lequel un bean session avec état cache ses modifications avant de les stocker dans la base de données. Par exemple, le bean ShoppingCartBean ne peut décider de sauvegarder dans la base les articles accumulés dans le cache de données qu'après notification du conteneur EJB, en particulier lors d'une notification afterCompletion.

Cela s'effectue par le biais de l'interface javax.ejb.SessionSynchronization, qui permet de mettre en place les trois types de notifications suivants par le conteneur :

- afterBegin : indique qu'une nouvelle transaction à été initialisée.
- beforeCompletion : indique que la transaction est sur le point d'être commitée en base.
- afterCompletion : indique que la transaction a été commitée.

L'interface SessionSynchronization est définie comme suit :

```
package javax.ejb;

public interface javax.ejb.SessionSynchronization {
    public abstract void afterBegin( ) throws RemoteException;
    public abstract void beforeCompletion( ) throws RemoteException;
    public abstract void afterCompletion(boolean committed)
        throws RemoteException;
}
```

La méthode `afterCompletion()` est toujours invoquée, que la transaction se soit bien déroulée lors d'un commit ou non lors d'un rollback. Si la transaction a bien été achevée, ce qui signifie que `beforeCompletion()` a été invoquée, le paramètre committed de la méthode `afterCompletion()` est égal à true ; si elle a échoué, il est égal à false.

Beans session avec état

Les beans session persistants, ou avec état, conservent par définition leur état entre l'invocation de leurs méthodes. Ils ont une association 1-1 avec un client et sont capables de conserver leurs états eux-mêmes.

Le conteneur d'EJB a en charge le partage et la mise en cache des beans session persistants, ceux-ci passant par la passivation et l'activation.

Création de beans session avec état

Pour créer un bean session avec état, il suffit de l'annoter avec @Stateful.

L'extrait de code suivant crée le bean avec état ShoppingCartBean, un exemple classique de caddy virtuel (à noter la propriété name pour le désigner) :

```
package com.webstock.chap09.stateful;
import javax.ejb.Remove;
```

```
import javax.ejb.Stateful;
import com.webstock.chap09.service.ShoppingCart;

@Stateful (name="ShoppingCart")
public class ShoppingCartBean implements ShoppingCart, ShoppingCartLocal {
// ...

@PostConstruct
  public void initialize() {
    cartItems = new ArrayList();
}

@PreDestroy
  public void exit() {
    System.out.println("Saved items list into database");
  }

@Remove
public void purchase() {
// persiste l'article dans la base
}
}
```

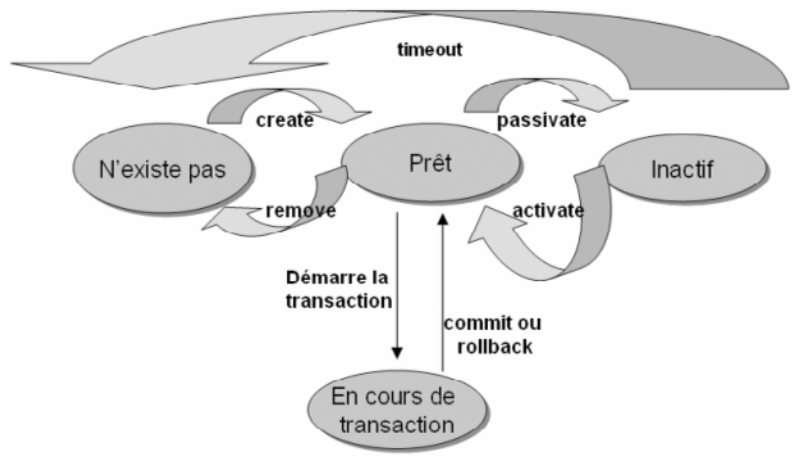
Ce code ne présente pas de difficulté particulière. L'annotation @Remove associée à la méthode de callback `purchase()` a été ajoutée afin de spécifier au conteneur quand libérer le bean pour sa réutilisation éventuelle. Lorsque cette méthode est exécutée, le bean envoie un signal au conteneur, lequel pourra le recycler ultérieurement. Il est possible d'avoir plusieurs méthodes associées à Remove, mais vous devez au moins en avoir une précédée de l'annotation @Remove.

Les beans session stateful supportent les événements de callback pour la construction, la destruction, l'activation et la passivation.

La figure 9.5 illustre les états associés à un bean session avec état.

Figure 9.5

Transition des états d'un bean stateful



Événements de callback des beans session

Le tableau 9.1 récapitule les événements de callback et les annotations correspondantes des beans de type session avec ou sans état.

Tableau 9.1 Événements de callback des beans session

Événement de callback	Type de bean	Annotation	Description
Init	Stateful	@Init	Désigne les méthodes d'initialisation pour le bean session avec état. La méthode @PostConstruct est appelée après la méthode @Init.
PreConstruct	Stateless et stateful	@PreConstruct	Invoque la méthode de callback associée juste après qu'une nouvelle instance d'un bean a été créée par le conteneur.

PostConstruct	Stateless et stateful	@PostConstruct	La méthode de callback associée s'exécute lors de l'instanciation par le conteneur du bean. Exemple :
			`@PostConstruct`
			`public void initialize()`
			`{`
			`  cartItems = new ArrayList();`
			`}`
			Si le bean utilise un mécanisme d'injection pour acquérir des ressources, l'événement PostConstruct se déclenche après que l'injection s'est effectuée et avant que la première méthode du bean soit appelée.

PreDestroy	Stateless et stateful	@PreDestroy	Annote une méthode comme méthode de callback invoquée juste avant que l'instance du bean soit détruite. S'utilise pour libérer certaines ressources. Exemple :
			`@Stateful public class ShoppingCartBean`
			`implements ShoppingCart {`
			`private float total;`
			`private Vector productCodes;`
			`public int someShoppingMethod(){...};`
			`...`
			`@PreDestroy endShoppingCart() {...};`
			`}`

PreActivate	Stateful	@PreActivate	Signale que l'instance vient juste d'être réactivée par le conteneur. Son objectif est de permettre à des beans session avec état de maintenir les ressources nécessitant d'être réallouées durant l'activation de l'instance. Exemple :
			`@Stateful`
			`public class MystatefulBean {`
			`...`
			`@PreActivate`
			`  public void passivate() {`
			`  // open socket connections, ...`
			`  }`
			`}`

Tableau 9.1 Événements de callback des beans session *(suite)*

Événement de callback	Type de bean	Annotation	Description

PrePassivate	Stateful	@PrePassivate	Invoquée juste avant que le bean passe à l'état inactif et stocke son état dans le cache. Exemple :
			`@Stateful`
			`public class MystatefulBean {`
			`...`
			`@PrePassivate`
			`public void passivate() {`
			`// close socket connections, ...`
			`}`
			`}`

Remove	Stateful	@Remove	Lorsque la méthode associée à l'annotation @Remove est appelée, le conteneur supprime l'instance du bean du pool après que la méthode s'est exécutée. Exemple :
			`@Remove`
			`public void stopSession()`
			`{`
			`//Corps de la methode pouvant etre vide`
			`System.out.println("Appel de la methode stopSession avec l'annotation @Remove…");`
			`}`

AroundInvoke	Stateless et stateful	@AroundInvoke	Mise en œuvre des méthodes intercepteur.

Beans session EJB2 vs EJB3

Le tableau 9.2 récapitule les principales différences entre EJB2 et EJB3.

Tableau 9.2 Principales différences entre EJB2 et EJB3

Bean session EJB2	Bean session EJB3
Interface distante (Remote)	Interface métier (Business)
Étend l'interface EJBObject et EJBLocalObject.	Simple POJO
Interface Home (interface d'accueil)	Pas d'interface Home
Étend l'interface EJBHome et EJBLocalHome	Non opérant
Doit disposer d'au moins une méthode create.	Non opérant
Les méthodes create doivent générer l'exception CreateException et RemoteException.	Non opérant
Classe d'implémentation du bean	
Implémente l'interface SessionBean.	Implémente l'interface metier.
Doit définir l'ensemble des méthodes de l'interface SessionBean.	Simple POJO
Définit les méthodes métier pour l'interface distante.	Définit les méthodes métier pour l'interface métier.
Type de bean défini dans le fichier descripteur	Type de bean défini par les annotations @Stateless ou @Stateful
Les callbacks sont supportées tout au long du cycle de vie de l'interface SessionBean.	Les callbacks sont supportées à travers les annotations @PreDestroy, @PrePassivate, @PostActivate et @Init.

Tableau 9.2 Principales différences entre EJB2 et EJB3 *(suite)*

Bean session EJB2	Bean session EJB3
Descripteur de déploiement	
Fichier descripteur ejb-jar.xml requis	Les descripteurs de déploiempent sont optionnels. Utilisation du fichier de persistance persistence.xml requis. Fichier orm.xml optionnel.
Bean session client	
Les clients effectuent un lookup pour l'interface d'accueil en utilisant le contexte JNDI.	Utilisation de l'injection de dépendances du conteneur avec l'annotation @EJB
Pour localiser une référence à un bean, la propriété ejb-ref est utilisée.	Non nécessaire. Utilisation de l'annotation @Resource pour l'accès aux ressources

Les beans message (Message Driven Beans)

Les beans orientés message sont utilisés pour gérer des types de traitement asynchrones. Leur objectif est d'écouter des messages sur une file de messages (file d'attente, ou Topic) *via* une interface de programmation API comme JMS (Java Messaging Service).

Rappelons que les beans MDB ne font pas partie de la spécification EJB et qu'ils ont été ajoutés par la suite pour résoudre les problématiques de traitement asynchrone. Avec la spécification EJB3, la classe de bean MDB est annotée avec @MessageDriven, qui spécifie quelle file de message le MDB va surveiller (dans notre exemple queue/mdb). Si la file n'est pas définie, le conteneur JBoss la crée automatiquement au déploiement, sans besoin d'un fichier de configuration XML particulier. La classe du bean doit toutefois implémenter l'interface MessageListener, qui définit une seule méthode `onMessage()` bien connue des développeurs EJB2.

Lorsqu'un message arrive dans la file surveillée par le bean MDB, le conteneur JBoss appel la méthode `onMessage()` de la classe du bean et passe le message entrant comme paramètre de sortie. Dans l'extrait de code suivant, un exemple classique de traitement de message, la méthode `CalculatorBean.onMessage()` récupère le corps du message, le traite en récupérant les paramètres de calcul, effectue l'opération de calcul (méthode calculate) et sauvegarde le résultat dans une classe statique gestionnaire de données RecordManager afin que la page JSP puisse ensuite la traiter.

Le Timestamp `sent` dans le gestionnaire de message se sert de l'identifiant unique pour l'enregistrement de l'opération. La page JSP d'affichage correspondante récupère l'enregistrement de l'opération en se fondant sur l'identification unique du message.

```
@MessageDriven(activationConfig =
{
     @ActivationConfigProperty(propertyName="destinationType",
        propertyValue="javax.jms.Queue"),
     @ActivationConfigProperty(propertyName="destination",
        propertyValue="queue/mdb")
})
public class CalculatorBean implements MessageListener {

  public void onMessage (Message msg) {
    try {
      TextMessage tmsg = (TextMessage) msg;
```

```
        Timestamp sent =
            new Timestamp(tmsg.getLongProperty("sent"));
        StringTokenizer st =
            new StringTokenizer(tmsg.getText(), ",");

        int start = Integer.parseInt(st.nextToken());
        int end = Integer.parseInt(st.nextToken());
        double growthrate = Double.parseDouble(st.nextToken());
        double saving = Double.parseDouble(st.nextToken());
        double result =
            calculate (start, end, growthrate, saving);
            RecordManager.addRecord (sent, result);

      } catch (Exception e) {
        e.printStackTrace ();
      }
    }

    // ... ...
}
```

Voici le détail des nouvelles annotations :

@MessageDriven : déclare un bean message à partir de sa classe.

@ActivationConfigProperty :

- propertyName : nom de la propriété.
- propertyValue : valeur de la propriété à affecter.
- EJB : injection automatique liée à un EJB (instanciation automatique).

Dans l'extrait de code ci-dessus, vous définissez la configuration du bean MDB *via* l'annotation @ActivationConfigProperty (définition du type et du nom de la file d'attente de destination).

Pour être complet, voici le code du client. La JSP calculator.jsp utilise l'API standard JMS pour obtenir la file de message cible du bean MDB en utilisant le nom de la file (queue/mdb) et envoyer ensuite le message à la file :

```
<%@ page import="com.test.mdb.*, javax.naming.*, java.text.*, javax.jms.*,
➥java.sql.Timestamp"%>

<%

  try {
     InitialContext ctx = new InitialContext();
     queue = (Queue) ctx.lookup("queue/mdb");
     QueueConnectionFactory factory =
        (QueueConnectionFactory) ctx.lookup("ConnectionFactory");
     cnn = factory.createQueueConnection();
     sess = cnn.createQueueSession(false,QueueSession.AUTO_ACKNOWLEDGE);

   } catch (Exception e) {
    e.printStackTrace ();
  }
```

```
    TextMessage msg = sess.createTextMessage(
      request.getParameter ("start") + "," +
      request.getParameter ("end") + "," +
      request.getParameter ("growthrate") + "," +
      request.getParameter ("saving")
    );

    sender = sess.createSender(queue);
    sender.send(msg);

  %>
```

Mise en œuvre du développement EJB3 avec Web Tools et JBoss

Afin de mettre en pratique avec JBoss et l'outillage Web Tools les concepts et beans session abordés précédemment, vous allez créer un bean session façade générique destiné à offrir un certain nombre de services de recherche d'articles à l'utilisateur.

Ce bean permettra d'accéder aux enregistrements de la couche back-end, enregistrements qui, pour des raisons didactiques, sont codés en dur à l'intérieur du bean. Cette couche back-end pourra ensuite être transformée en beans entité, que nous détaillons au chapitre suivant.

Prérequis et configuration

Nous supposons déjà configurés une version d'Eclipse 3.3 avec le serveur JBoss 4.2 et le JDK5.

Pour l'installation de l'outillage Eclipse et des sous-projets associés Web Tools et Dali, nous vous conseillons d'utiliser une version bundlée, directement installable et préconfigurée pour un support JEE. Plusieurs éditeurs proposent ce type de version, notamment Yoxos *(http://www.yoxos.com)*.

Vous allez commencer par déclarer le serveur JBoss à Web Tools :

1. Ouvrez la page Preferences d'Eclipse, puis choisissez Server et Installed Server Runtime Environment.
2. Cliquez sur Add.
3. Dans la liste proposée, sélectionnez JBoss 4.2, puis cliquez sur Next.
4. Spécifiez les valeurs correspondant à votre environnement (JRE compatible JDK5 et répertoire d'installation du serveur JBoss 4.2, par exemple C:/jboss-4.2.1.GA).
5. Cliquez sur Finish puis sur OK pour terminer la configuration du serveur.
6. Pour configurer le serveur JBoss ainsi défini, ouvrez la vue Servers.
7. Faites un clic droit dans la vue Servers, et choisissez New Server dans le menu contextuel.
8. La boîte de dialogue de configuration du serveur JBoss s'ouvre alors. Laissez les champs Server's Host Name et server runtime à leur valeur par défaut (respectivement localhost et JBoss 4.2). Cliquez sur Next.

9. L'écran de l'assistant de configuration du serveur JBoss vous propose l'adresse IP du serveur et les ports d'écoute http, JNDI et de la configuration du serveur (respectivement 127.0.0.1, 8080,1099 et default). Cliquez sur Finish, terminant ainsi la configuration du serveur JBoss.

Figure 9.6

Configuration du serveur JBoss

10. Démarrez votre serveur en le sélectionnant par clic droit et en choisissant Start dans le menu contextuel.

Création et configuration du projet

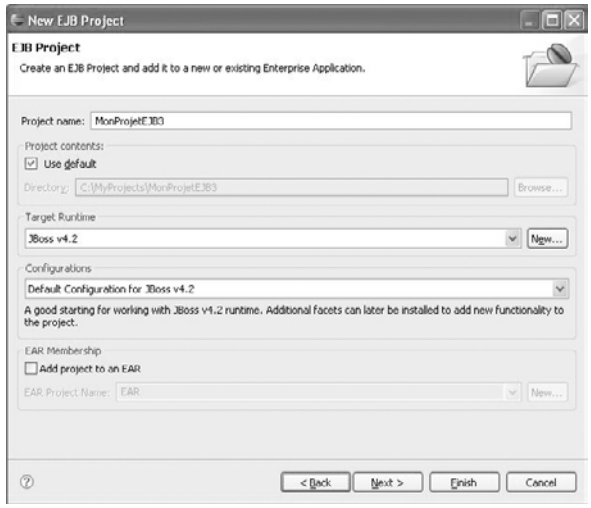
Curieusement, l'outillage et les assistants Web Tools 2.0 ciblent encore les projets J2EE 1.4, comme vous pouvez le constater en créant un projet d'entreprise ou un projet EJB. C'est pour cette raison que vous devez ajouter à la main les bibliothèques de support EJB3, comme vous allez le voir.

Création du projet JEE

1. Ouvrez l'assistant de création de projet Eclipse *via* File, New et Other.
2. Sélectionnez le type de projet EJB, puis cliquez sur Next.
3. Saisissez le nom de votre projet, par exemple MonProjetEJB, et laissez les autres champs inchangés (voir figure 9.7).

Figure 9.7

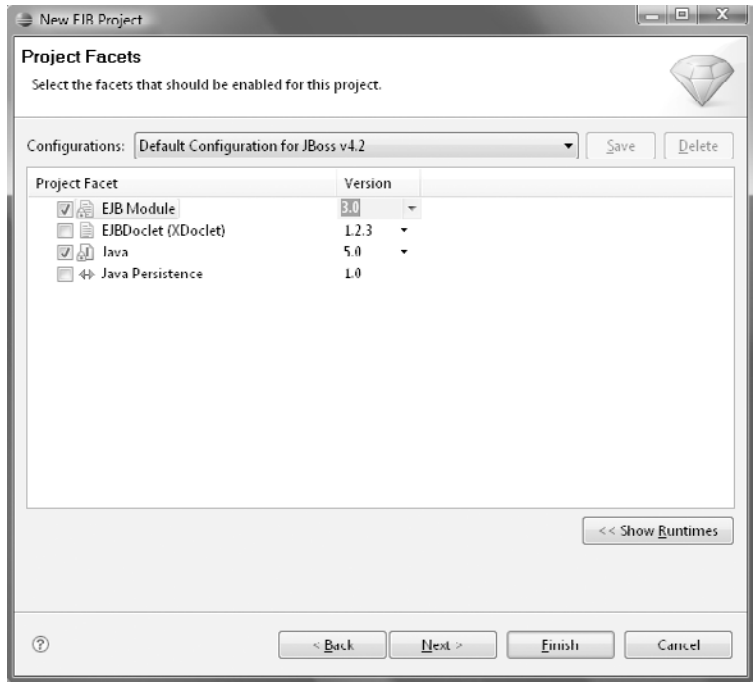
Assistant de configuration de projet EJB de Web Tools (1/2)



4. Cliquez sur Next. La boîte de dialogue illustrée à la figure 9.8 affiche les facets associées. Remarquez le support d'EJB 3.0, Java 5.0 et JPA, contrairement à la version JBoss 4.0, qui ne supporte pas nativement le conteneur EJB3. Le support de XDoclet est toujours proposé pour des raisons de compatibilité avec J2EE 1.4, mais n'est plus conseillé pour un développement JEE5.

Figure 9.8

Assistant de configuration de projet EJB de Web Tools (2/2)



5. Cliquez sur Finish. La structure de votre projet J2EE est créée.

Contrairement aux versions antérieures de JBoss, le chemin de compilation du projet fait désormais référence aux bibliothèques de support EJB3, comme ejb3-persistence.jar, jboss-ejb3.jar, jboss-ejb3x.jar.

Développement d'un bean session sans état

Le développement d'un bean session EJB3 sans état comporte les étapes clés suivantes :

1. Définition de l'interface du composant. Créez une interface sous le package com.eyrolles.chapitre09 (sous le module ejbModule déjà généré) et contenant le code suivant :

```
package com.eyrolles.chapter09;

import java.util.List;
import javax.ejb.Remote;

@Remote
public interface SearchFacade
{
  List articlesSearch(String articleType);
}
```

2. Implémentez le composant en créant la classe SearchFacadeBean. Les méthodes implémentées dans le bean doivent correspondre aux méthodes métier déclarées dans l'interface distante ou locale. Les autres méthodes qui ne possèdent pas d'équivalent au niveau de l'interface métier restent privées aux méthodes du bean.

Voici le code du bean SearchFacadeBean contenant une méthode `articleSearch()` qui a été déclarée dans l'interface métier locale et distante, celle-ci retournant une liste statique d'articles basés sur la catégorie de l'article :

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.Stateless;
import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

@Stateless(name = "SearchFacade")
public class SearchFacadeBean
  implements SearchFacade, SearchFacadeLocal
{
  public SearchFacadeBean() {
  }

  HashMap stockMap = new HashMap();

  public List articleSearch(String articleCategorie) {
    List articleList = new ArrayList();
    if (articleCategorie.equals("Disque Dur")) {
      articleList.add("Hitachi Deskstar 7K1000");
      wineList.add("Maxtor 200 Go Serial ATA II");
      wineList.add("Samsung SpinPoint P - SP2014N - 200Mo");
    }
    else if (articleType.equals("Cartes graphique")) {
      wineList.add("ATI Radeon 7000 - 64 Mo - AGP (ATI Radeon 7000)");
    }

    return articleList;
  }

@PostConstruct
  public void initializeStockList() {
    //countryMap is HashMap

    System.out.println("Invocation de la methode de callback initializeStockList()
    ➥sur evenement @PostConstruct");


    stockMap.put("Disque Dur", "210");
    stockMap.put("Ecrans", "510");
    stockMap.put("Claviers", "670");
    stockMap.put("Cartes graphique", "289");
  }
```

```
@PreDestroy
  public void destroyArticleList() {
    System.out.println("Invocation de la methode de callback destroyWineList()
    ➥sur evenement @PredDestroy");
    stockMap.clear();
  }

  public void afterBegin() {}

  public void beforeCompletion() {}

  public void afterCompletion (boolean b){
        // if (b==false)
          //... traitement echec transaction;
  }

// ...
}
```

Nous avons ajouté une méthode de callback PostConstruct `initializeStockList()` afin d'initialiser la liste des articles en stock lors de l'instanciation du bean. Idéalement, cette liste sera initialisée à partir d'un back-office, alimentée au besoin *via* un bean MDB connecté à une file de messages (typiquement MQ Series). Ici, nous avons simplement codé en dur pour des raisons de simplification ces valeurs dans une structure de type HashMap.

La méthode de callback PreDestroy `destroyArticleList()` survient avant que le conteneur détruise une instance du bean SearchFacade inutilisée ou expirée du pool d'objets. Ici, nous effaçons la HashMap contenant la liste des articles et des quantités associées.

Pour illustrer le mécanisme des intercepteurs, il suffit d'ajouter l'annotation @Around-Invoke au code d'implémentation du bean session SearchFacadeBean :

```
@AroundInvoke
public Object LogMethods (InvocationContext ctx) throws Exception
  {
    String beanClassName = ctx.getClass().getName();
    String businessMethodeNom = ctx.getMethod().getName();
    String target = beanClassName + "." + businessMethodName ;
    long startTime = System.currentTimeMillis();
    System.out.println ("Appel Méthode : " + target);
    try {
      return ctx.proceed();
    }
    finally {
    System.out.println("Sortie méthode " + target);
     long totalTime = System.currentTimeMillis() - startTime;
     System.out.println("Methode metier: " + businessMethodeNom +
     " dans  " + beanClassName + "prends " + totalTime + "ms pour
     s'exécuter...");
  }
}
```

La méthode d'interception LogMethods permet de tracer le temps mis par chaque méthode métier lorsque celle-ci est invoquée par le client Java. Voici sa sortie dans le fichier de log JBoss server.log :

```
2007-10-19 17:13:11,875 INFO  [STDOUT] Invocation de la méthode de callback initialize
➥StockList()  sur evenement @PostConstruct

2007-10-19 17:13:11,875 INFO  [STDOUT] Appel méthode : org.jboss.ejb3.interceptor.Invo-
➥cationContextImpl.articleSearch

2007-10-19 17:13:11,875 INFO  [STDOUT] Sortie méthode org.jboss.ejb3.interceptor.Invo-
➥cationContextImpl.articleSearch

2007-10-19 17:13:11,875 INFO  [STDOUT] Méthode métier  articleSearch dans
➥org.jboss.ejb3.interceptor.InvocationContextImpl prend  0ms pour s'exécuter...
```

Développement d'un bean session avec état

De manière similaire, vous allez développer la classe d'un bean session avec état et son interface métier destinée à conserver les articles ajoutés par un utilisateur dans son caddy virtuel.

Voici le code du bean ShoppingCartBean intégrant les méthodes métier définies (voir plus loin) dans les interfaces locales et distantes :

```
package com.eyrolles.chapitre09;

import java.util.ArrayList;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.Remove;
import javax.ejb.Stateful;

@Stateful(name = "ShoppingCart")
public class

ShoppingCartBean
  implements ShoppingCartRemote, ShoppingCartLocal
{
  public ArrayList cartArticles;

  public ShoppingCartBean() {
  }

  public void addArticle(String article) {
    cartArticles.add(article);
  }

  public void removeArticle(String article) {
    cartArticles.remove(article);
  }

  public void setCartArticles(ArrayList cartArticles) {
    this. cartArticles = cartArticles;
  }
```

```
  public ArrayList getCartArticles() {
    return cartArticles;
  }

  @PostConstruct
  public void initialize() {
    cartArticles = new ArrayList();
  }

  @PreDestroy
  public void exit() {
    System.out.println("Sauvegarde de la liste des articles dans la base….");
  }

  @Remove
  public void stopSession() {
    System.out.println("Méthode stopSession dans l'annotation @Remove");
  }
}
```

Le code précédent peut aussi s'écrire de manière équivalente en utilisant les annotation @Remote et @Local :

```
import javax.ejb.Remote;
import javax.ejb.Local;
import javax.ejb.Stateful;


@Local({ShoppingCartLocal.class})
@Remote({ShoppingCart.class})

@Stateful(name = "ShoppingCart")
public class

ShoppingCartBean
  //implements ShoppingCart, ShoppingCartLocal
{


// ….

}
```

Les interfaces locale et distante sont données ci-dessous à titre d'illustration, étant entendu que l'interface locale sera utilisée pour une utilisation au sein du conteneur Web :

```
package com.eyrolles.chapitre09;

import javax.ejb.Local;
@Local
public interface ShoppingCartLocal {

  void addArticle(String wine);
  void removeArticle(String item);
  ArrayList getCartArticles();
}
```

```
package com.eyrolles.chapitre09;

import javax.ejb.Remote;
@Remote
public interface ShoppingCartRemote {

  void addArticle(String wine);
  void removeArticle(String item);
  ArrayList getCartArticles();
}
```

Le bean ShoppingCartBean supporte des méthodes de callback pour la construction, avec l'annotation @PostConstruct, d'une liste d'articles avec la méthode `initialize()`. La méthode de callback `exit()` (laissée ici à titre d'exemple, mais sans implémentation réelle) se charge de la sauvegarder après que toutes les méthodes associées à l'annotation @Remove ont été exécutées.

Test de l'EJB session

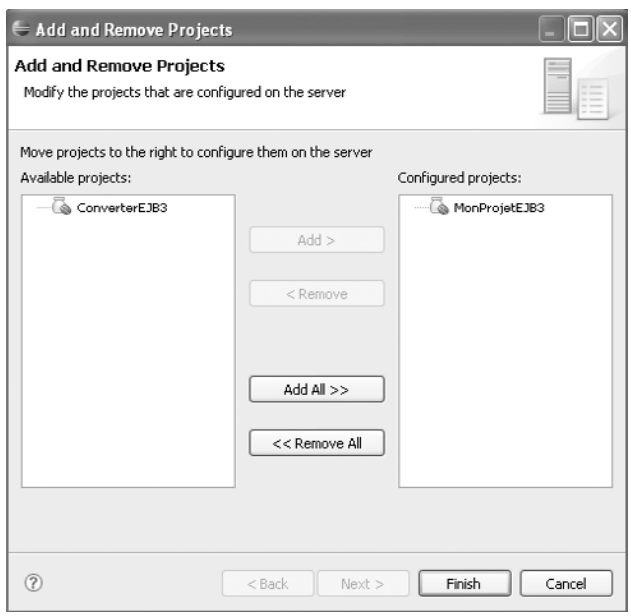
Le code de vos beans session stateless et stateful étant écrit (SearchFacade et ShoppingCart), il ne vous reste qu'à le tester par l'écriture d'une application cliente Java de type Remote.

Eclipse fournit un assistant de packaging et de déploiement des EJB relativement intuitif. Il se présente sous la forme d'une archive EAR qui sera déployée sur la configuration de déploiement du serveur JBoss (par défaut <Répertoire d'installation de JBoss>\server\default\deploy). Nous supposons que, durant toutes les étapes du packaging et du déploiement du projet, votre serveur JBoss a bien été démarré.

Vous devez déployer votre projet sur le serveur JBoss configuré *via* l'option Publish du menu contextuel Eclipse disponible dans la vue serveur, en ayant pris soin d'associer le projet à la configuration (voir figure 9.9).

Figure 9.9

Assistant de configuration du déploiement de projet Web Tools



Votre projet est désormais entièrement recompilé et redéployé automatiquement à chaque sauvegarde de chaque ressource fichier du projet (voir figure 9.10).

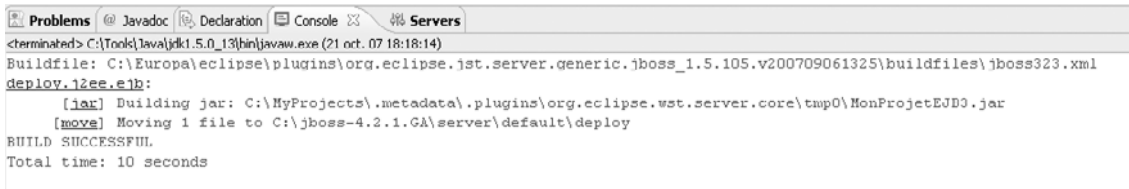


Figure 9.10

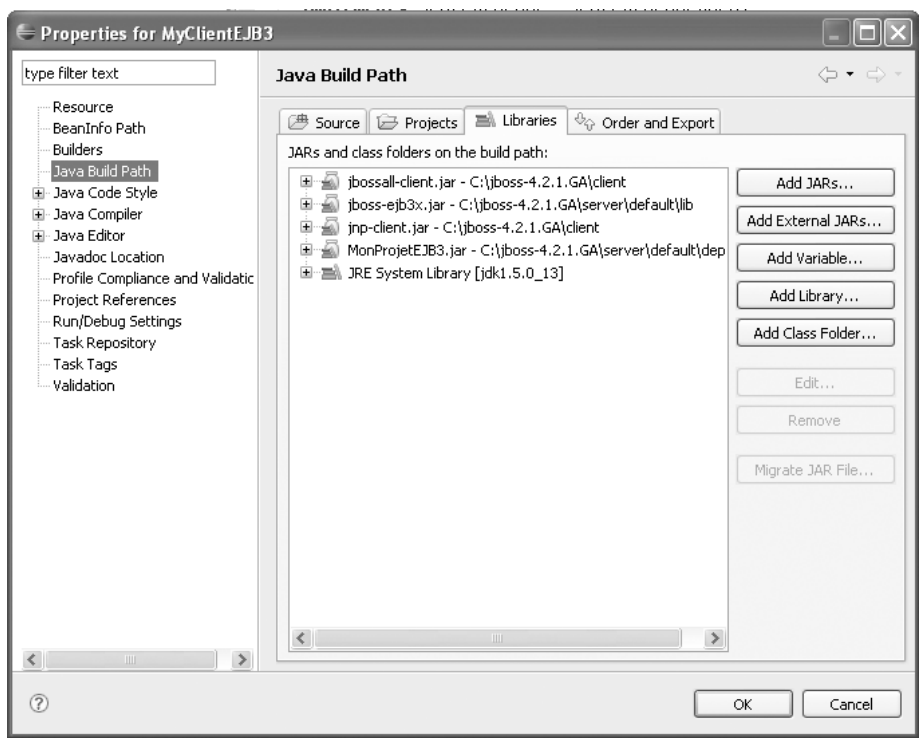
Déploiement du projet

Écriture du client Java

1. Créez un projet de type Java *via* l'assistant de création de projet Eclipse.
2. Ajoutez les bibliothèques clientes illustrées à la figure 9.11 dans le chemin de compilation des classes (y compris le jar du projet EJB3 déployé sous JBoss).

Figure 9.11

Ajout des jar nécessaires à l'application cliente



3. Dans l'option Project References, ajoutez la référence au projet EJB MonProjetEJB3.
4. Ajoutez les classes Java de type client distant suivantes (remarquez dans l'extrait l'invocation de l'interface distante SearchFacade/remote dans le lookup JNDI) :

Voici le code du client correspondant à l'invocation distante des méthodes métier du bean session sans état, l'invocation pour le bean avec état ShoppingCart se faisant de manière similaire :

```
package com.eyroles.chapitre09.client;

import com.eyrolles.chapitre09.*;

import java.util.List;

import java.util.Properties;
import javax.ejb.EJB;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class SearchFacadeClient {


public SearchFacadeClient() {

}

public static void main(String[] args) {

  try {
         Context context = new InitialContext();
         SearchFacade beanRemote = (SearchFacade)context.lookup("SearchFacade/remote");
         List winesList = beanRemote.articleSearch("Disque Dur");
         System.out.println("Affichage liste articles");
         for (String article: (List<String>)articlesList) {
             System.out.println(article);
           }

      } catch (NamingException e) {
         e.printStackTrace();
      }
   }

}
```

Configuration Eclipse/Web Tools/JBoss

La configuration utilisant l'injection de dépendances avec JNDI (en particulier l'utilisation de la balise @EJB) n'est apparemment pas fonctionnelle avec la configuration Eclipse/Web Tools/JBoss. Pour tester vos beans session, vous devez utiliser la configuration plus traditionnelle *via* le look-up JNDI, mais bien sûr « façon EJB3 ». Nous donnons toutefois le code utilisant l'injection de dépendances, que vous pourrez utiliser avec d'autres environnements, comme Ant.

En résumé

Ce chapitre vous a permis de découvrir les aspects innovants du développement EJB3 et les fondamentaux de ce nouveau modèle, en particulier pour le développement des beans session.

Vous verrez au chapitre suivant les aspects liés aux beans entité et à l'API Java Persistence, autre apport important de la spécification EJB3.