10

L'API JPA et la gestion des entités

Nous avons exploré au chapitre précédent les fondamentaux de l'architecture EJB3 ainsi que le modèle de développement des beans session supporté par la nouvelle spécification.

Nous présentons dans ce chapitre le modèle entité introduit par la spécification EJB 3.0, qui fait l'objet d'un document spécifique, preuve du soin apporté à ce sujet sensible entre tous qu'est la persistance des données.

La nouvelle API JPA (Java Persistence API) offre de nombreuses améliorations, qui tendent vers une plus grande simplicité que les précédentes versions du modèle EJB. Comme les beans session, les beans entité deviennent de simples POJO, découplés de leur framework de persistance. Cela permet de les utiliser à l'intérieur comme à l'extérieur de leur conteneur.

JPA (Java Persistence API)

L'API de persistance Java JPA est une spécification de Sun. Fondée sur le concept POJO pour la persistance Java, elle est relativement récente puisque sortie en même temps que JEE5, en mai 2006.

Disponible depuis les premières versions du JDK 1.5, JPA est une norme, et non une implémentation, chaque fournisseur étant libre d'implémenter son propre framework de persistance tant qu'il respecte les spécifications JPA.

JPA permet de mapper les objets POJO avec les tables de la base. Il devient dès lors possible d'utiliser JPA pour stocker les objets Java codés sans avoir à les sous-classer ou à implémenter une interface spécifique, à l'inverse de la lourdeur imposée par EJB 2.x.

La persistance traite des aspects de stockage et de récupération des données applicatives. Elle peut maintenant être programmée avec l'API de persistance Java, devenue un standard dans la spécification EJB 3.0 (JSR-220). Elle est apparue en réponse au manque de

flexibilité et à la complexité de la spécification J2EE 1.4, en particulier en ce qui concerne la persistance des beans entités.

Rappelons que c'est poussé et inspiré par les frameworks Open Source Spring, avec son mécanisme d'injection de dépendances, et Hibernate, et sa gestion de la persistance, qu'une grande partie des mécanismes de persistance EJB3 amenés par JPA ont été construits.

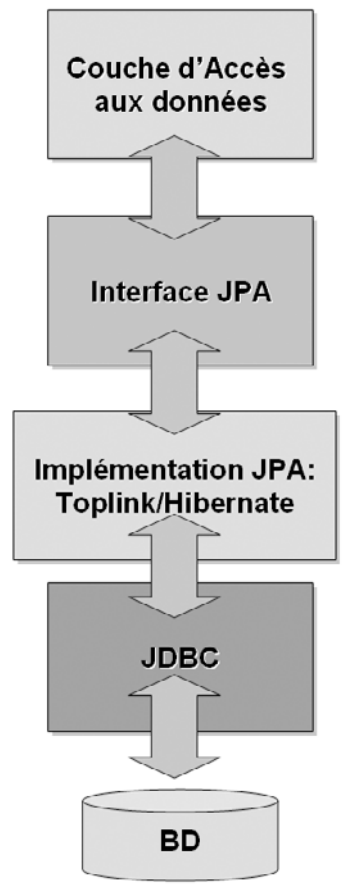
Un des grands atouts de l'API JPA est qu'elle est indépendante de tout fournisseur. Elle peut ainsi s'intégrer facilement à des serveurs d'applications JEE ou JSE (Tomcat).

JPA est implémentée par deux produits de référence : TopLink, un produit commercial (Oracle) devenu libre, et Hibernate.

L'architecture de JPA et son intégration dans l'architecture *n*-tiers sont illustrées à la figure 10.1. JDBC reste toujours la couche standard utilisée en Java pour l'accès aux données.

Figure 10.1

Architecture de JPA



Les aspects importants de cette nouvelle architecture sont ses relatives stabilité et standardisation. La couche d'accès aux données dialoguant avec les interfaces JPA, les développements gagnent en souplesse, puisqu'il n'est plus nécessaire de changer de modèle O/R ni de couche DAO (pour l'accès aux données) en fonction de l'outil de mapping utilisé. Quel que soit le produit qui implémente l'API, l'interface de la couche JPA reste inchangée.

Caractéristiques de JPA

Avec JPA, toute la complexité qui faisait frémir les développeurs d'applications Java appelés à développer des projets à base d'EJB est évacuée.

Ses principaux avantages sont les suivants :

- Disparition de la multitude d'interfaces (Home, Remote, Local, etc.).
- Possibilité d'utiliser JPA à l'intérieur comme à l'extérieur d'un conteneur JEE.
- Transformation des beans entité en simples POJO.
- Mapping O/R (objet-relationnel) avec les tables de la base facilitée par les annotations.

La figure 10.2 illustre le mécanisme permettant la transformation de toute classe régulière (exemple `Article`) en table correspondante, en utilisant les annotations.

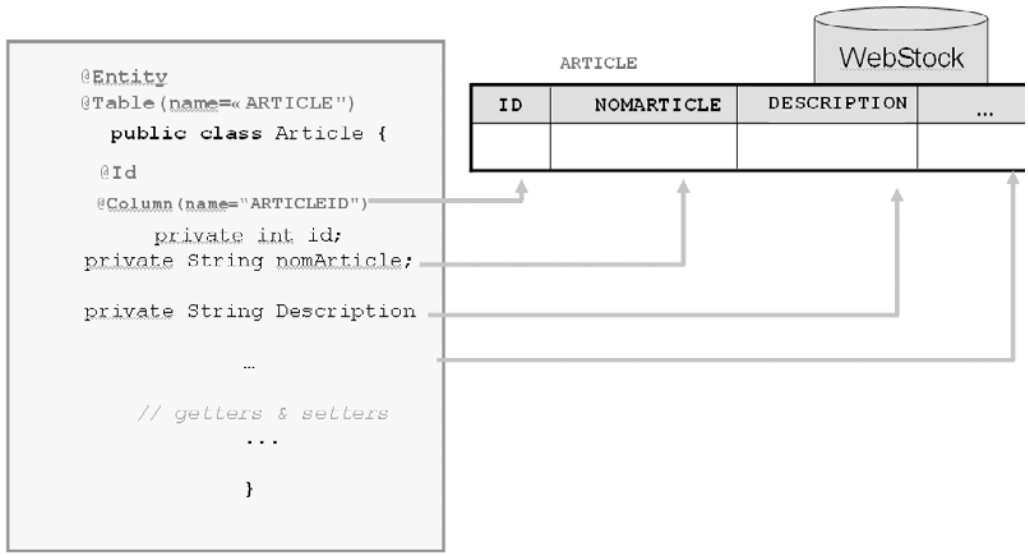


Figure 10.2

Mapping O/R avec JPA

En résumé, JPA fournit les services suivants :

- Mécanisme à la Hibernate permettant de définir déclarativement le mapping O/R et de mapper un objet à une ou plusieurs tables de la base de données grâce aux annotations de J2SE 5.0. Les annotations peuvent être utilisées pour définir des objets, des relations, du mapping O/R, de l'injection et de la persistance du contexte. JPA fournit en outre une option pour utiliser les descripteurs XML au lieu des annotations.
- API permettant de manipuler les beans entité pour effectuer les opérations CRUD de persistance, récupération et suppression des objets. Le développeur s'affranchit ainsi de toutes les tâches rébarbatives d'écriture du code de persistance des objets métier *via* JDBC et les requêtes SQL associées.

- Langage de requête standard pour la récupération des objets *(*JP QL, une extension de l'EJB QL d'EJB 2.x. C'est sans doute là un des aspects les plus important de la persistance des données, tant les requêtes SQL mal construites ralentissent la base de données. Cette approche affranchit les applications du langage de requête SQL propriétaire.

Les beans entité

Comme indiqué précédemment, avec EJB3 les entités deviennent des objets POJO ordinaires, expurgés de la tuyauterie si complexe d'EJB2. Ils représentent exactement le même concept que les entités de persistance Hibernate.

Dans la terminologie JPA, l'entité se réfère à un ensemble de données qui peuvent être stockées sur un support (base de données, fichier) et récupérées sous la forme d'un tout indissociable.

Une entité possède des caractéristiques spécifiques, comme la persistance ou l'identité (une entité est une instance forcément unique, identifiée grâce à sa clé) et gère les aspects liés à l'atomicité de la transaction.

Une classe entité doit respecter les prérequis suivants :

- Être annotée avec les annotations du package javax.persistance.Entity.
- Posséder un ou plusieurs constructeurs sans argument de type public ou protected.
- Ne pas être déclarée comme final. Aucune méthode ou variable d'instance de persistance ne doit être déclarée final.
- Hériter d'une autre entité ou d'une classe non-entité. Une classe non-entité peut également hériter d'une classe entité.

Les variables d'instance de persistance doivent être déclarées private ou protected et ne pas être accédées directement par les méthodes de la classe entité. Les clients doivent accéder à l'état de l'entité à travers l'invocation de méthodes accesseur.

En résumé, une entité EJB3 est un POJO ordinaire dont le mapping est défini par les annotations du JDK5 ainsi que par les annotations EJB3 et/ou tout framework JEE comme JBoss Seam par exemple.

Le mapping peut être de deux types :

- logique (associations de la classe, etc.) ;
- physique (décrivant le schéma physique, avec les tables, colonnes, index, etc.).

Annotations de persistance des beans entité

Comme nous l'avons vu, toute classe Java peut facilement être transformée en objet entité grâce aux annotations. Les spécifications EJB3 et JPA apportent un certain nombre d'annotations liées à la persistance des entités, comme le mapping à une table, les colonnes associées à des types de données simple, le mapping de clés primaires, le support de la génération automatique de l'identifiant de la clé ou la gestion des relations entre entités.

La figure 10.3 illustre les quatre entités principales du domaine webstock de notre étude de cas : `Article`, `Inventaire`, `Commande` et `Client`.

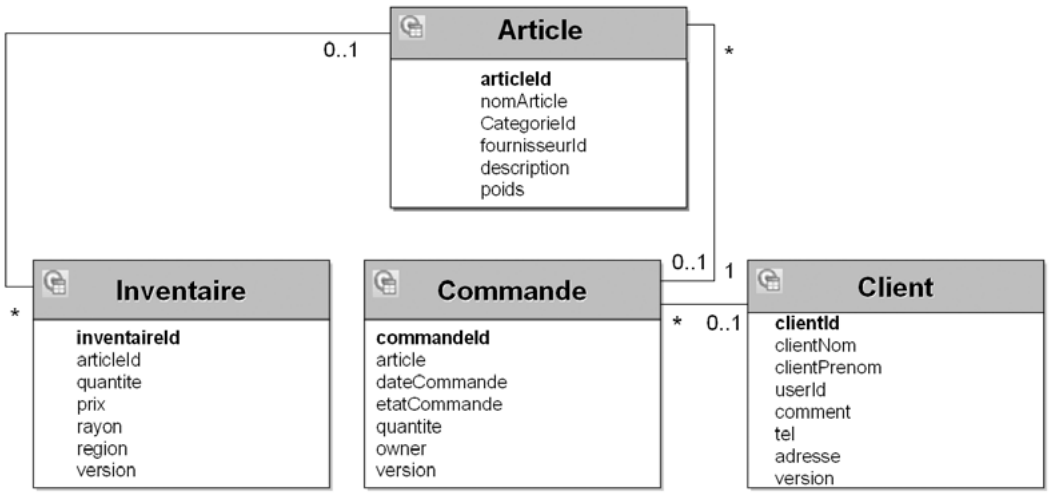


Figure 10.3

Sous-modèle objet des entités webstock

Les cardinalités des relations du modèle sont les suivantes :

- One-to-One (relation entre `Article` et `Inventaire`) : un article est relatif à une ligne d'un inventaire en magasin.
- One-to-One (relation entre `Article` et `Commande`) : chaque enregistrement d'inventaire contient un et un seul article.
- One-to-Many (relation entre `Client` et `Commande`) : spécifie qu'une entité est associée avec une collection d'autres entités. Dans notre exemple, un client peut passer plusieurs commandes (on représente d'ailleurs parfois l'extrémité de la relation, ici `Commande`, par le caractère « * », qui précise la multiplicité).
- Many-to-One (relation entre `Commande` et `Client`) : relation bidirectionnelle relativement utilisée pour spécifier que plusieurs commandes sont associées à un client.

Les cardinalités et les notions de directionnalité et de propriété de la relation (notion liée à la jointure du type de relation) sont essentielles dans la modélisation et le mapping O/. Nous y reviendrons lors du design de l'étude de cas autour de l'extrait du modèle métier webstock.

La marche à suivre pour utiliser une entité au sens JPA du terme est la suivante :

1. Création d'une classe représentant les données d'une ou plusieurs tables, soit un simple JavaBean possédant des setters et des getters, comme ici pour `Article` :

```
public classe Article {

int id ;
private String nomArticle ;
```

```
private String articleCategorieID;
private String fournisseurID;
private String description;
private long poids;


//constructeurs, getter et setter, etc …

public Article() {}
    public Article(int id) {
        this.id = id;
    }
public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
…

}
```

2. Ajout des métadonnées pour indiquer que la classe est une entité :

```
import javax.persistence.* ;

@Entity ❶

public class Article {

int id ;
private String nomArticle ;
private String articleCategorieID;
private String fournisseurID;
private String description;
private long poids;

//constructeurs, getter et setter, etc …

public Employe() {}
    public Employe(int id) {
        this.id = id;
    }
public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
…

}
```

3. Marquage de la classe avec l'annotation @Entity ❶ spécifiant au moteur de persistance que les objets créés avec cette classe peuvent utiliser le support de JPA pour les rendre persistants.

4. Ajout des métadonnées de mapping O/R sous forme d'annotation (ou de fichier XML au besoin) :

```
import javax.persistence.* ;

@Entity(name=?Article?) ❶

@Table (name= "Article", schema= "webstock") ❷

public class Article {

  @Id @GeneratedValue ❸

  @Column (name="ARTICLEID") ❹
  private int articleId;

  private String nomArticle ;
  private String articleCategorieID;
  private String fournisseurID;
  private String description;
  private long poids;

  public Article() {}
      public Article(int id) {
          this.id = id;
      }
  public int getId() {
          return id;
      }

      public void setId(int id) {
          this.id = id;
      }

// . . .

}
```

L'ajout de simples annotations à la classe `Article` permet de la rendre persistante.

Le tableau 10.1 récapitule l'ensemble des annotations standards appliquées aux classes entités.

Tableau 10.1 Annotations standards appliquées aux classes entités

Annotation	Description
@Entity ❶	Permet de qualifier la classe comme entité, en précisant au moteur de persistance que les instances d'objets créés à partir de cette classe peuvent utiliser le support de JPA pour être rendues persistantes. Toute entité est identifié par un nom (propriété name) et si non spécifié avec l'aide de la propriété name, prend le nom de la classe. Ce nom est utilisé par le langage JPQL (Java Persistence Query Language) pour référencer l'entité dans les requêtes. Remarque : si l'entité est passée par valeur lors de son transfert par le biais d'un bean session distant à partir de clients Java externes au conteneur EJB, le bean entité doit implémenter l'interface java.io.Serializable et utiliser le protocole RMI/IIOP à travers HTTP.
@Table ❷	Offre la possibilité de désigner la table sous-jacente à laquelle est associée cette classe, mais également de préciser le schéma de données associé.

Tableau 10.1 Annotations standards appliquées aux classes entités *(suite)*

Annotation	Description

@Id ❸	Indique au moteur de persistance quel champ de la classe (il peut en exister plusieurs) est la clé primaire ou l'identifiant de la classe. La valeur de l'identifiant du champ doit être unique par rapport à l'ensemble des instances d'entités. En complément de cette annotation, @GeneratedValue s'applique à la propriété clé primaire du champ en conjonction avec l'annotation @Id afin de permet une génération automatique de la clé selon le choix de la stratégie de génération adoptée : AUTO (par défaut), TABLE, SEQUENCE ou IDENTITY. Exemple :
	`@Entity`
	`public class Article {`
	`  @Id @GeneratedValue (strategy=GenerationType.AUTO)`
	`  private int id;`
	`}`
	Ce mode de création de l'identifiant génère automatiquement une valeur par le biais du fournisseur de persistance JPA sous-jacent (Hibernate ou TopLink) qui sera inséré dans le champs id de chaque entité Employé persistante. Précisons que le mode AUTO est à privilégier dans une phase de prototype ou de développement, les autres modes sont à privilégier en phase d'industrialisation et de production puisque s'appuyant sur le moteur de base de données sous-jacent.

@Column ❹	Permet d'indiquer que le nom de la colonne dans la base de données sera différent de celui de l'attribut défini dans la classe. Il est également possible de spécifier la taille de la colonne en nombre de caractères et de préciser si la colonne peut accepter les valeurs nulles. Exemple : `@Column(name = "PHONENUMERO", nullable = true , length=10)` Le framework de persistance fournit automatiquement la conversion des types de données des colonnes vers certains types de données Java correspondants. Le fournisseur de persistance génère un type de donnée associé à la colonne compatible lors de la génération des tables pour la classe entité lors du déploiement (les types numérique, chaîne et date sont automatiquement convertis).

@Version	Annote un attribut dont la valeur représentera un numéro de version (incrémenté à chaque modification) pour l'entité et sera utilisé pour savoir si l'état d'une entité a été modifiée entre deux moments différents (stratégie optimiste). Exemple :
	`@Entity`
	`public class Flight implements Serializable {`
	`...`
	`  @Version`
	`  @Column(name="OPTIONLOCK")`
	`  public Integer getVersion() { ... }`
	`}`

@Basic	Permet de déclarer la stratégie de récupération pour une propriété. Cette annotation influe sur les performances et la manière dont les données de l'entité sont accédées et fonctionne en combinaison avec le type fetch. Le mode FetchType permet de préciser au fournisseur de différer le chargement de l'état de l'attribut jusqu'à ce que celui-ci soit référencé. Exemple :
	`@Entity`
	`public class Article {`
	`  @Basic(fetch=FetchType.LAZY)`
	`  @Column(name= ?description?)`
	`  private String comments ;`
	`}`
	Ne pas avoir d'annotation pour une propriété est équivalent à l'annotation @Basic.

@Transient	Chaque propriété (champ ou méthode) non statique et non transiente d'un bean entité est considérée persistante, à moins que vous ne l'annotiez comme @Transient. Exemple :
	`@Transient`
	`String getLengthInMeter() { ... } // propriété transient`

Clés primaires composées

Si la clé primaire d'une entité mappe plusieurs colonnes de la table, cette clé primaire est dite composée. EJB3 propose deux annotations pour supporter cette fonctionnalité : @IdClass et @EmbeddedId.

L'annotation @IdClass

L'entité déclare chaque champ qui constitue la clé composée directement dans la classe de l'entité, annotant chacun de ceux-ci avec @Id et spécifiant la classe de la clé composée qui compose ces champs avec @IdClass, comme dans l'exemple suivant :

```
@Entity
@IdClass (ArticlePK.class)
public class Article {

@Id @GeneratedValue
@Column (name="ARTICLEID")
private int articleId;
@Id
private String nomArticle ;

public Integer getArticleId() { return articleId; }
public void setArticleId (Integer articleId) { this.articleId = articleId; }

public void setNomArticle (String nomArticle) { this.nomArticle = nomArticle; }
public String getNomArticle() { return nomArticle; }

// …
}
```

L'annotation @IdClass identifie un POJO ordinaire, c'est-à-dire sans annotations. Tous les mappings requis par les champs constituant la clé primaire sont spécifiés dans les champs de l'entité (les champs annotés par @Id dans l'entité doivent correspondre aux champs de la classe clé primaire composite) :

```
public class ArticlePK implements Serializable {
private integer articleId;
private String nomArticle

public void setArticleId (integer articleId) { this.articleid = articleId; }
public Integer getArticleId() { return articleId; }


public void setNomArticle (String nomArticle) { this.nomArticle=nomArticle; }
}
```

L'annotation @EmbeddedId

L'entité peut designer un ou plusieurs champs qui la constituent comme constituant de sa clé primaire en utilisant l'annotation @EmbeddedId en combinaison avec @Embeddable.

Toute annotation @EmbeddedId doit référencer une classe marquée @Embeddable, comme le montre l'extrait suivant d'une variante de l'entité `Article` :

```
@Entity
public class Article {
@EmbeddedId
private ArticlePK2 articleId ;
```

```
public Integer getArticleId() { return articleId ; }
public void setArticleid (Integer articleId)  { this.articleId = articleId; }

}

@Embeddable
public class ArticlePK2 {
  @Id
  @Column (name="ARTICLEID")
  private int articleId;
  String nomArticle.

  public void setArticleId (int articleId) { this.articleId =
  articleId; }
  public int getArticleId() { return articleId; }

  public void setNomArticle (String nomArticle) { this.nomArticle = nomArticle;}
 public String getNomArticle() { return nomArticle; }

}
```

Choisir @IdClass ou @EmbeddedId ?

@IdClass assure une compatibilité avec la spécification EJB2 en cas d'association de type M:N et si l'on souhaite que la clé primaire de la table association soit composée des clés étrangères vers les tables associées. Dans les autres cas, il est conseillé d'utiliser @EmbeddedId.

Relations entre beans entité

La plupart des beans entité disposent de relations entre eux ce qui produit les graphes de modèles de données si communs aux applications métier.

Dans les sections suivantes, vous allez explorer les différents types de relations qui peuvent exister entre beans entité et découvrir comment les définir et les faire correspondre avec le modèle de données sous-jacent en utilisant les annotations EJB3.

Avant de les aborder dans le détail, un bref rappel s'impose sur la notion de relation dans le mapping O/R.

Directionnalité de la relation

La relation d'une entité à une autre entité est définie par son attribut porté par la classe. Si l'autre entité pointée dispose d'un attribut qui pointe à son tour vers l'entité source, on parle de relation bidirectionnelle (entités `Client`/`Commande`). Si seulement une des entités de la relation possède une référence vers l'autre, la relation est dite unidirectionnelle.

Cardinalité

Un concept important dans la notion de relation entre les beans entité d'un modèle métier est la cardinalité, notion qui s'applique de manière identique au modèle relationnel et qui permet d'exprimer le nombre d'entités qui existent de part et d'autre de la relation liant les deux instances. Dans cette relation, chaque rôle possède sa propre cardinalité indiquant s'il existe une seule instance d'entité ou plusieurs.

Mapping des relations

Chacun des mappings avec la base de données est déterminé par la cardinalité de la source et le rôle de la cible.

Ce mapping s'exprime de la façon suivante :

- Many-To-One, ou plusieurs à un ;
- One-To-One, ou un à un ;
- One-to-Many, ou un à plusieurs ;
- Many-to-Many, ou plusieurs à plusieurs.

Ces noms de mappings sont aussi les noms des annotations utilisées pour indiquer le type de relation sur les attributs qui seront mappés, sachant que les mappings de relation peuvent être appliqués aux champs et aux propriétés des beans entité.

Une association entre une instance d'entité vers une autre instance d'entité dont la cardinalité de la cible est « un » est dite *monovaluée*. Les mappings de relation de type Many-to-One ou One-to-One entrent dans cette catégorie car l'entité source se réfère au plus à une entité cible.

Si l'entité source référence une ou plusieurs instances d'entité cible, l'association est dite *multivaluée*. Cela concerne les mappings One-to-Many et Many-to-Many.

Propriétés des relations et jointures

Dans une base de données, un mapping de relation entre deux tables signifie qu'une table possède une référence vers une autre table par le biais d'une colonne qui se réfère à une clé, habituellement la clé primaire de l'autre table. Cette colonne est appelée `clé étrangère`.

Transposé dans le modèle JPA, cette colonne est appelée colonne de jointure et est indiquée par l'annotation @JoinColumn.

Dans l'exemple de relation entre `Inventaire` et `Article`, la table `Inventaire` possède une clé étrangère articleId qui référence la table `Article`. Cette clé étrangère est la colonne de jointure qui associe les entités correspondantes `Article` et `Inventaire`.

Un autre concept important à prendre en compte dans la mise en œuvre d'une conception utilisant le mapping JPA est la celle « relation propriétaire », ou *owning side.* Dans une relation entre deux tables associées à deux entités, un des côtés de la relation dispose de la colonne de jointure dans la table. Ce côté est propriétaire de la relation. Le côté qui ne dispose pas de la colonne de jointure est appelé *non owning*, ou non propriétaire, ou encore côté inverse.

Les annotations qui définissent le mapping inter-entités avec les colonnes des tables de la base (en particulier @JoinColumn) sont toujours définies du côté de la relation *owning side*. Dans notre exemple de relation entre les entités `Article` et `Inventaire`, l'annotation @JoinColumn (name="articleId") signifie que la colonne articleId de l'entité source est la clé étrangère de l'entité cible, en l'occurrence `Article`. L'entité `Inventaire` est propriétaire de la relation (puisqu'elle détient la colonne de jointure).

Nom par défaut

Si aucune annotation @JoinColumn n'accompagne un mapping Many-to-One, un nom de colonne par défaut est supposé. Ce nom par défaut est formé de la combinaison des noms d'entités sources et du nom de la clé primaire de l'entité cible séparés par le caractère _.

Relations monovaluées

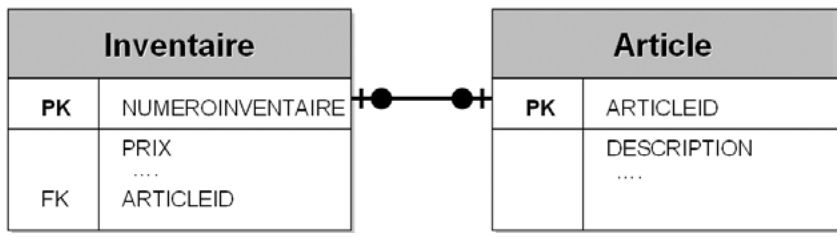
Mapping One-to-One

Ce type d'association implique que chaque entité est liée à une et une seule entité, et réciproquement.

Typiquement, une relation de ce type lie l'entité `Inventaire` à l'entité `Article` correspondante, comme dans les tables illustrées à la figure 10.4.

Figure 10.4

Relations One-to-One avec les tables Inventaire et Article



Ce type de mapping se décrit à l'aide de l'annotation @OneToOne. Le mapping One-to-One possédant une colonne de jointure dans la base nécessite de surcharger le nom de la colonne à l'aide de @JoinColumn lorsque le nom généré par défaut ne coïncide pas avec celui de la colonne de la table.

Dans l'extrait de code suivant, @JoinColumn est utilisée pour remplacer le nom de la colonne de jointure par défaut (soit ici `Article_userId` par `ARTICLEID`, clé étrangère de l'entité source `Inventaire`) :

```
Entity
@Table(schema="WEBSTOCK")
public class Inventaire {

@Id
@Column(table="Inventaire", name="inventaireId", insertable=false, updatable=false)
private long numInventaire;

@OneToOne
@JoinColumn(name="ARTICLEID", referencedColumnName="ARTICLEID")
protected Article article;

protected double prix;
protected int quantite;
protected String rayon;
protected String region;
protected int version;

// ….

}
```

La colonne de jointure `ARTICLEID` est déclarée avec @JoinColumn, qui possède le paramètre supplémentaire referencedColumnName. Ce paramètre sert à déclarer la colonne dans l'entité cible qui sera utilisée pour la jointure, soit `ARTICLEID`.

Mapping One-to-One bidirectionnel

Il arrive souvent qu'une entité cible d'une relation One-to-One possède une relation dite « reverse » vers l'entité source. Lorsque c'est le cas, nous appelons ce type de relation bidirectionnelle One-to-One. Elle est représentée en notation UML par une double flèche entre les deux entités.

Dans l'exemple de relation entre `Inventaire` et `Article`, le code suivant met en place ce type de relation bidirectionnelle entre les deux entités :

```
@Entity
@Table(schema="WEBSTOCK")
public class Article {

@Id
@Column(name="ARTICLEID")
@GeneratedValue
protected long articleId;

protected String nomArticle;

@oneToOne (mappedBy="article")
protected Inventaire inventaire;

//…

}
```

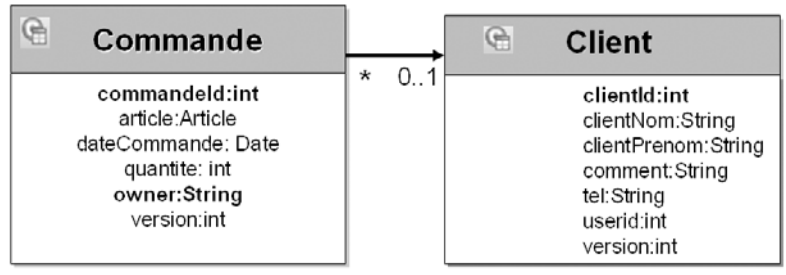
Nous devons dans ce cas ajouter la propriété mappedBy pour indiquer que le côté qui possède la relation est l'entité `Inventaire` et non `Article`. Nous devons préciser dans la valeur de l'attribut mappedBy le nom de l'attribut dans l'entité dite propriétaire de la relation et qui pointe à son tour sur l'entité `Article`, soit `article`. Comme l'entité `Article` n'est pas « propriétaire » de la relation, celle-ci n'a pas à fournir l'information de jointure et la colonne associée (pas de balise @JoinColumn).

Mapping Many-to-One

La figure 10.5 illustre le modèle objet correspondant à la relation de type Many-to-One unidirectionnelle entre les entités `Commande` et `Client`, relation illustrant le fait que plusieurs commandes peuvent être liées à un client. L'entité `Commande` est le côté « many » et la source de la relation, et l'entité `Client` le côté « one » et la cible.

Figure 10.5

Relation Many-to-One entre Commande et Client



La classe `Commande` possède un attribut appelé owner qui va contenir la référence à une instance de `Client`. Ce type de mapping Many-to-One est défini en annotant l'entité source (l'attribut qui se réfère à l'entité cible) avec l'annotation @ManyToOne.

Dans l'extrait de code suivant, l'annotation @ManyToOne est utilisée pour mapper cette relation. Le champ owner de l'entité `Commande` est l'attribut source qui sera annoté :

```
@Entity
@Table(schema="WEBSTOCK")
public class Commande {

@Id
@Column(name="commandeId")
@GeneratedValue
protected long numCommande;
@Temporal(DATE)
protected Date dateCommande;
protected Article article;
protected int quantite;
@Column(name="ETATCOMMANDE")
protected String etat;

@ManyToOne(optional=false)
@JoinColumn(name="CLIENTID", referencedColumnName = "clientId")
protected Client owner;

 // …
}
```

La colonne de jointure est déclarée avec @JoinColumn, qui ressemble à l'annotation @Column mais possède un paramètre de plus, nommé referencedColumnName. Ce paramètre déclare la colonne dans l'entité cible qui sera utilisée pour la jointure. La clé étrangère est nommée `CLIENTID` en lieu et place de la valeur par défaut qui serait utilisée si elle n'était pas spécifiée par le paramètre name de l'annotation @JoinColumn, soit ici `CLIENT_CLIENTID` (formé de la concaténation de l'entité cible et de sa clé primaire).

Relations multivaluées

Lorsque l'entité source référence une collection d'instances de l'entité cible, une association multivaluée est utilisée. Ce type d'association concerne les mappings One-to-Many et Many-to-Many.

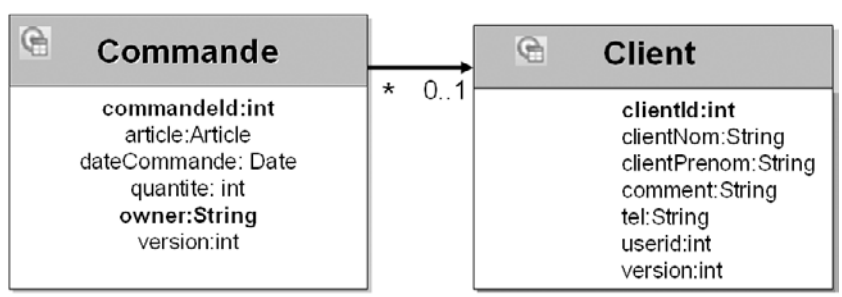
Mapping One-to-Many

Lorsqu'une entité est associée à une collection d'entités, ce type de mapping est de type One-to-Many. (type de cardinalité 1 à plusieurs ou 1..*).

La figure 10.6 illustre ce type de relation entre `Commande` et `Client`.

Figure 10.6

Relation One-to-Many entre Commande et Client



Dans cette portion du modèle, la clé étrangère CLIENTID de la table COMMANDE qui référence la table CLIENT correspond dans le jargon JPA à une entité Commande et à une colonne de jointure caractérisée par l'emploi de l'annotation @JoinColumn.

Dans ce type de relation, l'entité Commande « possède » la colonne, et l'entité Client dispose d'une relation One-to-Many utilisant l'annotation correspondante @OneToMany, comme dans l'extrait suivant :

```
public class Client {

@Id
@Column(table="Client")
@GeneratedValue
protected int clientId;

protected int userid;
protected String clientNom;
protected String clientPrenom;
protected String adresse;
@Column(name="TEL")
protected String telNumero;
protected String comment;
@Version
protected int version ;

@OneToMany
protected Collection <Commande> commandes;

// ….
}
```

Notez l'usage du type générique paramétré Collection pour stocker les entités de type Commande, garantissant l'existence de ce seul type dans la liste et éliminant l'usage des opérations de cast.

Mapping One-to-Many bidirectionnel

Une relation One-to-Many est souvent bidirectionnelle par nature et implique un mapping Many-to-One inverse vers l'entité source. Dans l'exemple précédent, il existe un mapping One-to-Many de l'entité Client vers Commande et un mapping retour Many-to-One de l'entité Commande vers Client.

Le code Many-to-One examiné plus haut reste inchangé dans l'exemple de relation bidirectionnelle puisque l'entité Commande « possède » la colonne de jointure CLIENTID et est propriétaire de la relation. Par contre, du côté opposé de la relation et de l'entité Client, il faut mapper la collection de commande de l'entité Commande comme relation de type One-to-Many en utilisant l'annotation @OneToMany, mais enrichie du paramètre name.

Le code correspondant à ce type de relation One-to-Many bidirectionnelle est le suivant :

```
public class Client {

@Id
@Column(table="Client")
@GeneratedValue
protected int clientId;
```

```
protected int userid;
protected String clientNom;
protected String clientPrenom;
protected String adresse;
@Column(name="TEL")
protected String telNumero;
protected String comment;
@Version
protected int version ;

@OneToMany (mappedBy="owner")
protected Collection <Commande> commandes;

// ….
}
```

En résumé, dans une relation One-to-Many bidirectionnelle :

- Le côté Many-to-One est propriétaire de la relation *(owning side)*. Par conséquent, la relation de jointure est définie sur ce côté.
- Le mapping One-to-Many est le côté inverse de la relation. L'élément mappedBy doit donc être utilisé en combinaison avec l'annotation @OneToMany. Si ce paramètre n'est pas mentionné, le fournisseur JPA la traite comme une relation unidirectionnelle de type One-to-Many utilisant une table de jointure.

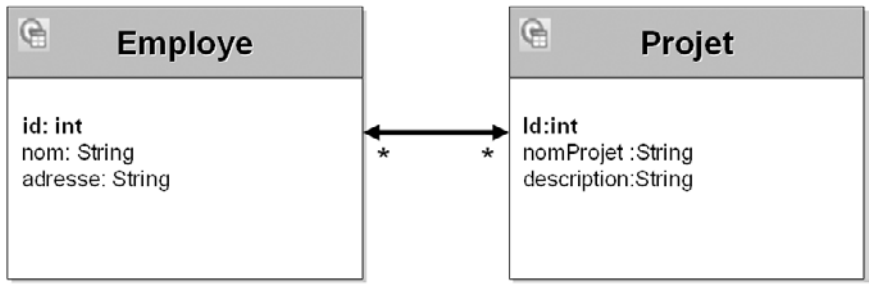
Mapping Many-to-Many

Le mapping Many-to-Many représente un type de relation entre une collection d'objets sources et une collection d'objets cibles. Ce type de mapping, moins usité que les précédentes relations, nécessite la création d'une table intermédiaire pour la gestion des associations entre les enregistrements sources et cibles.

La figure 10.7 illustre un exemple d'application de ce type de relation entre les entités `Employe` et `Projet`. Chaque employé peut travailler sur un ou plusieurs projets, et chaque projet peut à son tour être mis en œuvre par un ou plusieurs employés.

Figure 10.7

Mapping Many-to-Many bidirectionnel



Les mappings Many-to-Many utilisent une table contenant les colonnes des clés primaires pour les tables sources et destination.

Les clés primaires composites nécessitent une colonne pour chaque champ qui constitue la clé composite. Cette table doit exister avant toute utilisation de ce type de relation. La relation est aussi exprimée par l'annotation @ManyToMany pour chaque

attribut défini de part et d'autre de la relation, attribut de type collection, comme le montre l'extrait suivant :

```
@Entity
public class Employe {
    @Id private int id ;
     private String nom;
    private String adresse;
@ManyToMany
    Private Collection <Projet> projets;
//….
}

@Entity
public class Projet {
    @Id private int id ;
    private String nom;
    @ManyToMany (mappedBy="projets")
    private Collection <Employe> employes;
    //…
}
```

La principale différence avec les autres types de mappings, en particulier One-to-Many, est le fait qu'il n'existe pas de colonne de jointure de part et d'autre de la relation. De ce fait, la seule manière de mettre en œuvre ce type de mapping est la création d'une table de jointure séparée. La notion de propriétaire de la relation, notion fondamentale dans ce type de relation bidirectionnelle, doit donc être adaptée à ce contexte.

Il est nécessaire de choisir une des deux entités faisant partie de la relation comme propriétaire de la relation bidirectionnelle, l'autre devant être marquée comme « inverse ».

Dans notre exemple, nous avons choisi l'entité `Employe` mais aurions tout aussi bien pu choisir `Projet`. Comme toute relation bidirectionnelle, la relation inverse doit utiliser l'élément mappedBy pour identifier l'attribut propriétaire.

Utilisation des tables de jointure

Une table de jointure contient simplement deux clés étrangères, ou colonnes de jointure, qui permettent de référencer chacune des entités dans la relation, comme le montre la figure 10.8, qui permet de mettre en œuvre le mapping Many-to-Many.

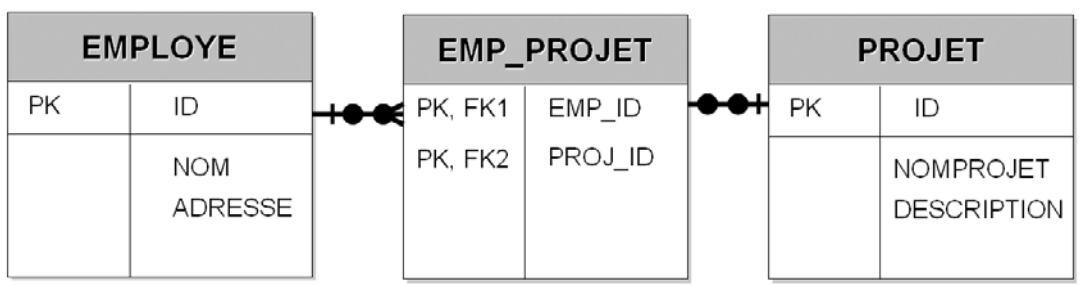


Figure 10.8

Table de jointure pour une relation Many-to-Many

Les tables EMPLOYE et PROJET ainsi que la table de jointure EMP_PROJET qui associe les entités Employe et Projet contiennent seulement les colonnes clés étrangères qui constituent la clé primaire composite. La colonne EMP_ID se réfère à la clé primaire de la table EMPLOYE, PROJ_ID, laquelle se réfère à la clé primaire de la table PROJET.

Afin de pouvoir mapper les tables ainsi décrites, il est nécessaire d'ajouter des informations de types métadonnées pour la classe Employe qui a été désignée dans notre exemple comme propriétaire de la relation.

L'extrait suivant décrit la relation Many-to-Many définie à l'aide des annotations de jointure :

```
@Entity
public class Employe {
     @Id private int id ;
      private String nom
      private String adresse;
      @ManyToMany
      @JoinTable (name="EMP_PROJET",
                       joinColumns=@JoinColumn(name="EMP_ID"),
      inverseJoinColumns=@JoinColumn(name="PROJ_ID"))
      private Collection <Projet> projets;
     // …..
}
```

L'annotation @JoinTable est utilisée pour configurer la table de jointure pour les relations entre les entités Employe et Projet. Cette annotation définit un name, un tableau de colonnes de jointure et un tableau de colonnes de jointure inverse. Ces dernières sont les colonnes de la table d'association qui référencent la clé primaire d'Employe (l'autre extrémité de la relation).

L'interface Entity Manager

Pour assurer la persistance de l'entité dans la base, il est nécessaire d'invoquer une interface JPA spécifique, appelée EntityManager, que nous appelons dans la suite gestionnaire d'entités. Cette interface correspond à l'état d'une connexion avec la base de données et permet de gérer l'ensemble des opérations touchant aux objets @Entity du contexte de persistance.

L'ensemble des instances d'entités qui sont cachées et gérées par le gestionnaire d'entités est appelé contexte de persistance, notion fondamentale pour la compréhension du gestionnaire d'entités JPA.

Les instances d'entités requêtées à travers le gestionnaire d'entités EntityManager peuvent être invoquées à partir du client au sein (JEE) et en dehors (JSE) du conteneur EJB3, les instances pouvant être ensuite mises à jour comme des objets Java ordinaires. Pour appliquer les modifications vers la base, le client invoque la méthode merge() sur l'EntityManager au sein du contexte transactionnel. Les données de l'entité sont ensuite sauvegardées dans la base.

La figure 10.9 illustre l'environnement d'exécution d'un client JPA mettant à jours ses entités en base à l'aide d'un gestionnaire d'entités spécifique.

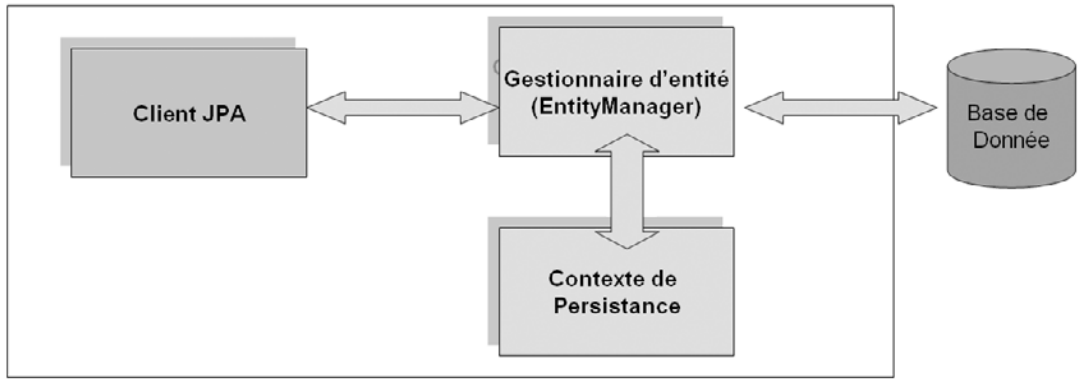


Figure 10.9

Architecture et contexte de persistance d'une application avec JPA

Types de gestionnaire d'entités

Il existe deux types de gestionnaires d'entités :

- Un gestionnaire d'entités géré par le conteneur du serveur d'applications, le contexte de persistance étant limité à une seule transaction.
- Un gestionnaire d'entités géré par l'application, c'est-à-dire en dehors du conteneur du serveur d'applications. Le contexte de persistance reste attaché au gestionnaire d'entités durant toute son existence. C'est l'application qui décide de la durée de vie du gestionnaire d'entités.

Nous aurons l'occasion en fin de chapitre de voir l'incidence sur le code de l'EntityManager, en particulier en termes d'invocation et d'utilisation à partir d'un client Java ou d'une application Web.

EntityManager et beans BMP/CMP

Les concepts qui recouvrent la notion de gestionnaire d'entités ressemblent à ceux concepts spécifiés par EJB2, à savoir les beans entité dont la persistance est gérée par le bean (BMP) et les beans dont la persistance est gérée par le conteneur (CMP). Meur mise en œuvre est cependant simplifiée, puisque cette persistance est simplement gérée par l'API JPA, sans nécessité d'interface particulière.

Méthodes de cycle de vie d'une entité EJB3

Une instance d'entité passe par plusieurs états durant sa durée d'exécution en tant qu'objet Java. Les classes d'entité EJB3 sont créées par des constructeurs simples, en lieu et place des « lourdes » interfaces fabriques familières des développeurs EJB2 Home et LocalHome. Elles peuvent ainsi être facilement transférées d'un conteneur EJB vers le client et mises à jour par ce dernier sans crainte d'une surcharge due à l'invocation d'une méthode de callback imposée par l'ancienne norme EJB.

Le tableau 10.2 récapitule les opérations prises en charge par le gestionnaire d'entités sur ses entités gérées.

Tableau 10.2 Opérations prises en charge par le gestionnaire d'entités

Opération	Description
persist()	Insère l'état d'une entité dans la base de données. Cette nouvelle entité devient alors une entité gérée.
remove()	Supprime l'état de l'entité gérée et ses données correspondantes de la base.
refresh()	Synchronise l'état de l'entité à partir de la base, les données de la BD étant copiées dans l'entité.
merge()	Synchronise les états des entités « détachées » avec le PC. La méthode retourne une entité gérée qui a la même identité dans la base que l'entité passée en paramètre, bien que ce ne soit pas le même objet.
find()	Exécute une requête simple de recherche de clé.
CreateQuery()	Crée une instance de requête en utilisant le langage JPQL.
createNamedQuery()	Crée une instance de requête spécifique.
createNativeQuery()	Crée une instance de requête SQL.
contains()	Spécifie si l'entité est managée par le PC.
flush()	Toutes les modifications effectuées sur les entités du contexte de persistance gérées par le gestionnaire d'entités sont enregistrées dans la BD lors d'un flush du gestionnaire.

Les états associés au cycle de vie d'une instance d'entité sont illustrées à la figure 10.10.

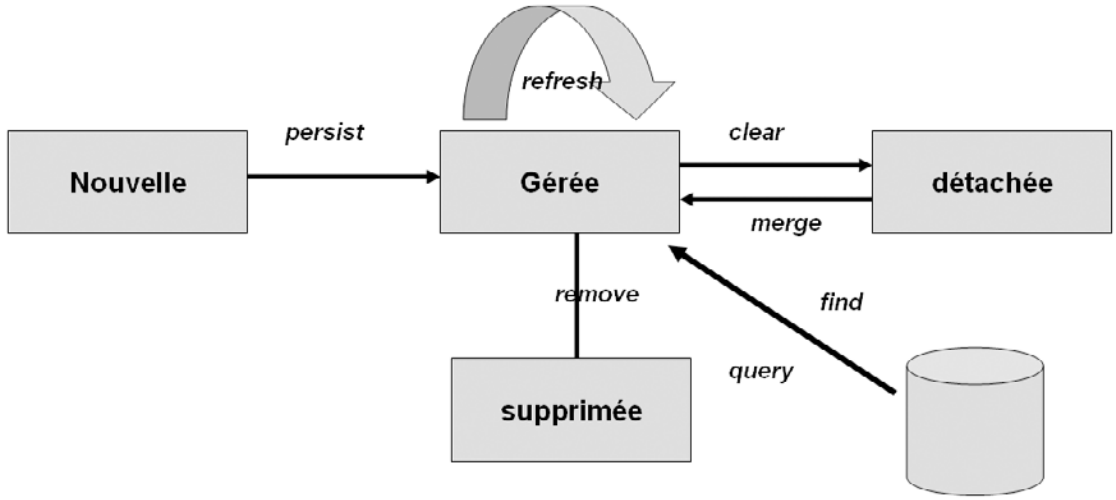


Figure 10.10

États d'une instance d'entité avec JPA

Pour illustrer l'emploi de ces méthodes, vous utiliserez par la suite l'entité `Employe` suivante (en supposant que l'ensemble des opérations s'exécute en dehors du conteneur JEE, c'est-à-dire sous le contrôle d'une JVM 5 et d'un conteneur JEE) :

```
@Entity
@Table (schema="WEBSTOCK", name="EMPLOYE")
public class Employe {
```

```
    @Id
    @Column(name="EMPLOYEID")
    private employeid;

    private String nom;
    private String adresse;
    private long salaire;
    public Employe() {}
    public Employe(int id) { this.id = id; }
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getNom() { return name; }
    public void setNom(String name) { this.name = name; }
    public String getAdresse() { return adresse; }
    public void setAdresse (String adrese) {this.adresse= adresse; }
    public long getSalaire() { return salaire; }
    public void setSalaire (long salaire) {this.salaire = salaire;}

    public String toString() {
        return "Identifiant Employe: " + getEmployeId() + " nom: " + getNom()
        ➥+ " Adresse: "+ getAdresse() +  " Salaire: " + getSalaire();
    }
}
```

Les étapes de l'utilisation de cette interface *via* un client JPA sont décrites succinctement ci-après.

Obtention d'une fabrique EntityManagerFactory

Une instance de l'objet EntityManager est toujours obtenue à partir d'une interface EntityManagerFactory. Celle-ci permet la création d'un contexte de persistance :

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("EmployeService");
```

`EmployeService` est le nom de la persistance-unit, ou unité de persistance, définie dans le fichier persistence.xml.

Cette fabrique est liée à une unité de persistance précise. On se rappelle que le fichier de configuration METAINF/persistence.xml permet de définir des unités de persistance et la source de données associée.

Ce fichier imposé par la spécification JPA est créé automatiquement par la facet JPA suivante :

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
    <persistence-unit name="EmployeeService" transaction-type="RESOURCE_LOCAL">
        <class>com.webstock.chap10.employe</class>
        <properties>
            <property name="toplink.jdbc.driver"
             value="org.apache.derby.jdbc.ClientDriver"/>
             <property name="toplink.jdbc.url"
    value="jdbc:derby://localhost:1527/EmpServDB;create=true"/>
            <property name="toplink.jdbc.user" value="APP"/>
            <property name="toplink.jdbc.password" value="APP"/>
        </properties>
    </persistence-unit>
</persistence>
```

La propriété transaction-type indique que vous fonctionnerez dans un mode non-JTA. Les transactions longues et le commit à deux phases ne seront donc pas pris en charge, ce qui n'est pas un souci dans cet exemple didactique. L'élément `class` spécifie la liste des entités qui font partie de l'unité de persistante, ici la simple classe `employe`. Si vous en possédiez plusieurs, chaque entité serait référencée entre chaque balise <class>.

TopLink

TopLink, implémentation de référence de JPA, comporte des extensions de la norme JPA telles que la génération automatique de table (propriété toplink.ddl-generation), associée à trois valeurs possibles : none, create-tables et drop-and-create-tables. Nous ne saurions trop vous conseiller de vous référer à la documentation du fournisseur pour en connaître les caractéristiques précises.

Contextes de persistance

Un contexte de persistance ne peut appartenir qu'à une et une seule unité de persistance associée à un ensemble de classes entités. À l'inverse, une unité de persistance peut contenir plusieurs contextes de persistance. Il faut donc veiller à ce qu'une entité n'appartienne qu'à un seul contexte de persistance pour éviter toute incohérence dans la base. Le gestionnaire d'entités est supprimé avec la méthode `close()` de la classe EntityManager.

Création d'un EntityManager

La ligne suivante permet de créer un gestionnaire d'entités à partir d'une fabrique créée précédemment :

```
EntityManager em = emf.createEntityManager() ;
```

Avec ce gestionnaire d'entités, il est ensuite possible de travailler sur les entités persistantes du modèle logique associé.

Principales opérations disponibles sur les entités

Cette section décrit les principales opérations disponibles pour gérer les entités, en supposant l'initialisation des actions d'obtention d'une fabrique et du gestionnaire d'entités associé.

Persistance d'une entité

Cette opération consiste à prendre une entité *transiente*, c'est-à-dire dont l'état ne peut être sérialisée ou qui ne possède pas de représentation persistante dans la base de données, et à stocker son état afin qu'elle puisse être récupérée ensuite.

Voici le code correspondant à cette opération :

```
Employe emp = new Employe (10) ;
em.persist(emp);
```

Si le gestionnaire d'entités rencontre un problème en appelant la méthode `persist()`, il génère une exception PersistenceException. Autrement, l'employé est stocké dans la base de données.

Lorsque l'appel de la méthode `persist()` est effectuée, l'instance emp renvoyée est un bean entité managé, c'est-à-dire pris en charge au sein du contexte de persistance du gestionnaire de déploiement.

L'extrait suivant illustre l'intégration de la méthode `persist()` dans la méthode `createEmploye()` de création d'un nouvel employé :

```
public Employe createEmploye(int employeid, String employenom,
String adresse, long salaire) {
   Employe article = new Article(employeid);
   emp.setEmployenom (employenom);
   emp.setAdresse (adresse);
   emp.salaire (salaire) ;
   em.persist(emp) ;
   return emp ;
}
```

Le code suivant illustre l'utilisation de la méthode `persist()` appliquée au contexte JEE avec l'exemple d'un bean session acquérant une instance d'EntityManager à travers l'injection d'un contexte automatisé (@PersistenceContext) attaché à l'unité de persistance définie précédemment (`EmployeService`) :

```
@Stateless
public class EmployeManager {
@PersistenceContext("EmployeeService ")
private EntityManager em;
public void createEmploye() {
 final Employe emp = new Employe();
 emp.setNom("John Doe");
 emp.setAdresse("Main street");
 em.persist(emp);
}
}
```

Méthode *persit()*

L'appel à la méthode `persist()` ne garantit pas qu'une instruction SQL `insert` soit effectuée immédiatement. Cette décision relève du gestionnaire d'entités, lequel peut décider de le faire dans la foulée ou plus tard, avant de commiter définitivement la transaction. Si les données de la base ont été modifiées (et validées) en parallèle dans la base, les données récupérées ne tiennent pas compte de ces modifications, ce qui peut provoquer des incohérences dans la base.

Recherche d'entités et requêtes JPQL

Lorsqu'une entité persiste dans la base, il est souvent nécessaire, pour les besoins de l'application, d'en récupérer les informations.

Le code correspondant à la recherche d'un employé avec un numéro donné est déconcertant de simplicité :

```
public Employe findEmploye (int employeId) {
    return em.find (Employe.class, 10);
]
```

Pour récupérer une instance d'`Employe`, les seules informations nécessaires à faire passer à la méthode `find()` sont le nom de la classe de l'entité ainsi que l'identifiant de clé primaire associée.

La méthode `find()` ne présente pas la souplesse ni la richesse d'un requête SQL, loin s'en faut. Heureusement, JPA fournit une API Query supportant l'interrogation de la base de données. Les interrogations s'effectuent à travers les entités du modèle métier et utilisent la syntaxe JPQL (Java Persistence Query Language).

L'API Query est essentielle à la jonction entre l'application et l'exécution des requêtes EJB-QL. Les méthodes de celles-ci sont regroupées dans l'interface javax.persistence.Query. L'intérêt principal de cette API est qu'elle permet de créer des requêtes dynamiques sous la forme de simples chaînes de caractères, et non de manière statique, comme c'est le cas au sein d'un descripteur de déploiement ou avec les annotations.

Une requête dynamique est une requête dont les clauses sont fournies à l'exécution, à l'inverse des requêtes nommées, abordées au chapitre suivant, où l'utilisateur doit fournir les critères de la requête ainsi que son nom avant toute utilisation.

Les principales méthodes de cette API sont regroupées dans l'interface EntityManager :

```
public interface EntityManager {
   public Query createQuery(String requeteEJB-QL);
   public Query createNamedQuery(String requeteNommee);
   public Query createNativeQuery(String requeteSQL);
   public Query createNativeQuery(String requeteSQL, Class résultat);
   public Query createNativeQuery(String requeteSQL, String résultat);

}
```

Vous vous intéresserez ici à la première méthode, `CreateQuery()`, qui permet de créer de requêtes dynamiques. Ces requêtes sont moins performantes que les requêtes nommées, car elles ne sont pas précompilées avant leur exécution par le moteur JPA.

L'exemple simple suivant montre comment l'objet Query est créé avec l'interface EntityManager et comment la chaîne JPQL est transmise avec les arguments de la requête exécutée pour avoir la liste des employés de la base :

```
Query query = em.createQuery ("select emp from Employe emp");
Collection emps = query.getResultList();
```

La chaîne de la requête se réfère à l'entité `Employe` et non à la table `Employe` de la base de données. Elle affiche la liste de tous les objets `Employe` associés. L'exécution proprement dite de la requête s'effectue grâce à l'invocation de la méthode `getResultList()`, qui retourne un objet `List` contenant la liste des employés satisfaisant au critère de la requête.

Le code de la méthode `findAllEmployes()` associée retourne un objet `List`, auquel nous appliquons un cast pour le transformer en objet `Collection`, beaucoup plus facile à utiliser :

```
public Collection<Employe> findAllEmployes() {
     Query query = em.createQuery ("select emp from Employe emp") ;
     return (Collection<Employe>) query.getResulList():
}
```

L'exemple de requête suivant est un peu plus paramétré grâce à l'utilisation de l'argument setParameter appliqué à l'objet query pour le passage des arguments de la requête :

```
String queryString =
"SELECT e FROM Employe e "
+ " WHERE e.salaire >= :salaire";
Query query = em.createQuery(queryString);
query.setParameter("salaire", "1000");
List<Employe> liste =
```

```
query.getResultList();
for (Employe e : liste) {
System.out.println(e.getNom());
}
em.close();
emf.close()
```

Blocs try-finally et catch

Pour des raisons évidente de robustesse du code, l'extrait de code précédent devrait être inclus dans un bloc try-finally et pourrait contenir éventuellement des blocs catch pour gérer les exceptions (RuntimeException) générées par les méthodes d'EntityManager.

Suppression d'une entité

La suppression d'une instance de bean entité `Employe` suppose que celle-ci soit « managée », c'est-à-dire qu'elle soit présente dans le contexte de persistance. Cela signifie que l'application à l'origine de la suppression soit déjà chargée ou qu'elle accède préalablement à l'entité par le biais de la méthode `find()`, comme dans l'extrait suivant :

```
public void suppressionEmploye(int employeId) {
Employe emp = em.find(Employe.class, employeId);
       if (emp != null) {
           em.remove(emp);
       }
}
```

La méthode `suppressionEmploye()` permet de vérifier, avant toute suppression de l'instance proprement dite avec la méthode `remove()`, son existence dans le contexte du gestionnaire d'entités.

Mise à jour d'une entité

La mise à jour d'une entité suppose que vous disposiez d'une référence à l'entité managée, référence obtenue *via* l'appel de la méthode `find()`, comme dans l'exemple suivant de mise à jour du salaire de l'instance de l'entité `employé` :

```
public Employe augmenteEmployeSalaire(int employeId, long augmentation) {
Employe emp = em.find(Employe.class, id);
if (emp != null) {
  emp.setSalaire(emp.getSalaire() + augmentation);
}
return emp;
}
```

Voici, mis bout à bout, le code complet de création, lecture, modification et suppression d'une entité `employe`, regroupées dans une seule classe des différentes méthodes CRUD décrites précédemment :

```
package com.webstock.chap10.model;
import java.util.Collection;

import javax.persistence.EntityManager;
import javax.persistence.Query;
```

```
public class EmployeService {

String texteRequete= "SELECT e FROM Employe e";

protected EntityManager em;

    public EmployeService(EntityManager em) {
        this.em = em;
    }

    public Employe createEmploye(int employeid, String nom, String adresse,
    ➥long salaire) {
        Employe emp = new Employe(employeid);
        emp.setNom(nom);
        emp.setAdresse(adresse) ;
        emp.setSalaire(salaire);
        em.persist(emp);
        return emp;
    }

    public void removeEmploye(int employeid) {
        Employe emp = findEmploye(employeid);
        if (emp != null) {
            em.remove(emp);
        }
    }

    public Employe augmenteEmployeSalaire (int employeid, long augmentation) {
        Employe emp = em.find(Employe.class, employeid);
        if (emp != null) {
            emp.setSalaire(emp.getSalaire() + augmentation);
        }
        return emp;
    }

    public Employe findEmploye(int employeid) {
        return em.find(Employe.class, employeid);
    }

    public Collection<Employe> findAllEmployes() {
        Query query = em.createQuery(texteRequete);
        return (Collection<Employe>) query.getResultList();
    }

}
```

Gestion des transactions JTA et non-JTA

Le gestionnaire d'entités offre des méthodes pour commencer (begin), commiter (commit), et défaire (rollback) des transactions pour une utilisation des transactions locale (non-JTA) et non locale (JTA, pour le support des transactions distribuées à travers plusieurs gestionnaires de BD).

Le conteneur EJB fournit des services de support à la gestion transactionnelle disponible par défaut pour les beans session et MDB. En dehors d'un serveur d'applications, une application JTA doit utiliser l'interface javax.persistence.EntityTransaction pour pouvoir

travailler avec des transactions locales à une ressource. Une instance de ce type peut s'obtenir à l'aide de la méthode `getTransaction()` de l'EntityManager.

Les six méthodes de support à la gestion transactionnelle offerte par cette interface sont les suivantes :

```
public interface EntityTransaction {
   public void begin() ;
   public void commit() ;
   public void rollback() ;
   public void setRollbackOnly();
   public void getRollbackOnly();
   public void isActive();
}
```

Exemple :

```
EntityManager em;
...
 try {
 em.getTransaction().begin()
 ...
 em.getTransaction().commit();
 }
 finally {
  em.close();
 }
```

Nous enjoignons le lecteur à approfondir ces aspects important liés à la gestion transactionnelle en examinant les détails de l'interface EntityTransaction dans la documentation du fournisseur JPA.

Méthodes de callback

Nous avons évoqué au chapitre précédent le concept d'invocation des méthodes selon des étapes liées au cycle de vie des beans.

Ce concept est applicable aux beans entité. Les méthodes peuvent être annotées pour indiquer qu'elles seront appelées par le fournisseur de persistance quand une entité passera dans une étape donnée de son cycle de vie. Ces méthodes peuvent appartenir à une classe entité ou une classe « à l'écoute » de ces événements (entité listener). Elles peuvent avoir n'importe quel nom, mais doivent posséder une signature ne prenant aucun paramètre et possédant un type de donnée void.

Les méthodes de type final ou static ne sont pas des types de callback valides. Les exceptions générées par ces méthodes ne peuvent donc pas être interceptées par un gestionnaire d'exceptions. La transaction est en ce cas simplement abandonnée.

Les différents types de méthodes de callback sont les suivantes :

- @PrePersist : quand persist (ou merge) s'est terminée avec succès.
- @PostPersist : après insertion dans la BD.
- @PreRemove : quand remove est appelée.
- @PostRemove : après suppression dans la BD.
- @PreUpdate : avant modification dans la BD.

- @PostUpdate : après modification dans la BD.
- @PostLoad : après lecture des données de la BD pour construire l'entité.

L'exemple qui suit illustre une utilisation possible de ces méthodes pour la mesure du temps de la dernière synchronisation du bean entité `Employe` avec la base de données (un bean étant considéré comme synchronisé avec la base chaque fois que ce dernier est lu ou sauvegardé dans celle-ci) :

```
@Entity
public class Employe {
    @Id
    @Column(name="EMPLOYEID")
    private employeid;
    private String nom;
    private String adresse;
    @Transient private long syncTime;
    // ...
    @PostPersist
    @PostUpdate
    @PostLoad
    private void resetSyncTime() {
       syncTime = System.currentTimeMillis();
    }
    public long getCachedAge() {
       return System.currentTimeMillis() - syncTime;
    }
  // ...
}
```

Invocation à partir d'un client Java et d'un client Web

Le code de gestion de votre entité `Employe` ayant été conçu à l'aide de JPA, vous pouvez développer la couche client d'invocation correspondante selon les deux modes permis par cette API :

- Mode client Java, qui offre un contrôle manuel du gestionnaire d'entités en mode J2SE simple. Dans ce mode, c'est l'application qui gère le cycle de vie du gestionnaire d'entités au lieu du conteneur.
- Mode full Web, avec prise en charge automatisé par le conteneur EJB des services de gestion du contexte et de l'interface EntityManager en utilisant les mécanismes embarqués de gestion des dépendances.

Gestionnaire d'entités géré par l'application (client Java)

Reprenez la classe `EmployeService` contenant le code fonctionnel de la gestion de l'entité `Employe`.

Vous avez juste besoin d'invoquer les méthodes d'obtention et d'initialisation d'une fabrique du gestionnaire d'entités, comme indiqué dans le code complet d'un client Java suivant (notez l'utilisation des méthodes de synchronisation du gestionnaire d'entités à l'aide de `begin()` et `commit()` dans Java SE) :

```
package com.webstock.chap10.client;
```

```
import java.util.Collection;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import com.webstock.chap10.model.Employe;
import com.webstock.chap10.model.EmployeService;

public class EmployeCRUDJava {

    public static void main(String[] args) {
        EntityManagerFactory emf =
                Persistence.createEntityManagerFactory("EmployeService");
        EntityManager em = emf.createEntityManager();
        EmployeService service = new EmployeService(em);

        // creation et persistance de l'employe
        em.getTransaction().begin();
        Employe emp = service.createEmploye(10, "John Doe","Main Street", 12000);
        em.getTransaction().commit();
        System.out.println("Employe sauvegarde : " + emp);

        // Recherche employe
        emp = service.findEmploye(10);
        System.out.println("Trouve " + emp);

        // Recherche tous les employes
        Collection<Employe> emps = service.findAllEmployes();
        for (Employe e : emps)
            System.out.println("Employe trouve : " + e);

        // Mise a jour employe
        em.getTransaction().begin();
        emp = service. augmenteEmployeSalaire (10, 1000);
        em.getTransaction().commit();
        System.out.println("Employe mis a jour:  " + emp);

        // supprime un employe
        em.getTransaction().begin();
        service.removeEmploye(10);
        em.getTransaction().commit();
        System.out.println("Employe 10 supprime");

        // Fermeture du gestionnaire d'entite EM et de l'EntityManagerFactory EMF
        em.close();
        emf.close();
    }
}
```

Gestionnaire d'entités géré par le conteneur (client Web)

La création d'applications dont le gestionnaire d'entités est géré par le conteneur est simple. Le seul prérequis est que l'EntityManagerFactory crée l'instance.

Ce qui différencie les deux approches tient à la manière dont la fabrique est obtenue. Une fois la fabrique obtenue, cette dernière peut être utilisée pour créer le gestionnaire d'entités, lequel peut ensuite être utilisé de la même façon que dans l'approche J2SE.

Pour illustrer ce mode, le code suivant crée une interface `EmployeService` censée décrire les opérations CRUD sur l'entité `Employe` :

```
public interface EmployeService {
   public Employe createEmploye(int id, String name, long salary);
   public void deleteEmploye(int id);
   public Employe updateEmployeSalaire (int id, long newSalaire);
   public Employe findEmploye(int id);
   public Collection<Employe> findAllEmployes();
}
```

Ensuite, vous pouvez utiliser l'annotation @PersistenceContexte pour déclarer une dépendance sur le contexte de persistance et permettre une acquisition automatisée.

L'extrait de code suivant illustre l'utilisation de l'annotation @PersistenceContext pour l'acquisition d'un gestionnaire d'entités à travers le mécanisme d'injection de dépendances offert par le conteneur. L'élément unitName spécifie le nom de l'unité de persistance sur laquelle s'appuie le contexte :

```
@Stateless
public class EmployeeServiceBean implements EmployeService {
@PersistenceContext(unitName="EmployeeService")
EntityManager em;
// ...
}
```

Le code suivant présente l'implémentation du bean session `EmployeService` décrivant les opérations CRUD sur l'entité `Employe` dans un contexte JEE (et non plus J2SE). Mesurez au passage toutes les facilités offertes par les services du conteneur, comme @Persistence-Context, qui permet l'injection automatique du contexte pour acquérir le gestionnaire d'entités, et l'élément unitName spécifiant l'unité de persistance :

```
@Stateless
public class EmployeServiceBean implements EmployeService {
  @PersistenceContext(unitName="EmployeService")
  protected EntityManager em;

  public EntityManager getEntityManager() {
     return em;
  }
  public Employe createEmploye(int employeId, String nom, String adresse, long salaire)
 {
     Employe emp = new Employe(employeid);
     emp.setNom(name);
     emp.setAdresse (adresse);
     emp.setSalaire(salaire);
     getEntityManager().persist(emp);
     return emp;
 }
public void deleteEmploye(int employeId) {
```

```
     Employe emp = findEmploye(employeId);
     if (emp != null) {
        getEntityManager().remove(emp);
     }
}
public Employe updateEmployeSalaire(int employeid, long nouvSalaire) {
     Employe emp = findEmployee(employeid);
     if (emp != null) {
        emp.setSalaire(nouvSalaire);
     }
     return emp;
}

public Employe findEmploye(int employeId) {
      return getEntityManager().find(Employe.class, id);
}

public Collection<Employe> findAllEmployes() {
       Query query = getEntityManager().createQuery("SELECT emp
           FROM Employe emp ");
       return (Collection<Employe>) query.getResultList();
}
}
```

Intégration de la couche de présentation

Il ne vous reste plus qu'à invoquer votre bean session stateless à partir d'une servlet. Les paramètres de la requête déterminent l'action correspondante, qui invoque ensuite la méthode dans le bean « injecté » `EmployeService` :

```
import com.webstock.service.EmployeService;

public class EmployeServlet extends HttpServlet {

    private final String TITRE =
        "Chapitre 10: Employee Service Exemple";

    private final String DESCRIPTION =
        "Gestion des opretaions CRUD sur l'entite employe " ;


    // injecte une reference vers EmployeService
    @EJB EmployeService service;

    public void doPost(HttpServletRequest request, HttpServletResponse response)
    ➥throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        printHtmlHeader(out);

        // process request
        String action = request.getParameter("action");
        if (action == null) {
            // Ne rien faire si aucune action demandee
        } else if (action.equals("Creation")) {
            Employe emp = service.createEmploye(
```

```
                parseInt(request.getParameter("createId")),
                request.getParameter("nom"),
                request.getParameter("adresse");
                parseLong(request.getParameter("salaire")));
            out.println("Employe cree:  " + emp);
        } else if (action.equals("Supression")) {
            String id = request.getParameter("removeId");
            service.removeEmploye(parseInt(employeId));
            out.println("Suppression Employe avec identifiant: " + id);
        } else if (action.equals("Update")) {
            String id = request.getParameter("raiseId");
            Employee emp = service.changeEmployeeSalaire(
                    parseInt(employeid),
                    parseLong(request.getParameter("raise")));
            out.println("employe mise a jour " + emp);
        } else if (action.equals("Find")) {
            Employe emp = service.findEmploye(
                    parseInt(request.getParameter("findId")));
            out.println("Trouve " + emp);
        } else if (action.equals("FindAll")) {
            Collection<Employe> emps = service.findAllEmployes();
            if (emps.isEmpty()) {
                out.println("Pas d'employes trouves ");
            } else {
                out.println("Employes trouves: </br>");
                for (Employe emp : emps) {
                    out.print(emp + "<br/>");
                }
            }
        }

        printHtmlFooter(out);
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response)
    ➥throws ServletException, IOException {
        doPost(request, response);
    }

    private int parseInt(String intString) {
        try {
            return Integer.parseInt(intString);
        } catch (NumberFormatException e) {
            return 0;
        }
    }

    private long parseLong(String longString) {
        try {
            return Long.parseLong(longString);
        } catch (NumberFormatException e) {
            return 0;
        }
    }

    private void printHtmlHeader(PrintWriter out) throws IOException {
        out.println("<body>");
```

```
        out.println("<html>");
        out.println("<head><title>" + TITRE + "</title></head>");
        out.println("<center><h1>" + TITRE + "</h1></center>");
        out.println("<p>" + DESCRIPTION + "</p>");
        out.println("<hr/>");
        out.println("<form action=\"EmployeServlet\" method=\"POST\">");
        // form to create
        out.println("<h3>Creation Employe</h3>");
        out.println("<table><tbody>");
        out.println("<tr><td>employeId:</td><td><input type=\"text\" name=\"createId\"/>
        ➥(int)</td></tr>");
        out.println("<tr><td>Nom:</td><td><input type=\"text\" name=\"nom\"/>(String)
        ➥</td></tr>");
        out.println("<tr><td>Adresse:</td><td><input type=\"text\" name=\"adresse\"/>
        ➥(String)</td></tr>");
        out.println("<tr><td>Salaire:</td><td><input type=\"text\" name=\"salary\"/>
        ➥(long)</td>" +
                    "<td><input name=\"action\" type=\"submit\" value=\"Creation\"/>
                    ➥</td></tr>");
        out.println("</tbody></table>");
        out.println("<hr/>");
        out.println("<h3>Suppresion Employe</h3>");
        out.println("<table><tbody>");
        out.println("<tr><td>Id:</td><td><input type=\"text\" name=\"removeId\"/>
        ➥(int)</td>" +
                    "<td><input name=\"action\" type=\"submit\" value=\"Remove\"/></td>
                     ➥</tr>");
        out.println("</tbody></table>");
        out.println("<hr/>");
        // form to update
        out.println("<h3>Mise a jour employe</h3>");
        out.println("<table><tbody>");
        out.println("<tr><td>Id:</td><td><input type=\"text\" name=\"raiseId\"/>
        ➥(int)</td></tr>");
        out.println("<tr><td>Augmentation:</td><td><input type=\"text\" name=\
        ➥"raise\"/>(long)</td>" +
                    "<td><input name=\"action\" type=\"submit\" value=\"Update\"/>
                    ➥</td></tr>");
        out.println("</tbody></table>");
        out.println("<hr/>");
        // form to find
        out.println("<h3>recherche employe</h3>");
        out.println("<table><tbody>");
        out.println("<tr><td>Id:</td><td><input type=\"text\" name=\"findId\"/>
        ➥(int)</td>" +
                    "<td><input name=\"action\" type=\"submit\" value=\"Find\"/>
                    ➥</td></tr>");
        out.println("</tbody></table>");
        out.println("<hr/>");
        out.println("<h3>recherche des employee</h3>");
        out.println("<input name=\"action\" type=\"submit\" value=\"FindAll\"/>");
        out.println("<hr/>");
    }


    private void printHtmlFooter(PrintWriter out) throws IOException {
        out.println("</html>");
```

```
            out.println("</body>");
            out.close();
        }
    }
```

Pour gérer les dépendances externes de l'application qui peuvent concerner des ressources de type JDBC, file de messages ou bean session, le conteneur EJB3 fournit l'annotation @EJB pour une gestion des dépendances simplifiée.

À l'aide de cette annotation, les références aux composants JEE référencés sont résolus dynamiquement par le conteneur ou au sein du code de l'application lorsque l'instance du composant est créée.

Le code complet de ces exemples est disponible sur la page Web dédiée à l'ouvrage (chapitre 10).

En résumé

Ce chapitre a mis en lumière la puissance et la richesse de l'API JPA, fondement de la spécification EJB3. Vous avez vu notamment comment mettre en œuvre les principales annotations JPA appliquées au mapping des relations inter-entités, des méthodes de call-back appliquées aux entités ainsi que des principales opérations permises sur une entité par le biais de l'interface du gestionnaire d'entités.

Le chapitre suivant présente le projet Dali, qui vise à simplifier le mapping O/R.