TP – Architectures des applications persistantes

Gestion du personnel

UVFIP INF210 – INF211

Année 2015-2016

M.T. Segarra, P. Tanguy

Objectifs

L'objectif de ce travail est d'effectuer le développement d'une application organisée en trois couches et déployée selon une architecture 3-tier en utilisant une plate-forme Java EE et, en particulier, le serveur d'applications Glassfish. L'intérêt de ce travail est double. Il permettra aux étudiants :

- de comprendre les avantages des architectures 3-tier en termes de structuration des fonctions d'une application persistante ainsi que des facilités de programmation offertes par les serveurs d'applications en général ;
- de mettre en pratique des technologies Java pour la construction d'applications 3-tier ;
- d'être confrontés aux différents choix de conception et de localisation des traitements d'une application.

1 Organisation du travail

Le travail est structuré en quatre parties principales :

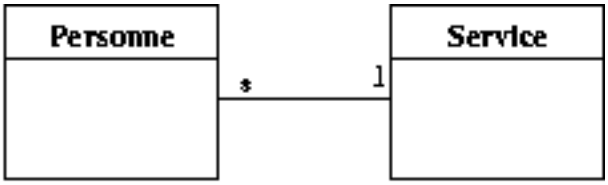
- la première partie concerne l'étude du schéma de base de données utilisé par l'application de gestion de personnel présentée dans le paragraphe 2. Vous devez uniquement expérimenter avec les fonctionnalités qui sont déjà mises en œuvre ;
- la deuxième partie consiste à étudier et comprendre les limites d'une architecture en 2 couches déployée selon le modèle 2-tier de l'application ayant les fonctionnalités ci-après ;
- la troisième partie consiste à proposer une version en 3 couches et déployée selon le modèle 3-tier de l'application ayant les fonctionnalités présentées ci-après.

2 Présentation de l'application

L'application proposée concerne la gestion du personnel d'une entreprise.

2.1 Schéma conceptuel de données

La figure ci-dessous donne une version simplifiée du schéma conceptuel de données manipulées par cette application. Les employés sont regroupés par service. Dans un service on peut avoir plusieurs employés alors qu'un employé ne peut appartenir qu'à un seul service à un moment donné.



2.2 Fonctionnalités de l'application

Nous souhaitons que notre application, offre quatre fonctionnalités :

- la consultation des services (dans le sens de départements) d'une entreprise ;
- l'ajout d'un service (dans le sens de département) ;
- la consultation de l'annuaire de l'entreprise (dans le sens personnel de l'entreprise) ;
- l'embauche d'une personne dans un département.

3 Les outils

Pour réaliser le TP, vous utiliserez l'IDE Eclipse, un serveur d'applications Glassfish qui agit en temps que conteneur Web et EJB et un serveur de base de données PostgreSQL.

Pour limiter le travail de configuration, nous avons créé une machine virtuelle VMWare. Le document *« Description de l'environnement de travail »*, disponible sur Moodle, introduit brièvement la virtualisation de manière générale puis présente les caractéristiques de la machine virtuelle que nous avons mis à votre disposition ainsi que son utilisation. Un autre document disponible aussi sur Moodle, *« Installation et configuration des logiciels de l'UV »*, décrit comment faire les installations nécessaires au TP sur votre machine personnelle.

4 Organisation des sources

Vous trouverez dans le répertoire `/home/user/tpArchis` de la machine virtuelle, les sources nécessaires à la réalisation de ce TP :

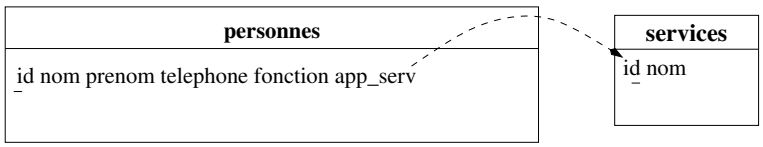
- un *script* SQL `install_baseGestPers.sql` pour la création rapide du schéma de base de données et pour l'introduction de quelques données dans la base ;
- un répertoire `gestpers2Tier` correspondant au projet Eclipse avec les sources de la version 2 couches et 2-tier de l'application ;

— un répertoire `gestpers3Tier` contenant 3 projets Eclipse (`gestpers3Tier-ear`, `gestpers3Tier-web`, et `gestpers3Tier-ejb`) avec les sources de la version 3 couches et 3-tier de l'application ;
— un répertoire `EJBRemoteClient` correspondant au projet Eclipse avec le squelette d'une interface Java Swing très simple pour l'application.

Par la suite nous appellerons `$CP_PROJ$` le répertoire où se trouvent ces différents éléments.

5 Schéma relationnel de données

Le schéma logique de la base de données utilisée par notre application consiste en deux tables présentées dans la figure ci-après et une vue. La table *services* contient des informations sur les différents services existant dans la société. La table *personnes* contient des informations sur le personnel travaillant dans la société. Les flèches en pointillées indiquent l'existence d'une contrainte de clé étrangère entre les attributs.



6 Accès direct à la base

Dans la première partie du TP, vous allez manipuler le serveur de bases de données PostgreSQL en utilisant le client disponible avec Eclipse.

Un serveur PostgreSQL est automatiquement lancé lors de l'exécution de la machine virtuelle. Il écoute sur le port 54321 de *localhost* et contient la base de données utilisée dans ce TP, *gestPersDB*. Pour vous connecter à cette base de données avec Eclipse utilisez l'onglet *Data Source Explorer* puis déroulez *Database Connections*. Sélectionnez la connexion `PostgreSQL-gestpersDB` déjà créée et, à l'aide du menu contextuel, demandez à vous connecter. Le mot de passe demandé est *dbpwd*.

Maintenant que vous êtes connectés à votre base de données, vous allez créer votre schéma logique de données. Pour ceci, ouvrez le *script* `$CP_PROJ$/install_baseGestPers.sql` avec l'éditeur d'Eclipse (menu *File -> Open File*) et demandez son exécution sur votre base de données (choisissez les bonnes options dans le *Connection profile* puis, à l'aide du menu contextuel, demandez l'exécution du *script*).

7 Architecture 2-tier

La version 2-tier de l'application offre une interface Web pour accéder à la base de données et pour générer la présentation des résultats.

Dans cette version de l'application, un navigateur Web joue le rôle d'interface avec l'utilisateur. Il obtient les pages HTML correspondant à ses requêtes (par exemple annuaire des employés) de la part du serveur Web. Ces pages HTML étant dynamiques [1], le serveur Web utilise des programmes

1. Une page HTML dynamique contient des informations qui peuvent changer au cours du temps. Ainsi, par exemple, les employés de la société à un instant ne sont pas forcément les mêmes qu'à l'instant t+1. Généralement, ces informations dynamiques sont obtenues en accédant à une base de données.

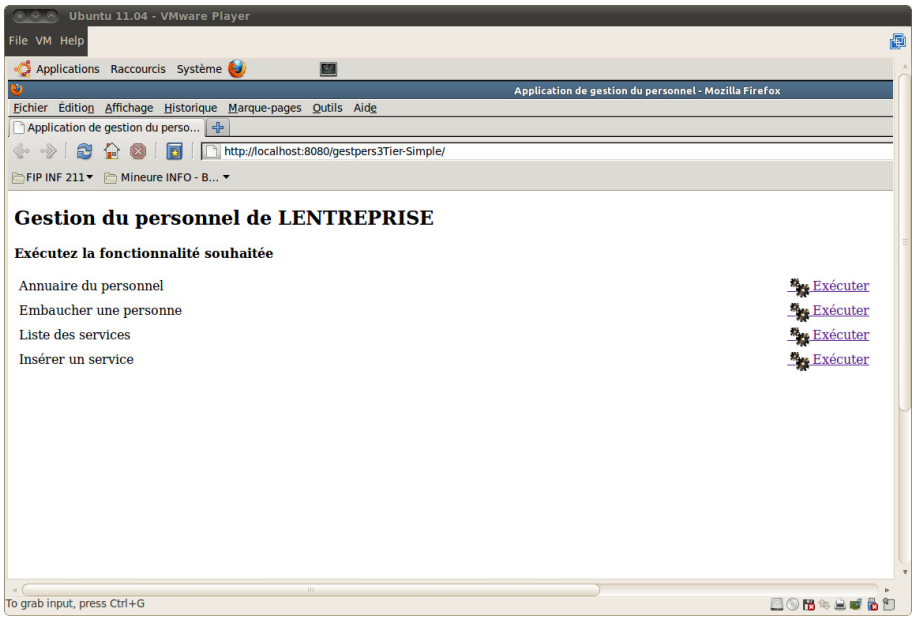


FIGURE 1 – Page d'accueil de l'application de gestion du personnel

particuliers que l'on appelle *servlets* pour les construire. Ces programmes (Java) sont exécutés par un *conteneur Web* (dans notre cas Glassfish) et leur rôle est de répondre à des requêtes HTTP par des pages HTML dont le contenu n'est pas fixe. Pour compléter la partie variable d'une page, nos *servlets* accèdent à la base de données en utilisant l'interface standard JDBC (« *Java Database Connectivity* »).

Ouvrez le projet `gestpers2Tier` dans Eclipse (menu *File -> Import*).

7.1 Exécution de l'application

Vous allez maintenant exécuter l'application qui vous est fournie. Pour cela, vous devez la déployer sur le conteneur Web qui sait l'exécuter. Comme mentionné précédemment, il s'agit pour nous de Glassfish. Utilisez le menu contextuel *Run As* qui apparaît lorsque vous sélectionnez votre projet. Vous pouvez ensuite accéder à l'application en utilisant un navigateur Web avec l'URL `http://localhost:8080/gestpers2Tier`.

Vérifiez que votre application est bien déployée sur Glassfish en allant dans l'onglet *Servers* du navigateur d'Eclipse et en sélectionnant Glassfish.

Lorsque vous accédez à l'adresse `http://localhost:8080/gestpers2Tier` vous demandez à Glassfish d'accéder à la page *index.jsp* de votre application [2]. Elle crée la page HTML de la figure 1 qui vous permet d'accéder aux pages générées par les autres *servlets*.

Enfin, testez le bon fonctionnement de l'application.

2. Vous pouvez voir le contenu de cette page à l'aide d'Eclipse : soit dans l'onglet *Project Explorer* dans la partie *WebContent* de votre projet, soit dans le répertoire `gestpers2Tier/WebContent` dans l'onglet *Navigator*

7.2 Dans les sources

Pour l'application proposée nous avons developpé quatre *servlets* dont les sources se trouvent dans `$CP_PROJ$/gestpers2Tier/src`. Chacune de ces *servlets* contient une méthode `doGet` (et `doPost`) qui est exécutée à chaque fois qu'une requête HTTP GET (ou POST) doit être traitée [3]. C'est l'exécution de cette méthode qui va permettre de générer le contenu de la page qui sera affichée sur le navigateur. Ces méthodes contiennent deux paramètres :

- un objet `HttpServletRequest` qui permet de récupérer les paramètres éventuels passés à la *servlet* ;
- un objet `HttpServletResponse` dans lequel la *servlet* écrit le contenu HTML à afficher. La composition de la page HTML comporte une partie statique (en-tête, corps du document HTML, titre...) et une partie dynamique écrites sur le flot de sortie. Pour récupérer cette dernière, la *servlet* se connecte directement à la base de données qui contient les informations requises.

7.3 Les *servlets* (optionnel)

Analysez les sources de la *servlets* `AnnuaireServlet`.

▷ **Question 0.1 :**
Où est générée la partie statique de la page HTML de la *servlet* ? Où est générée la partie dynamique ?

▷ **Question 0.2 :**
Qui est en charge des couches présentation, traitement et gestion de données ?

▷ **Question 0.3 :**
Pourquoi cette application a une architecture en deux couches ?

▷ **Question 0.4 :**
Pourquoi cette application a une architecture 2-tier ?

▷ **Question 0.5 :**
On souhaite créer une interface Java (i.e. Swing) pour l'application. Quelles classes et/ou méthodes pourraient être réutilisées par cette nouvelle interface ? Comment cette réutilisation pourrait être mise en place ?

▷ **Question 0.6 :**
Aurait-il été possible d'exécuter le serveur HTTP sur une machine différente de celle qui exécute le navigateur Web ?

8 Architecture à trois couches déployée selon le modèle 3-tier

La version de l'application sur laquelle vous allez travailler dans cette partie concerne une architecture 3 couches et 3-tier de type Java EE offrant une interface Web.

Comme dans la version précédente de l'application, des *servlets* sont utilisées pour générer les pages HTML en réponse à des requêtes HTTP. À la différence de la version précédente, ces *servlets*

3. Une requête d'un de ces types est générée par le navigateur à chaque demande d'affichage d'une page HTML.

n'accèdent pas directement à la base de données pour compléter les parties variables d'une page mais utilisent des EJB *(« Entreprise Java Beans »)*.

Un EJB est un logiciel écrit en Java qui doit s'exécuter dans un *conteneur EJB*. Ce conteneur lui offre des services destinés à faciliter notamment la programmation des transactions, de la sécurité et de l'accès à distance. C'est pour cette raison qu'ils sont utilisés pour contenir la logique métier dans une application 3-tier.

Pour simplifier la configuration, nous avons choisi d'utiliser un même outil pour le déploiement des *servlets* et des EJB, i.e., Glassfish. Il joue donc le rôle de conteneur Web et de conteneur EJB. En particulier cet outil intègre :

- un serveur Web (ou HTTP). Il écoute sur le port 8080, reçoit les requêtes des navigateurs Web et les transmet au conteneur Web ;
- un conteneur Web, qui exécute les *servlets* de l'application ;
- un conteneur EJB, qui exécute les EJBs de l'application.

L'application est constituée de quatre projets Eclipse `gestpers3Tier-*`. Importez ces projets (menu *File -> Import*) pour accéder aux sources. Attention, sélectionnez le répertoire dans lequel se trouvent les projets. Ceci permettra d'importer les quatre projets à la fois.

8.1 Exécution de l'application

Vous allez maintenant exécuter l'application qui vous est fournie. Pour cela, vous devez déployer les *servlets* sur le conteneur Web et les EJBs sur le conteneur EJB. Comme mentionné précédemment, nous utilisons un seul outil (Glassfish) qui joue le rôle de conteneur Web et EJB. Vous devez donc simplement déployer le projet `gestpers3Tier-ear` sur le serveur Glassfish. Pour cela, utilisez le menu contextuel *Run As* qui apparaît lorsque vous sélectionnez le projet. Vous pouvez ensuite accéder à l'application en utilisant un navigateur Web ouvert avec l'URL `http://localhost:8080/gestpers3Tier-web`.

Vérifiez que votre application est bien déployée en allant dans l'onglet *Servers* du navigateur d'Eclipse et en sélectionnant Glassfish.

Lorsque vous accédez à l'adresse `http://localhost:8080/gestpers3Tier-web` vous demandez à Glassfish d'accéder à la page *index.jsp* de votre application [4]. Elle crée la page HTML de la figure 1 qui vous permet d'accéder aux pages générées par les *servlets*.

Testez le bon fonctionnement de l'application. Attention, la seule fonctionnalité fournie est la liste des services de l'entreprise.

8.2 Dans les sources

L'application est constituée principalement d'un module WAR [5] et d'un module EJB [6], regroupés dans un projet EAR `gestpers3Tier-ear` :

4. Vous pouvez voir le contenu de cette page à l'aide d'Eclipse : soit dans l'onglet *Project Explorer* dans la partie *WebContent* du projet `gestpers3Tier-web`, soit dans le répertoire `gestpers3Tier-web/WebContent` dans l'onglet *Navigator*.

5. Terme utilisé dans les spécifications Java EE pour désigner un logiciel constitué au moins d'un élément générant des pages HTML dynamiques et, donc, déployable dans un conteneur Web.

6. Terme utilisé dans les spécifications Java EE pour désigner un logiciel constitué au moins d'un EJB et, donc, déployable dans un conteneur EJB.

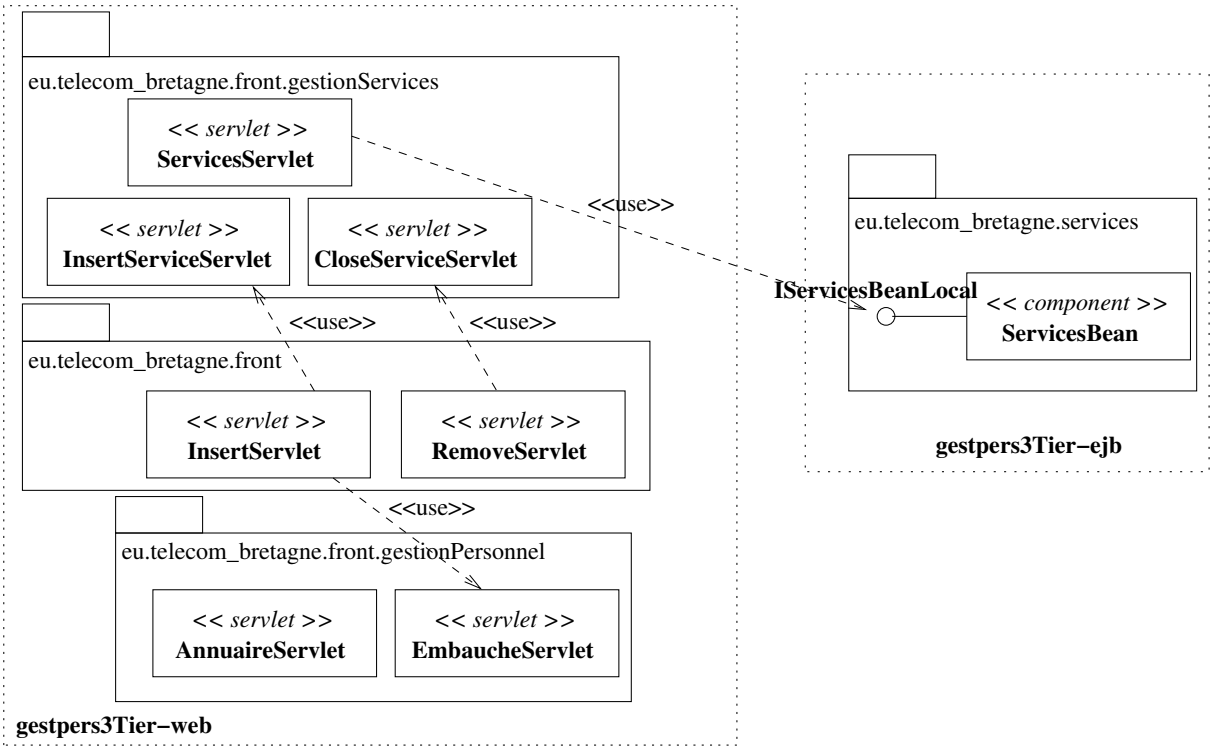


FIGURE 2 – Structuration des sources : gestpers3Tier-web

— le projet `gestpers3Tier-web` correspond au module WAR de l'application et contient donc, principalement nos *servlets* ;
— le projet `gestpers3Tier-ejb` correspond au module EJB de l'application et contient donc principalement nos EJBs ;
— le projet `gestpers3Tier-ear` qui regroupe les deux modules précédents. Il représente donc notre application ;

Plus spécifiquement, le projet `gestpers3Tier-web` contient toutes les *servlets* nécessaires à notre application. Notamment :

— le paquetage `eu.telecom_bretagne.front` contient une *servlet* `InsertServlet` qui construit le formulaire pour ajouter une personne ou un service ;
— le paquetage `eu.telecom_bretagne.front.gestionServices` contient les *servlets* `ServicesServlet` et `InsertServiceServlet`. La première construit la page HTML pour l'annuaire des services. La deuxième ajoute un service correspondant aux informations du formulaire d'ajout ;
— le paquetage `eu.telecom_bretagne.front.gestionPersonnel` contient les *servlets* `AnnuaireServlet` et `EmbaucheServlet`. La première construit la page HTML pour l'annuaire du personnel. La deuxième ajoute un nouveau salarié dans la base de données.

Concernant le projet `gestpers3Tier-ejb` il regroupe l'ensemble de classes utilisées par les *servlets* pour accéder à la base de données. Il contient notamment :

— deux classes (`Service` et `Personne` dans le paquetage `eu.telecom_bretagne.data.model`) qui spécifient le mapping objet-relationnel pour les services et le personnel dans la base de données. Il s'agit donc de nos entités dans la terminologie JPA ;
— deux classes (EJBs), `ServicesBean` (dans `eu.telecom_bretagne.services`) et `ServicesDAO`

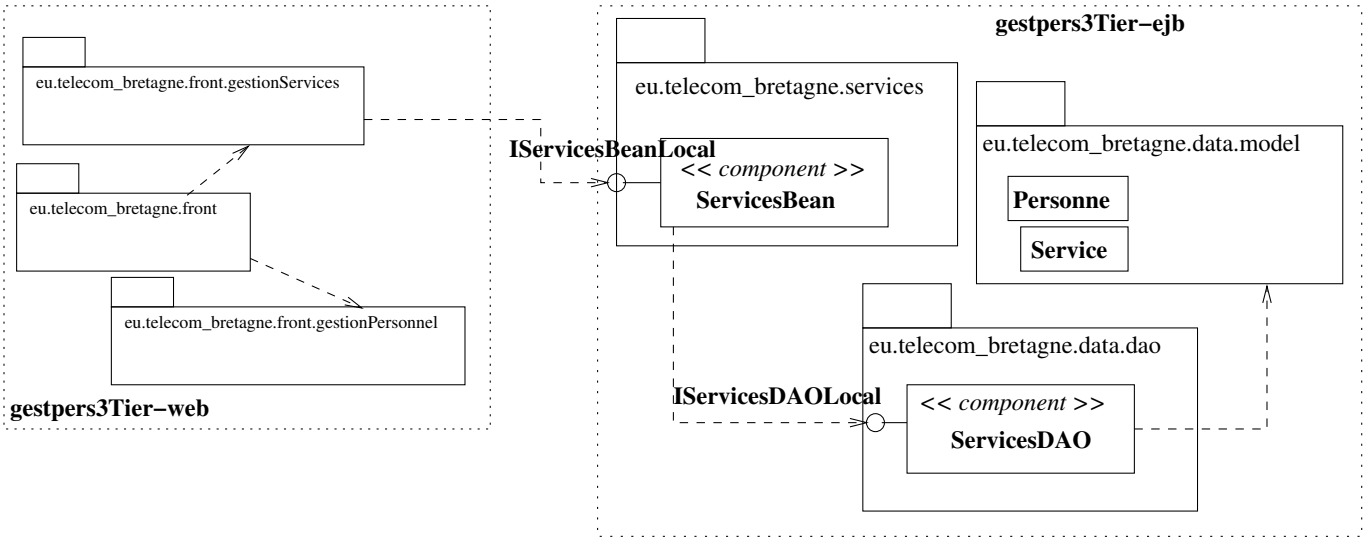


FIGURE 3 – Structuration des sources : gestpers3Tier-ejb

(dans `eu.telecom_bretagne.data.dao`). La première fournit une méthode permettant d'obtenir des informations sur les services existant dans l'entreprise. Il utilise pour ceci la deuxième, dont le rôle est de masquer les accès aux entités.

L'architecture en paquetages de ce projet est représentée dans la figure 3.

8.3 Le projet `gestpers3Tier-ejb` : le métier et l'accès aux données

8.3.1 Injection de dépendances

Analysez les sources de la classe `ServicesBean` se trouvant dans le paquetage `eu.telecom_bretagne.gestionServices`.

`ServicesBean` est un EJB et pas une classe standard Java. Pour indiquer qu'une classe Java est un EJB, il suffit d'ajouter l'annotation `@Stateless` juste avant la définition de la classe. Vérifiez que c'est le cas de notre classe `ServicesBean`.

Pour que l'EJB `ServicesBean` accède aux services de la classe `ServicesDAO` il utilise le mécanisme d'*injection de dépendances* (ici en utilisant l'annotation `@EJB`). Ce mécanisme consiste à demander au conteneur EJB de créer une instance, le programmeur n'a donc pas besoin de le faire. Dans le code, la référence `sDAO` sera initialisée par le conteneur EJB avec un objet de type `IServicesDAOLocal`.

8.3.2 `EntityManager` et conteneur EJB

Concernant le modèle de programmation des `EntityManager`, il est différent en fonction que l'on soit dans un conteneur EJB ou pas.

Création/destruction d'instances. Dans le cas de l'exécution au sein d'un conteneur EJB, le mécanisme d'injection de dépendances est utilisé pour l'initialisation des références à la classe `Entitymanager`. Cette initialisation se fait dans les classes de type DAO (`ServicesDAO` pour notre

application) et pas dans les classes métier. Identifiez cette référence dans la classe `ServicesDAO` : elle est précédée de l'annotation `@PersistenceContext` ; le conteneur EJB se chargera d'instancier la classe `EntityManager` pour initialiser la référence `em`. Le paramètre de cette annotation donne au conteneur EJB les informations nécessaires à la connexion à la base de données. Ces informations ont été configurées directement sur Glassfish via l'interface d'administration disponible à `localhost:4848`.

Gestion des transactions. Concernant la délimitation des transactions, elle devrait apparaître dans les classes métier (pour notre application, `ServicesBean`). Or, aucun code correspondant aux transactions n'est présent. En effet, pour un EJB et par défaut, une transaction est ouverte au début de toute méthode. De même, la transaction est fermée à la fin de la méthode.

En résumé, concernant le modèle de programmation des `EntityManager` :

- les EJBs sont instanciés par le conteneur EJB (`@EJB`) ;
- la classe `EntityManager` est instanciée et les instances détruites par le conteneur EJB (`@PersistenceContext`) ;
- les transactions sont délimitées par le conteneur EJB.

8.4 Le projet `gestpers3Tier-web` : la présentation

Enfin, analysez les sources du module WAR (projet `gestpers3Tier-web`). Chacune des *servlets* fournies contient une méthode `doGet` (et `doPost`) qui est exécutée à chaque fois qu'une requête HTTP GET (ou POST) doit être traitée [7]. C'est l'exécution de cette méthode qui va permettre de générer le contenu de la page qui sera affichée sur le navigateur. La méthode `doGet` (et `doPost`) contient deux paramètres :

- un objet `HttpServletRequest` qui permet de récupérer les paramètres éventuels passés à la *servlet* ;
- un objet `HttpServletResponse` dans lequel la *servlet* écrit le contenu HTML à afficher. La composition de la page HTML comporte une partie statique (en-tête, corps du document HTML, titre...) et une partie dynamique écrites sur le flot de sortie. Pour récupérer cette dernière, la *servlet* utilise un EJB session.

Analysez les sources de la *servlet* `ServicesServlet`.

▷ **Question 0.7 :**
Comment obtient elle une référence à l'EJB `ServicesBean` qu'elle utilise ?

9 L'accès à distance des EJBs

On souhaite maintenant créer une interface client lourd Java Swing pour notre application. Cette interface utilisera les EJBs de l'application déployés sur le serveur Glassfish pour accéder aux fonctionnalités de celle-ci.

Vous trouverez dans `$CP_PROJ$` le projet Eclipse `EJBRemoteClient` qui contient les sources d'un squelette d'application Java Swing minimaliste. Importez ce projet dans Eclipse et lancez son exécution en sélectionnant la classe `Main` puis *Run As -> Java Application*. Une fenêtre s'ouvre alors avec un bouton *Access EJB*. Actuellement un simple message s'affiche lorsque vous appuyez

7. Une requête de ces types est générée par le navigateur à chaque demande d'affichage d'une page HTML.

sur le bouton. L'objectif est d'ajouter le code nécessaire dans la classe `Main` pour que lorsque le bouton est appuyé un appel à un EJB se produise.

9.1 Coté *serveur*

L'appel à un EJB à partir de notre interface Swing est un appel *distant*, i.e., il provient d'un logiciel qui s'exécute sur une autre JVM que celle où l'EJB se trouve. Il faut donc créer une interface accessible à distance pour les EJBs dont on souhaite avoir ce type d'accès. Suivez le guide ...

- Ouvrez dans l'éditeur d'Eclipse l'EJB `ServicesBean`. Modifiez la définition de la classe pour indiquer qu'elle réalise aussi une interface `IServicesBeanRemote`. Ajoutez lui l'annotation `@Remote` ;
- Ajoutez à cette nouvelle interface une méthode `listeServices` ayant la même signature que celle présente dans la classe et déclarée dans l'interface `IServicesBeanLocal` ;

9.2 Coté *client*

Maintenant on va écrire le code nécessaire coté client pour accéder aux EJBs. Ici, le mécanisme d'injection de dépendances ne peut pas être utilisé, le client n'étant pas déployé sur le serveur Glassfish où se trouvent les EJBs. La solution qui existe dans Java EE est l'utilisation de la librairie JNDI (*« Java Naming and Directory Interface »*) que tout conteneur EJB fournit : à partir du nom d'un EJB, JNDI renvoie le *stub* qui permettra l'accès distant à l'EJB. Lisez la documentation JNDI à l'adresse `http://docs.oracle.com/javaee/6/tutorial/doc/gipjf.html`.

Analysez les sources de la classe `Main` du projet `EJBRemoteClient`. Dans la méthode `actionPerformed` se trouve une partie du code nécessaire à l'appel à l'EJB en utilisant JNDI. Complétez ce code pour que l'appel ait effectivement lieu. Pour cela vous devez indiquer le nom de l'EJB à utiliser. Lors du déploiement de l'application, Glassfish indique dans son log le nom global des EJBs déployés.

▷ **Question 0.8 :**
Quel est le nom de l'EJB `ServicesBean` ?

▷ **Question 0.9 :**
Complétez le code du client pour qu'il réalise effectivement l'appel et qu'il affiche le résultat de celui-ci.

Pour aller plus loin dans la compréhension de JNDI et de l'accès distant à des EJB, vous trouverez à l'adresse `http://docs.oracle.com/cd/E18930_01/html/821-2418/beans.html#beanw` et `https://blogs.oracle.com/chengfang/entry/glassfish_to_glassfish_remote_ejb` les informations pertienentes.

10 Enrichir la version 3-tier

Dans la version actuelle, c'est la fonctionnalité consistant à lister les services de l'entreprise qui est entièrement fonctionnelle : `ServicesServlet` utilise l'EJB `ServicesBean` pour obtenir la liste des services. Et celui-ci utilise l'EJB `ServicesDAO` pour « calculer » cette liste.

Les fonctionnalités à ajouter à l'application sont :

— ajouter un service ;
— supprimer un service ;
— service d'annuaire ;
— embauche d'une personne et
— licencier une personne.

Toutes les *servlets* correspondant à ces fonctionnalités sont disponibles dans le projet `gestpers3Tier-web`. Vous devrez uniquement les compléter avec les appels aux bons EJBs. Inspirez-vous de la *servlet* `ServicesServlet` pour l'implanter.

Commencez par ajouter une méthode `insertService(String name)` à l'EJB `ServicesBean`. Puis modifiez l'EJB `ServicesDAO` si nécessaire. Ensuite, ajoutez dans la *servlet* `InsertServiceServlet` l'appel à la nouvelle méthode créée dans l'EJB `ServicesBean`. Testez enfin le bon fonctionnement de la fonctionnalité.

Procédez de la même manière pour les deux autres fonctionnalités.

11 Les aspects transactionnels (optionnel)

Comme mentionné précédemment, tous les composants EJB sont des composants transactionnels, même si à aucun moment vous n'avez écrit du code en relation aux transactions. Dans cet exercice nous allons introduire des aspects transactionnels associés aux EJBs.

Lisez les pages 239-243 de l'annexe 14.1.

▷ **Question 0.10 :**
Quel doit être le type des transactions utilisées par notre composant de gestion du personnel ? La ressource devra-elle de type XA ? Justifiez votre réponse.

Dans l'application que nous utilisons la valeur par défaut s'applique pour la gestion des transactions. Cette valeur peut être modifiée à l'aide de l'annotation `@TransactionAttribute`. Lisez maintenant les pages 243-247 de l'annexe 14.1.

▷ **Question 0.11 :**
Quelle est la valeur par défaut pour cette annotation et qu'est-ce qu'elle signifie ?

Nous allons modifier la valeur de cette annotation pour modifier le fonctionnement de la méthode `closeService` de l'EJB `ServicesBean`. Cette méthode recherche le service à supprimer de la base, cherche les personnes qui y travaillent, les licencie puis supprime le service.

▷ **Question 0.12 :**
Modifiez l'attribut transactionnel de cette méthode pour qu'elle n'accepte pas des appels transactionnels.

Pour voir les effets d'autres valeurs possibles de cette attribut, il faut d'abord modifier le niveau des traces du service transactionnel de Glassfish. Pour ceci, accéder avec le navigateur à l'adresse `http://localhost:4848/common/index.jsf` (interface d'administration de Glassfish) et sélectionnez *Configurations -> Logger Settings* puis l'onglet *Log Levels*. Demandez à avoir un niveau *FINE* pour les propriétés *javax.enterprise.system.core.transaction* et *javax.enterprise.resource.jta*. Ensuite, assurez vous que le serveur collecte des informations sur les transactions effectuées. Pour cela utilisez à nouveau l'interface d'administration de Glassfish et sélectionnez *Monitoring Data*. Demandez à configurer le *monitoring* et en particulier le *Transaction Service*. Une fois ces modifications effectuées, redémarrez le serveur Glassfish.

Modifiez ensuite l'attribut transactionnel de la méthode `remove` de `PersonnelDAO` et utilisez les

informations monitorées par Glassfish pour connaître le nombre de transactions réellement effectuées.

12 Un petit exercice final

Ce que nous avons vu dans ce TP ce sont les aspects distribution, injection de dépendances et aspect transactionnels associés aux composants EJB. Ce sont les conteneurs qui les exécutent qui offrent ces services et des facilités de programmation de ces aspects qui ont fait des composants EJB un des modèles de programmation les plus utilisés pour le développement d'applications au sein des SIs des entreprises.

Nous allons voir dans cette section un aspect pour lequel les EJB n'ont pas apporté de solution : la gestion de plusieurs implantations d'une même interface.

▷ **Question 0.13 :**
Ajoutez à votre application un nouveau EJB `PDAO` qui réalise l'interface `IPersonnelDAOLocal`. Tentez de déployer votre application sur Glassfish. Quel est le résultat ?

D'autres platformes à composants se sont intéressés à ce type de problème, notamment OSGi. Si vous etes intéressés par le sujet, une bonne référence est le livre « OSGi in Depth » de Alexandre de Castro Alves, ed. Manning.

13 Quelques références

- Documentation JavaDoc pour Java EE `http://download.oracle.com/javaee/6/api/`
- Tutoriel des technologies Java EE `http://download.oracle.com/javaee/6/tutorial/doc/`
- Des informations sur comment construire un client d'une application d'entreprise ainsi que sur le nommage JNDI pour les EJB `https://glassfish.dev.java.net/javaee5/ejb/EJB_FAQ.html`
- Des informations sur Glassfish, le serveur d'applications utilisé. Vous trouverez notamment un document qui explique le déploiement d'applications sur le serveur `https://glassfish.dev.java.net/docs/index.html`
- Un document intéressant sur les annotations `http://download-llnw.oracle.com/docs/cd/E12840_01/wls/docs103/ejb30/annotations.html`
- Un document dédiée aux annotations pour définir le mapping O/R `http://docs.jboss.org/hibernate/stable/annotations/reference/en/html/entity.html`

14 Annexes

14.1 Introduction aux transactions dans les EJBs

Extrait du livre *« Beginning Java EE 6 Platform with GlassFish 3 »*, Antonio Goncalves, Apress, 2010.

CHAPTER 9

Transactions and Security

Transaction and security management are important matters for enterprises. They allow applications to have consistent data and secure the access to it. Both services are low-level concerns that a business developer shouldn't have to code himself. EJBs provide these services in a very simple way: either programmatically with a high-level of abstraction or declaratively using metadata.

Most of an enterprise application's work is about managing data: storing it (typically in a database), retrieving it, processing it, and so on. Often this is done simultaneously by several applications attempting to access the same data. A database has low-level mechanisms to preserve concurrent access, such as pessimistic locking, and uses transactions to ensure that data stays in a consistent state. EJBs make usage of these mechanisms.

Securing data is also important. You want your business tier to act like a firewall and authorize some actions to certain groups of users and deny access to others (e.g., only employees are allowed to persist data, but users and employees are authorized to read data).

The first part of this chapter is devoted to exploring transaction management in EJB 3.1. I'll introduce transactions as a whole, and then discuss the different types of transaction demarcation supported by EJBs. In the second part of the chapter, I'll focus on security.

Transactions

Data is crucial for business, and it must be accurate regardless of the operations you perform and the number of applications concurrently accessing it. A *transaction* is used to ensure that the data is kept in a consistent state. It represents a logical group of operations that must be performed as a single unit, also known as a *unit of work*. These operations can involve persisting data in one or several databases, sending messages, or invoking web services. Companies rely on transactions every day for their banking and e-commerce applications or business-to-business interactions with partners

These indivisible business operations are performed either sequentially or in parallel over a relatively short period of time. Every operation must succeed in order for the transaction to succeed (we say that the transaction is committed). If one of the operations fails, the transaction fails as well (the transaction is rolled back). Transactions must guarantee a degree of reliability and robustness and follow the ACID properties.

ACID

ACID refers to the four properties that define a reliable transaction: Atomicity, Consistency, Isolation, and Durability (described in Table 9-1). To explain these properties, I'll take the classical example of a banking transfer: you need to debit your savings account to credit your current account.

Table 9-1. *ACID Properties*

Property	Description
Atomicity	A transaction is composed of one or more operations grouped in a unit of work. At the conclusion of the transaction, these operations are either all performed successfully (commit), or none of them are performed at all (rollback) if something unexpected or irrecoverable happens.
Consistency	At the conclusion of the transaction, the data is left in a consistent state.
Isolation	The intermediate state of a transaction is not visible to external applications.
Durability	Once the transaction is committed, the changes made to the data are visible to other applications.

When you transfer money from one account to the other, you can imagine a sequence of database accesses: the savings account is debited using a SQL update statement, the current account is credited using a different update statement, and a log is created in a different table to keep track of the transfer. These operations have to be done in the same unit of work (Atomicity) because you don't want the debit to occur but not the credit. From the perspective of an external application querying the accounts, only when both operations have been successfully performed are they visible (Isolation). Consistency is when transaction operations (either with a commit or a rollback) are done within the constraints of the database (such as primary keys, relationships, or fields). Once the transfer is completed, the data can be accessed from other applications (Durability).

Local Transactions

Several components have to be in place for transactions to work and follow the ACID properties. Let's first take the simplest example of an application performing several changes to a single resource (e.g., a database). When there is only one transactional resource, all that is needed is a local transaction. Distributed transactions (à la JTA) can still be used, but are not strictly necessary. Figure 9-1 shows the application interacting with a resource through a transaction manager and a resource manager.

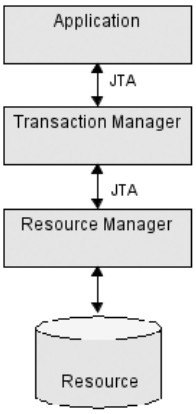


Figure 9-1. *A transaction involving one resource*

The components shown in Figure 9-1 abstract most of the transaction-specific processing from the application:

- The *transaction manager* is the core component responsible for managing the transactional operations. It creates the transactions on the behalf of the application, informs the resource manager that it is participating in a transaction (an operation known as *enlistment*), and conducts the commit or rollback on the resource manager.
- The *resource manager* is responsible for managing resources and registering them with the transaction manager. An example of a resource manager is a driver for a relational database, a JMS resource, or a Java connector.
- The *resource* is persistent storage from which you read or write (a database, a message destination, etc.).

It is not the application's responsibility to preserve ACID properties. The application just decides to either commit or roll back the transaction, and the transaction manager prepares all the resources to successfully make it happen.

In Java EE, these components handle transactions through the Java Transaction API (JTA) specified by JSR 907. JTA defines a set of interfaces for the application to demarcate transactions' boundaries, and it also defines APIs to deal with the transaction manager. These interfaces are defined in the `javax.transaction` package, and some of them are described in Table 9-2.

Table 9-2. *Main JTA Interfaces*

Interface	Description
`UserTransaction`	Defines the methods that an application can use to control transaction boundaries programmatically. It is used by EJBs with bean-managed transaction (BMT) to begin, commit, or roll back a transaction (as discussed in the "Bean-Managed Transaction" section).
`TransactionManager`	Allows the EJB container to demarcate transaction boundaries on behalf of the EJB.

Continued

Table 9-2. *Continued*

Interface	Description
`Transaction`	Allows operations to be performed against the transaction in the target `Transaction` object.
`XAResource`	Serves as Java mapping of the industry standard X/Open XA interface (as discussed in the next section).

XA and Distributed Transactions

As you've just seen, a transaction using a single resource (shown previously in Figure 9-1) is called a local transaction. However, many enterprise applications use more than one resource. Returning to the example of the fund transfer, the savings account and the current account could be in separate databases. You would then need transaction management across several resources, or resources that are distributed across the network. Such enterprise-wide transactions require special coordination involving XA and Java Transaction Service (JTS).

Figure 9-2 shows an application that uses transaction demarcation across several resources. This means that in the same unit of work, the application can persist data in a database and send a JMS message, for example.

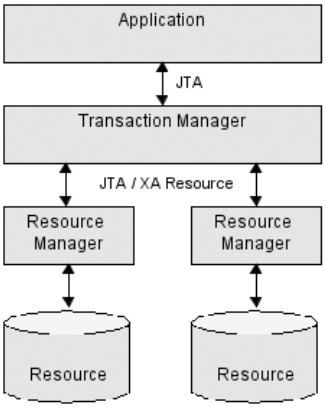


Figure 9-2. *An XA transaction involving two resources*

To have a reliable transaction across several resources, the transaction manager needs to use an XA resource manager interface. XA is a standard specified by the Open Group (`http://www.opengroup.org`) for distributed transaction processing (DTP) that preserves the ACID properties. It is supported by JTA and allows heterogeneous resource managers from different vendors to interoperate through a common interface. XA uses a two-phase commit (2pc) to ensure that all resources either commit or roll back any particular transaction simultaneously.

In our fund transfer example, suppose that the savings account is debited on a first database, and the transaction commits successfully. Then the current account is credited on a second database, but the transaction fails. We would have to go back to the first database and undo the committed changes. To avoid this data inconsistency problem, the two-phase commit performs an additional preparatory step before the final commit as shown in Figure 9-3.

During phase 1, each resource manager is notified through a "prepare" command that a commit is about to be issued. This allows the resource managers to declare whether they can apply their changes or not. If they all indicate that they are prepared, the transaction is allowed to proceed, and all resource managers are asked to commit in the second phase.

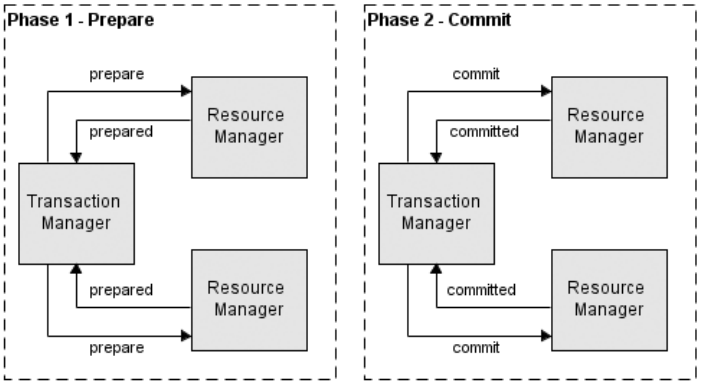


Figure 9-3. *Two-phase commit*

Most of the time, the resources are distributed across the network (see Figure 9-4). Such a system relies on JTS. JTS implements the Object Management Group (OMG) Object Transaction Service (OTS) specification, allowing transaction managers to participate in distributed transactions through Internet Inter-ORB Protocol (IIOP). JTS is intended for vendors who provide the transaction system infrastructure. As an EJB developer, you don't have to worry about this; just use JTA, which interfaces with JTS at a higher-level.

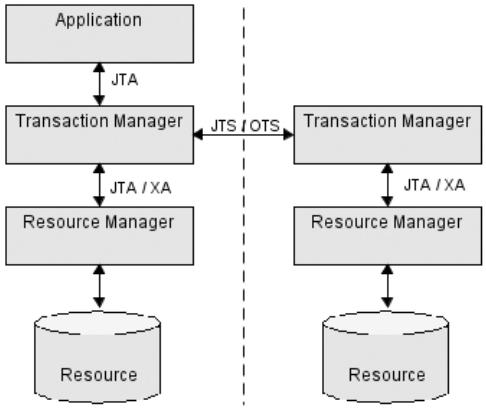


Figure 9-4. *A distributed XA transaction*

Transaction Support in EJB

When you develop business logic with EJBs, you don't have to worry about the internal structure of transaction managers or resource managers because JTA abstracts most of the underlying complexity. With EJBs, you can develop a transactional application very easily,

leaving the container to implement the low-level transaction protocols, such as the two-phase commit or the transaction context propagation. An EJB container is a transaction manager that supports JTA as well as JTS to participate in distributed transactions involving other EJB containers. In a typical Java EE application, session beans establish the boundaries of a transaction, call entities to interact with the database, or send JMS messages in a transaction context.

From its creation, the EJB model was designed to manage transactions. In fact, transactions are natural to EJBs, and by default each method is automatically wrapped in a transaction. This default behavior is known as a container-managed transaction (CMT), because transactions are managed by the EJB container (a.k.a. declarative transaction demarcation). You can also choose to manage transactions yourself using BMTs, also called programmatic transaction demarcation. Transaction demarcation determines where transactions begin and end.

Container-Managed Transactions

When managing transactions declaratively, you delegate the demarcation policy to the container. You don't have to explicitly use JTA in your code (even if JTA is used underneath); you can leave the container to demarcate transaction boundaries by automatically beginning and committing transactions based on metadata. The EJB container provides transaction management services to session beans and MDBs (see Chapter 13 for more on MDBs).

In Chapter 7, you saw several examples of session beans, annotations, and interfaces, but never anything specific to transactions. Listing 9-1 shows the code of a stateless session bean using CMT. As you can see, there is no extra annotation added nor special interface to implement. EJBs are by nature transactional. With configuration by exception, all the transaction management defaults are applied (`REQUIRED` is the default transaction attribute as explained later in this section).

Listing 9-1. *A Stateless Bean with CMT*

```
@Stateless
public class ItemEJB {

    @PersistenceContext(unitName = "chapter09PU")
    private EntityManager em;
    @EJB
    private InventoryEJB inventory;

    public List<Book> findBooks() {
        Query query = em.createNamedQuery("findAllBooks");
        return query.getResultList();
    }

    public Book createBook(Book book) {
        em.persist(book);
        inventory.addItem(book);
        return book;
    }
}
```

You might ask what makes the code in Listing 9-1 transactional. The answer is the container. Figure 9-5 shows what happens when a client invokes the `createBook()` method. The client call is intercepted by the container, which checks immediately before invoking the method whether a transaction context is associated with the call. By default, if no transaction context is available, the container begins a new transaction before entering the method and then invokes the `createBook()` method. Once the method exits, the container automatically commits the transaction or rolls it back (if a particular type of exception is thrown, as you'll see later in the "Exception Handling" section).

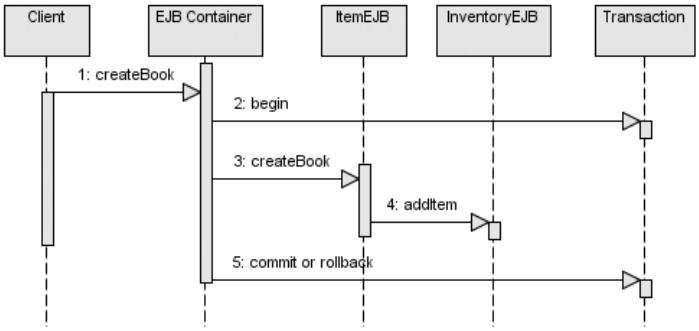


Figure 9-5. *The container handles the transaction.*

It's interesting to note in Listing 9-1 and Figure 9-5 that a business method of a bean (`ItemEJB.createBook()`) can be a client of a business method of another bean (`InventoryEJB.addItem()`).The default behavior is that whatever transaction context is used for `createBook()` (from the client or created by the container) is applied to `addItem()`. The final commit happens if both methods have returned successfully. This behavior can be changed using metadata (annotation or XML deployment descriptor). Depending on the transaction attribute you choose (`REQUIRED`, `REQUIRES_NEW`, `SUPPORTS`, `MANDATORY`, `NOT_SUPPORTED`, or `NEVER`), you can affect the way the container demarcates transactions: the container uses the client's transaction, runs the method in a new transaction, runs the method with no transaction, or throws an exception. Table 9-3 defines the transaction attributes.

Table 9-3. *CMT Attributes*

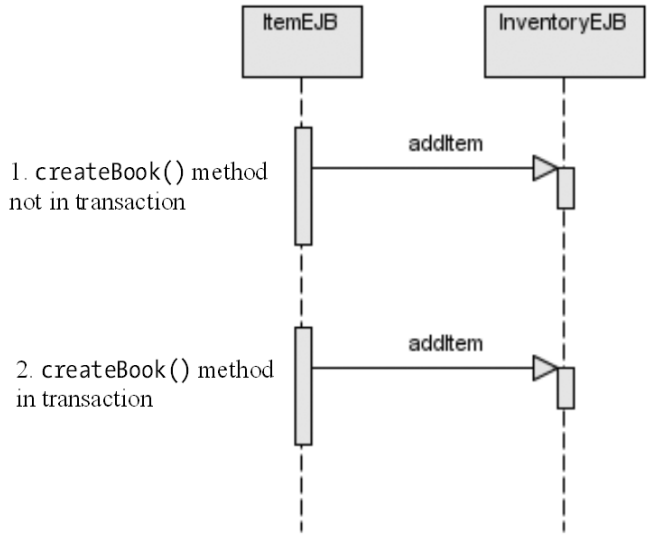
Attribute	Description
`REQUIRED`	This attribute, the default value, means that a method must always be invoked within a transaction. The container creates a new transaction if the method is invoked from a nontransactional client. If the client has a transaction context, the business method runs within the client's transaction. You should use `REQUIRED` if you modify any data and you don't know whether the client has started a transaction or not.
`REQUIRES_NEW`	The container always creates a new transaction before executing a method, regardless of whether the client is executed within a transaction. If the client is running within a transaction, the container suspends that transaction temporarily, creates a second one, commits it, and then resumes the first transaction. This means that the success or failure of the second transaction has no effect on the existing client transaction. You should use `REQUIRES_NEW` when you don't want a rollback to affect the client.

Continued

Table 9-3. *Continued*

Attribute	Description
SUPPORTS	The EJB method inherits the client's transaction context. If a transaction context is available, it is used by the method; if not, the container invokes the method with no transaction context. You should use SUPPORTS when you have read-only access to the database table.
MANDATORY	The container requires a transaction before invoking the business method but should not create a new one. If the client has a transaction context, it is propagated; if not, a javax.ejb.EJBTransactionRequiredException is thrown.
NOT_SUPPORTED	The EJB method cannot be invoked in a transaction context. If the client has no transaction context, nothing happens; if it does, the container suspends the client's transaction, invokes the method, and then resumes the transaction when the method returns.
NEVER	The EJB method must not be invoked from a transactional client. If the client is running within a transaction context, the container throws a javax.ejb.EJBException.

Figure 9-6 illustrates all the possible behaviors that an EJB can have depending on the presence or not of a client's transaction context. For example, if the createBook() method doesn't have a transaction context and invokes addItem() with a MANDATORY attribute, an exception is thrown. The bottom part of Figure 9-6 shows the same combinations but with a client that has a transaction context.



CMT Attribute	What happens?
REQUIRED	New transaction
REQUIRES_NEW	New transaction
SUPPORTS	No transaction
MANDATORY	Exception
NOT_SUPPORTED	No transaction
NEVER	No transaction

CMT Attribute	What happens?
REQUIRED	Caller's transaction
REQUIRES_NEW	New transaction
SUPPORTS	Caller's transaction
MANDATORY	Caller's transaction
NOT_SUPPORTED	No transaction
NEVER	Exception

Figure 9-6. *Two calls made to InventoryEJB with different transaction policies*

To apply one of these six demarcation attributes to your session bean, you have to use the @javax.ejb.TransactionAttribute annotation or the deployment descriptor (setting the <trans-attribute> element in the ejb-jar.xml). This metadata can be applied either to individual methods or to the entire bean. If applied at the bean level, all business methods will inherit the bean's transaction attribute value. Listing 9-2 shows how the ItemEJB uses a

SUPPORT transaction demarcation policy and overrides the createBook() method with REQUIRED.

Listing 9-2. *A Stateless Bean with CMT*

```
@Stateless
@TransactionAttribute(TransactionAttributeType.SUPPORTS)
public class ItemEJB {

    @PersistenceContext(unitName = "chapter09PU")
    private EntityManager em;
    @EJB
    private InventoryEJB inventory;

    public List<Book> findBooks() {
        Query query = em.createNamedQuery("findAllBooks");
        return query.getResultList();
    }

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public Book createBook(Book book) {
        em.persist(book);
        inventory.addItem(book);
        return book;
    }
}
```

■ **Note** Client transaction context does not propagate with asynchronous method invocation. MDBs support only the REQUIRED and NOT_SUPPORTED attributes as explained in Chapter 13.

Marking a CMT for Rollback

You've seen that the EJB container demarcates transactions automatically and invokes begin, commit, and rollback operations on your behalf. But as a developer, you might want to prevent the transaction from being committed if some error or business condition is encountered. It is important to stress that a CMT bean is not allowed to roll back the transaction explicitly. Instead, you need to use the EJB context (see the "Session Context" section in Chapter 7) to inform the container to roll back.

As you can see in Listing 9-3, the InventoryEJB has a oneItemSold() method that accesses the database through the persistence manager, and sends a JMS message to inform the shipping company that an item has been sold and should be delivered. If the inventory level is equal to zero (which means no more items are available), the method needs to explicitly roll back the transaction. To do so, the stateless bean first needs to obtain the

`SessionContext` through dependency injection and then call its `setRollbackOnly()` method. Calling this method doesn't roll back the transaction immediately; instead a flag is set for the container to do the actual rollback when it is time to end the transaction.

Listing 9-3. *A Stateless Bean Marks the Transaction for Rollback*

```
@Stateless
public class InventoryEJB {

    @PersistenceContext(unitName = "chapter09PU")
    private EntityManager em;
    @Resource
    private SessionContext ctx;

    public void oneItemSold(Item item) {
        em.merge(item);
        item.decreaseAvailableStock();
        sendShippingMessage();

        if (inventoryLevel(item) == 0)
            ctx.setRollbackOnly();
    }
}
```

Similarly, a bean can call the `SessionContext.getRollbackOnly()` method, which returns a `boolean`, to determine whether the current transaction has been marked for rollback.

Another way to programmatically inform the container to roll back is through throwing specific types of exceptions.

Exception Handling

Exception handling in Java has been confusing since the creation of the language (as it involves both checked exceptions and unchecked exceptions). Associating transactions and exceptions in EJBs is also quite intricate. Before going any further, I just want to say that throwing an exception in a business method will not always mark the transaction for rollback. It depends on the type of exception or the metadata defining the exception. In fact, the EJB 3.1 specification outlines two types of exceptions:

- *Application exceptions*: Exceptions related to business logic handled by the EJB. For example, an application exception might be raised if invalid arguments are passed to a method, the inventory level is too low, or the credit card number is invalid. Throwing an application exception does not automatically result in marking the transaction for rollback. As detailed later in this section in Table 9-4, the container doesn't roll back when checked exceptions (which extend `java.lang.Exception`) are thrown, but it does for unchecked exceptions (which extend `RuntimeException`).

- *System exceptions*: Exceptions caused by system-level faults, such as JNDI errors, JVM errors, failure to acquire a database connection, and so on. A system exception must be a subclass of a `RuntimeException` or `java.rmi.RemoteException` (and therefore a subclass of `javax.ejb.EJBException`). Throwing a system exception results in marking the transaction for rollback.

With this definition, we know now that if the container detects a system exception, such as an `ArithmeticException`, `ClassCastException`, `IllegalArgumentException`, or `NullPointerException`, it will roll back the transaction. Application exceptions depend on various factors. As an example, let's change the code from Listing 9-3 and use an application exception as shown in Listing 9-4.

Listing 9-4. *A Stateless Bean Throwing an Application Exception*

```
@Stateless
public class InventoryEJB {

  @PersistenceContext(unitName = "chapter09PU")
  private EntityManager em;

  public void oneItemSold(Item item) throws InventoryLevelTooLowException{
    em.merge(item);
    item.decreaseAvailableStock();
    sendShippingMessage();

    if (inventoryLevel(item) == 0)
      throw new InventoryLevelTooLowException();
  }
}
```

`InventoryLevelTooLowException` is an application exception because it's related to the business logic of the `oneItemSold()` method. Depending on whether you want to roll back the transaction or not, you can either make it extend from a checked or an unchecked exception or annotate it with `@javax.ejb.ApplicationException` (or the XML equivalent in the deployment descriptor). This annotation has a `rollback` element that can be set to `true` to explicitly roll back the transaction. Listing 9-5 shows the `InventoryLevelTooLowException` as an annotated checked exception.

Listing 9-5. *An Application Exception with* `rollback = true`

```
@ApplicationException(rollback = true)
public class InventoryLevelTooLowException extends Exception {

    public InventoryLevelTooLowException() {
    }

    public InventoryLevelTooLowException(String message) {
        super(message);
    }
}
```

If the `InventoryEJB` in Listing 9-4 throws the exception defined in Listing 9-5, it will mark the transaction for rollback, and the container will do the actual rollback when it is time to end the transaction. That's because the `InventoryLevelTooLowException` is annotated with `@ApplicationException(rollback = true)`. Table 9-4 shows all the possible combinations with application exceptions. The first line of the table could be interpreted as "If the application exception extends from `Exception` and has no `@ApplicationException` annotation, throwing it will not mark the transaction for rollback."

Table 9-4. *Combination of Application Exceptions*

Extends from	@ApplicationException	Description
`Exception`	No annotation	By default, throwing a checked exception doesn't mark the transaction for rollback.
`Exception`	`rollback = true`	The transaction is marked for rollback.
`Exception`	`rollback = false`	The transaction is not marked for rollback.
`RuntimeException`	No annotation	By default, throwing an unchecked exception marks the transaction for rollback.
`RuntimeException`	`rollback = true`	The transaction is marked for rollback.
`RuntimeException`	`rollback = false`	The transaction is not marked for rollback.

Bean-Managed Transactions

With CMT, you leave the container to do the transaction demarcation just by specifying a transaction attribute and using the session context or exceptions to mark a transaction for rollback. In some cases, the declarative CMT may not provide the demarcation granularity that you require (for example, a method cannot generate more than one transaction). To address this, EJBs offer a programmatic way to manage transaction demarcations with BMT. BMT allows you to explicitly manage transaction boundaries (begin, commit, rollback) using JTA.

To turn off the default CMT demarcation and switch to BMT mode, a bean simply has to use the `@javax.ejb.TransactionManagement` annotation (or the XML equivalent in the `ejb-jar.xml` file) as follows:

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class ItemEJB {
    ...
}
```

With BMT demarcation, the application requests the transaction, and the EJB container creates the physical transaction and takes care of a few low-level details. Also, it does not propagate transactions from one BMT to another.

The main interface used to carry out BMT is `javax.transaction.UserTransaction`. It allows the bean to demarcate a transaction, get its status, set a timeout, and so on. The `UserTransaction` is instantiated by the EJB container and made available through dependency injection, JNDI lookup, or the `SessionContext` (with the `SessionContext.getUserTransaction()` method). The API is described in Table 9-5.