11

Mapping JPA avec Dali

Au chapitre précédent, vous avez découvert les concepts fondamentaux de l'API JPA. Extraits de code à l'appui, vous avez vu sa relative facilitée de mise en œuvre en même temps que sa puissance.

Dans ce chapitre, vous allez la mettre en œuvre dans Dali, un sous-projet de Web Tools dont le but est de faciliter la mise en œuvre de cette API.

Les outils de support à l'API JPA sur plate-forme Eclipse ont trouvé leur maturité avec le projet Dali. Issu de la mouvance Eclipse, il complète harmonieusement l'outillage Web Tools en offrant des outils de mapping capables de transformer un type de donnée objet en son pendant relationnel dans la base de données.

Eclipse Web Tools 2.0 intègre complètement le projet Dali, rendant plus facile sa mise en œuvre dans les développements autour de la plate-forme Europa.

Le projet Dali

L'objectif de Dali *(http://www.eclipse.org/dali/)* est de faciliter la mise en œuvre de JPA (Java Persistence API). Lancé en 2005 par Oracle, Dali a rejoint JBoss fin 2005.

La cible du projet est le support de la persistance des EJB 3.0 *via* l'API JPA. Depuis son lancement (la version courante est la version Dali 1.5), Dali supporte l'implémentation de référence JPA TopLink Essentials, contribution d'Oracle.

Ce projet offre en particulier les fonctionnalités suivantes :

- Configuration d'un projet Java pour le support de JPA et des annotations pour l'environnement Java SE ou JEE.
- Support des approches de développement top-down (descendante), bottom-up (ascendante) et meet-in-the-middle (mixte).
- Support simplifié des différentes bases de données *via* les nombreux connecteurs disponibles et les différentes vues et perspectives de manipulation des données.

- Intégration d'outils de mapping en liaison avec des vues dédiées, comme JPA Details et JPA Structure, qui permettent de naviguer dans les champs persistants des entités et de la classe, de changer les types de relation, etc.
- Génération des fichiers de persistance persistence.xml et orm.xml pour le support des métadonnées.

Scénarios de développement et configuration

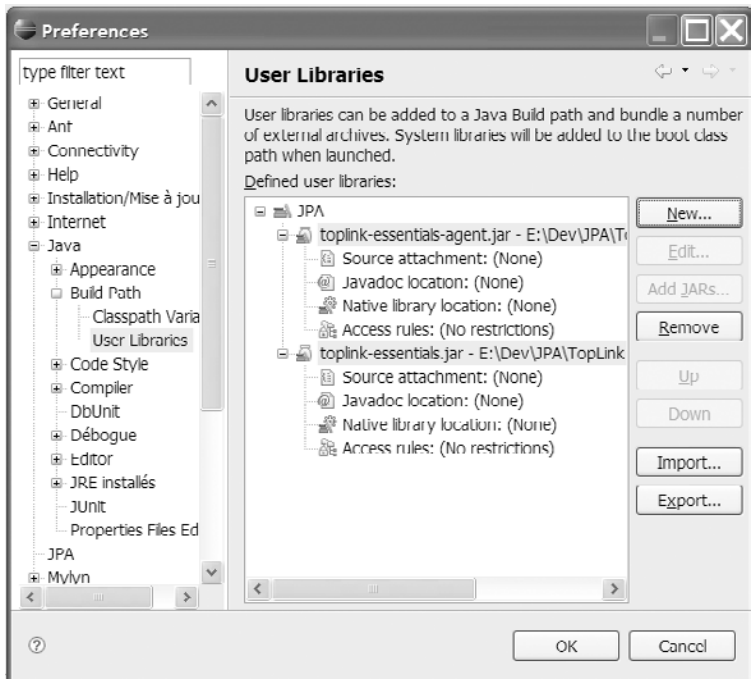
Dali propose différents scénarios de développement autour de JPA et des technologies EJB, Web et POJO. La différence essentielle entre ces scénarios réside dans la configuration de la variable d'environnement classpath.

Si votre application s'exécute au sein d'un conteneur Web ou JEE, elle hérite des bibliothèques JAR du conteneur. Si vous développez une application simple avec JSE5 (sans le support du conteneur), la configuration est entièrement de votre responsabilité. Il suffit pour cela d'ajouter au projet Java sous Eclipse les références aux bibliothèques de référence de TopLink Essentials, l'implémentation de référence de JPA.

La figure 11.1 illustre ce dernier scénario, avec le menu Preferences d'Eclipse configuré pour le support JPA *via* l'option User Libraries.

Figure 11.1

Configuration du support JPA via *le menu Preferences d'Eclipse (1/2)*

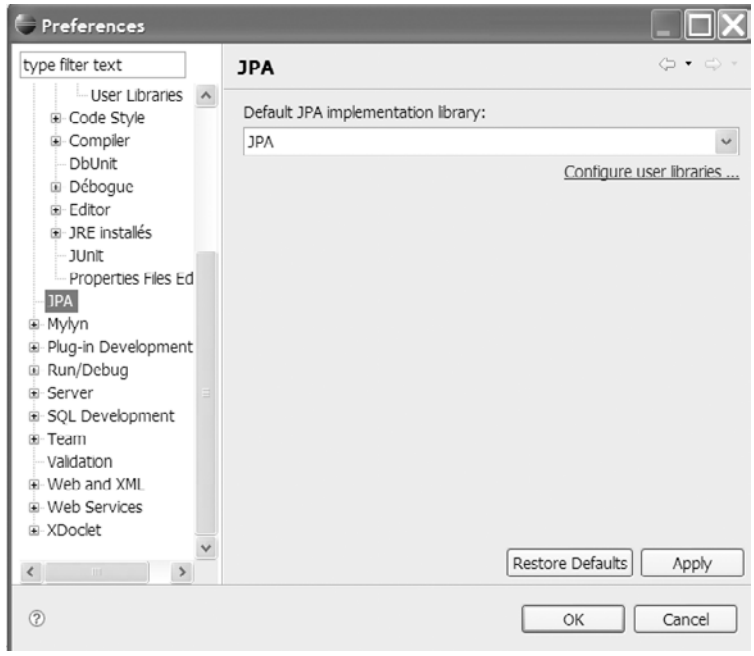


Une fois définie la bibliothèque personnalisée, il suffit de l'associer à l'option JPA en prenant bien soin de cliquer sur Apply pour valider l'environnement, comme l'illustre la figure 11.2.

Vous pouvez à ce stade ajouter à votre projet Java le support JPA en choisissant Properties, Java Build Path, Add Library et User Library dans le menu contextuel du projet puis en sélectionnant la bibliothèque JPA précédemment créée.

Figure 11.2

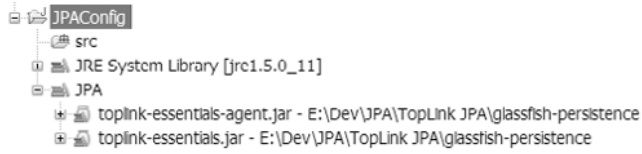
Configuration du support JPA via le menu Preferences d'Eclipse (2/2)



Le chemin d'accès aux bibliothèques JPA se présente comme illustré à la figure 11.3.

Figure 11.3

Arborescence du projet après l'ajout des bibliothèques de référence JPA



Avec votre chemin de compilation correctement positionné, vous pouvez créer et compiler vos entités (ici `Employe`) après import des annotations nécessaires. Le code annoté est compilé dans l'éditeur Eclipse, comme illustré à la figure 11.4.

Figure 11.4

Compilation de l'entité dans Eclipse

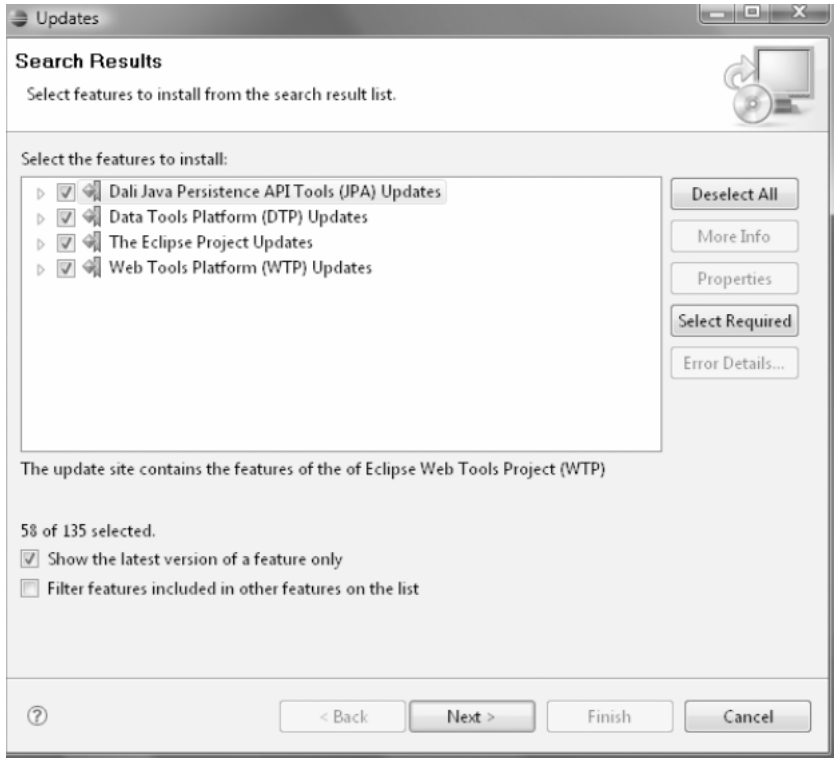
Mise en œuvre de l'API JPA avec Dali

Dali ne requiert qu'une installation d'Europa 3.3 avec le support de l'outillage Web Tools 2.0.

Il importe de mettre à jour la configuration avec les fonctionnalités illustrées à la figure 11.5 *via* le gestionnaire de mise à jour d'Eclipse.

Figure 11.5

Mise à jour de la configuration de l'outillage de mapping Dali



Nous supposons installée et configurée votre base WEBSTOCKDB (voir en annexe pour la configuration de la base HSQLDB livrée en standard avec JBoss 4.2). Notez que Dali permet de régénérer le modèle physique à partir de DDL en entrée.

Nous supposons également configuré (*via* le menu Preferences d'Eclipse) le driver Hypersonic DB, comme illustré à la figure 11.6.

L'implémentation de référence de TopLink JPA est disponible en téléchargement sur le site de l'éditeur Oracle à l'adresse *http://www.oracle.com/technology/products/ias/toplink/jpa/download.html.*

Pour la configurer, il suffit d'ouvrir une sessions DOS et de saisir la commande suivante (selon la version de TopLink JPA, ici la v2, build 41) :

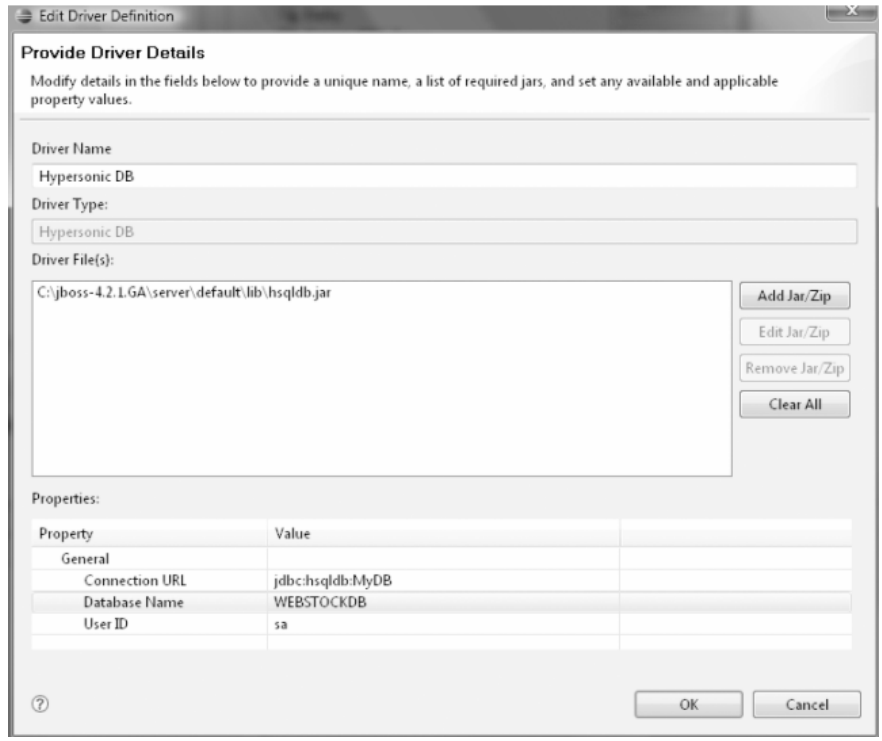
```
java -jar glassfish-persistence-installer-v2-41.jar
```

L'acceptation des conditions de licence a pour effet de décompresser dans le répertoire courant l'ensemble des bibliothèques TopLink requises.

Il faut ensuite démarrer le serveur JBoss pour amorcer le démarrage de la base HSQLDB.

Figure 11.6

Configuration du driver Hypersonic DB



Création du projet JPA et connexion à une source de données

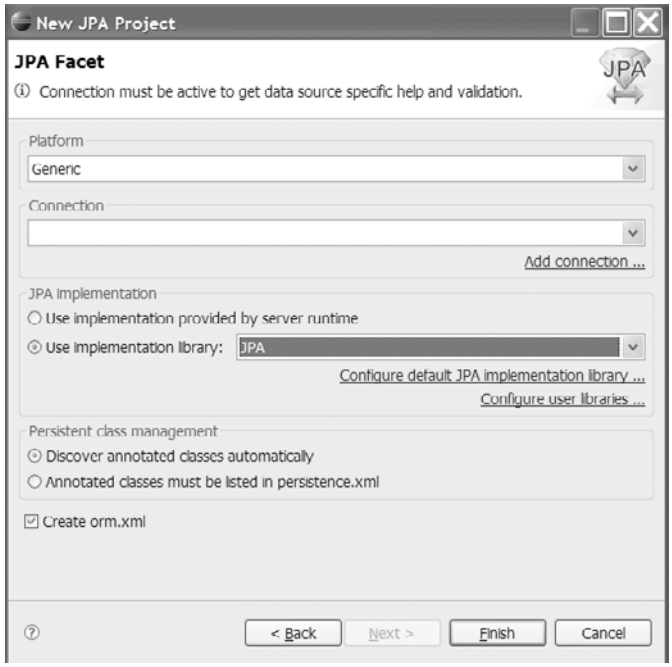
Dans cette première étape, vous allez créer un projet supportant JPA et configurer votre connexion à une source HSQLDB, la base de données livrée et embarquée pour la distribution de JBoss 4.2.

1. Cliquez sur File et New project.
2. Dans l'assistant de création de projet, sélectionnez JPA project.
3. Cliquez sur Next, et saisissez MonProjetJPA. Laissez les autres options inchangées (Target runtime et Utility Configuration en particulier, désignant le serveur d'applications cible et la configuration choisie, soit le support JPA avec JSE 5).

   Si vous disposez déjà d'une connexion existante configurée, vous pouvez la réutiliser en la spécifiant dans le champ Configurations de l'assistant de création de projet JPA.
4. Cliquez sur Next, et laissez les projets Facet proposés par défaut (Java/Java Persistence et Utility Module). Cliquez sur Suivant.
5. L'écran suivant permet de configurer la source de données associée et d'activer la validation. Sélectionnez la bibliothèque JPA comme illustré à la figure 11.7, et cliquez sur Add connection pour configurer la connexion à la base de données en laissant les autres options inchangées.
6. Sélectionnez HSQLDB Connection Profile dans la liste proposée par l'assistant de création de nouvelles connexions (voir figure 11.8), puis cliquez sur Next.

Figure 11.7

Configuration du projet JPA avec Dali

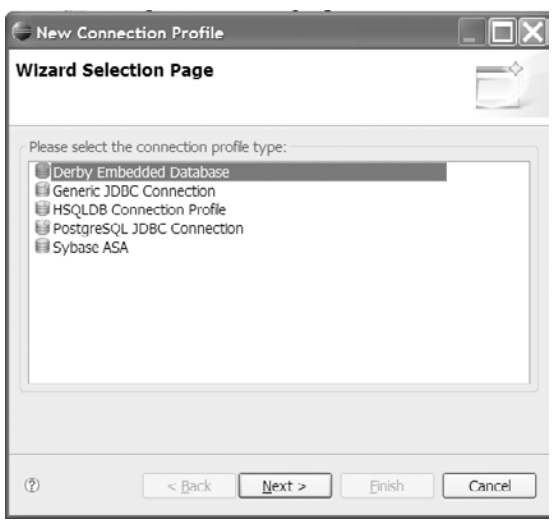


Persistent class management

Dans la zone « Persistent class management », Dali permet d'indiquer si le runtime JPA doit découvrir les entités dynamiquement ou si elles doivent être listées dans le fichier persistence.xml. La case à cocher Create orm.xml permet de spécifier un fichier de mapping XML au lieu des annotations standards.

Figure 11.8

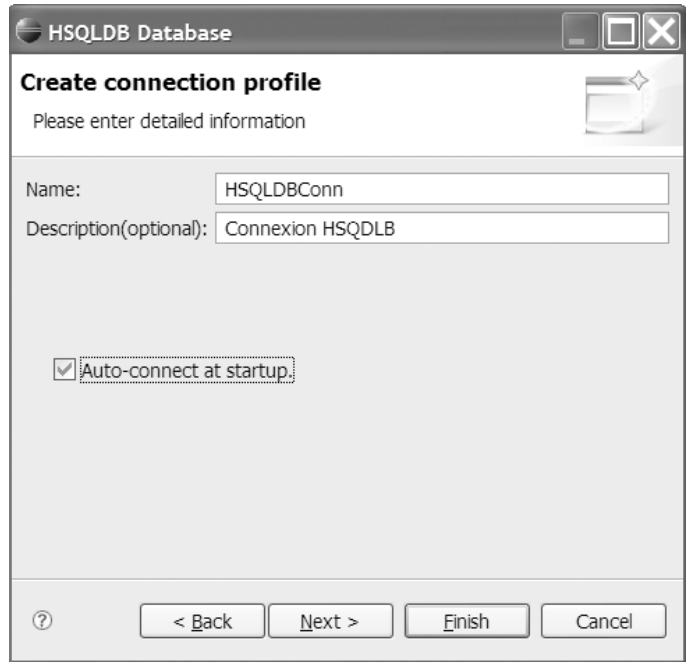
Liste des sources de données supportées par Dali



7. Dans l'assistant de création de profil de connexion, donnez un nom à cette connexion, par exemple HSQLDBConn, et cochez l'option « Auto-connect at startup » pour une connexion automatique à la source de données (voir figure 11.9). Cliquez sur Suivant.

Figure 11.9

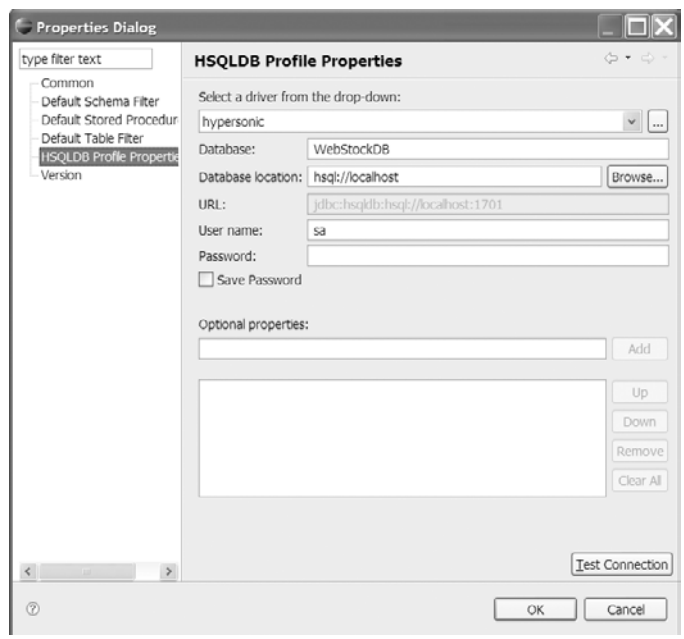
Création d'un profil de connexion (1/2)



8. L'écran suivant permet de configurer la connexion à la base HSQLDB, ici SAMPLE (voir figure 11.10).

Figure 11.10

Création d'un profil de connexion (2/2)

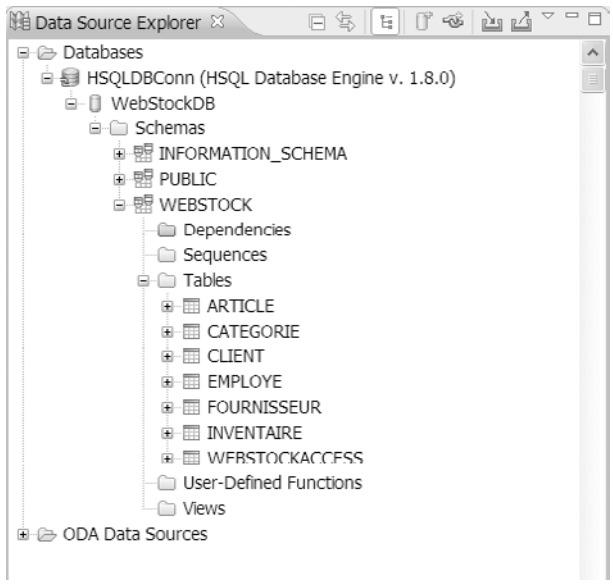


9. Cliquez sur le bouton Test Connection pour valider la connexion, puis cliquez sur OK pour terminer la configuration de la connexion.

Vous pouvez vérifier le succès de la connexion dans la vue Data Source Explorer, qui contient une référence à la source de données Hypersonic définie précédemment et la liste des tables du modèle webstock, comme l'illustre la figure 11.11.

Figure 11.11

Vue Explorer de source de données



Remarquez la création du fichier persistence.xml suivant sous le répertoire src\META-INF :

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

<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
➥xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation
➥="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/
➥persistence_1_0.xsd">

<persistence-unit name="MonProjetJPA-Unit">
</persistence-unit>

<properties>
    <property name="toplink.logging.level" value="FINEST"/>
    <property name="toplink.jdbc.driver"
   value="org.apache.derby.jdbc.ClientDriver"/>
    <property name="toplink.jdbc.url"
   value="jdbc:derby://localhost:1527/WebStockDB;create=true"/>
    <property name="toplink.jdbc.user" value="dali"/>
    <property name="toplink.jdbc.password" value="dali"/>
</properties>
</persistence-unit>

</persistence>
```

Le nom de l'unité de persistance est par défaut celui de votre projet. Ce dernier est associé à la connexion définie plus haut.

Vous avez achevé la définition de l'unité de persistance du projet avec les propriétés associées à la connexion à la base WebStockDB.

Votre projet est prêt à utiliser l'API JPA avec Dali.

Création des entités persistantes du modèle et mapping vers la base

Votre modèle contient trois entités, `Inventaire`, `Article` et `Commande`, que vous allez intégrer dans votre projet et qui vont servir de cadre à la mise en œuvre de l'outillage Dali.

1. Créez un package appelé com.webstock.chap11.model.
2. Cliquez sur New et Class, puis saisissez le nom de la classe `Inventaire`.
3. Répétez cette étape pour `Article` et `Commande`.
4. Ajoutez les champs récapitulés au tableau 11.1 pour chacune des entités.

Tableau 11.1 Structure du modèle physique de la base WebStockDB

Table	Attribut	Type java	Colonne associée	Type de donnée SQL
Inventaire	numInventaire	long	numeroinventaire	double (clé primaire)
	article	article	articleid	integer not null
	quantite	int	quantite	integer not null
	prix	double	prix	decimal (8,2)
	rayon	string	rayon	varchar (2)
	region	string	region	varchar (15)
Article	articleid	int	articleid	integer not null (clé primaire)
	nomArticle	string	nomarticle	varchar (30) not null
	articleCategorie	string	articlecategorieid	varchar (5)
	fournId	integer	fournisseurid	integer
	desc	string	description	varchar (50)
	poids	double	poids	decimal (10,2)
	image_url	string	image_url	varchar (50)
Commande	numCommande	long	numerocommande	integer not nul (clé primaire)
	dateCommande	date	datecommande	date not null
	article	article	articleid	integer not null (clé étrangère vers article)
	owner	string	clientid	integer not null
	quantite	int	quantite	integer not null
	etat	string	etatcommande	varchar (10)
Client	clientId	int	clientid	integer not null (clé primaire)
	userid	int	userid	integer (clé étrangère)
	clientNom	string	clientnom	varchar (15)
	clientPrenom	string	clientprenom	varchar (15)
	adresse	string	adresse	varchar (25)
	telNumero	string	tel	varchar (10)
	comment	string	comment	varchar (25)
	version	int	version	int

5. Pour la classe `Inventaire`, générez les getters et setters pour les champs suivants, sauf pour l'attribut numInventaire :

```
public class Inventaire {

private long numInventaire;
protected Article article;
protected double prix;
protected int quantite;
protected String rayon;
protected String region;
protected int version ;

public double getCost() {
    return cost;
}
public void setCost(double cost) {
    this.cost = cost;
}
public Article getArticle() {
   return article;
}
public void setArticle(Article article) {
    this.article = article;
}
public double getPrix() {
    return prix;
}
public void setPrix(double prix) {
    this.prix = prix;
}
// A completer…
```

6. Pour la classe `Article`, générez les getters et setters pour les champs suivants :

```
public class Article {

protected int articleId;
protected String nomArticle;
protected String articleCategorie;
protected int fournId;
protected String desc;
protected double poids;
protected String image_url;

}
```

7. Pour la classe `Commande`, générez les getters et setters pour les champs suivants :

```
public class Commande {

protected long numCommande;
protected Date dateCommande;
protected List <Article> articles;
protected int quantite;
protected String etat;
 protected Client owner ;

}
```

8. Pour la classe `Client`, générez les getters et setters pour les champs suivants :

```
public class Client {

protected int clientId;

protected int userid;

protected String clientNom;
protected String clientPrenom;
protected String adresse;
protected String telNumero;
protected String comment;
protected int userid;
protected int version ;

protected List <Commande> commandes ;



}
```

Votre modèle est créé. Vous pouvez passer à la création des entités persistantes et les associer aux tables correspondantes de la base.

Mapping des classes avec le modèle physique de données

Vous allez transformer chacune des classes en entité persistante en associant chaque entité du modèle avec sa table de la base :

1. Ouvrez le fichier Article.java dans la vue Package Explorer d'Eclipse. Vous pouvez voir s'afficher la vue JPA Structure correspondante.
2. Sélectionnez la classe `Article`, puis, dans la vue JPA Details, sélectionnez dans la liste déroulante Map As la valeur Entity, comme illustré à la figure 11.12.

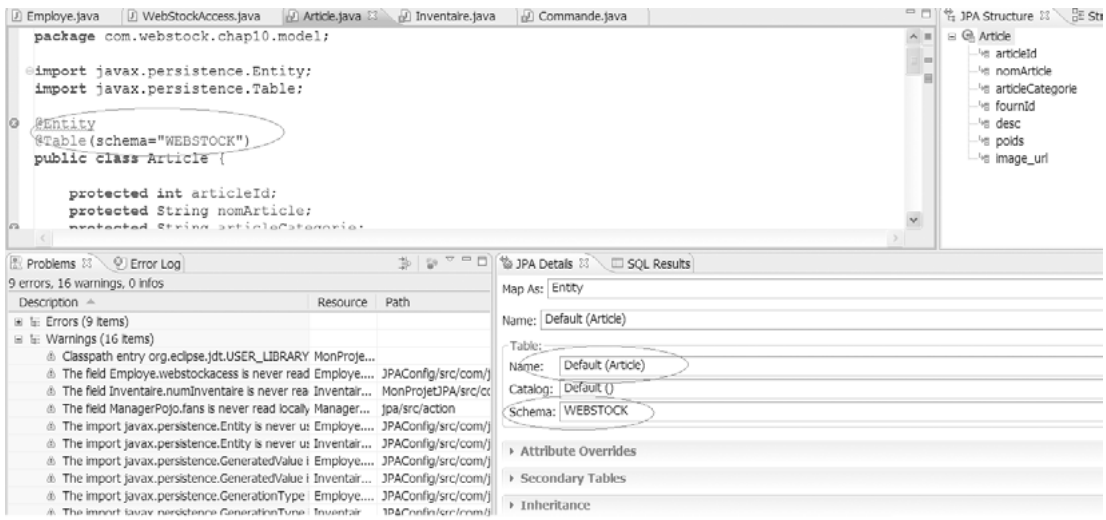


Figure 11.12

Détails de la vue JPA associée à l'entité Article

Schema

Il ne faut pas omettre de préciser le schéma webstock dans l'option Schema de la vue JPA Details afin de mapper exactement la table de la base à l'entité.

Vous pouvez constater qu'une balise @Entity a été automatiquement ajoutée dans le code de la classe `Article`, indiquant qu'elle est devenue une entité persistante.

3. Dans la vue Package Explorer, faites un clic droit sur le fichier persistence.xml, et sélectionnez JPA Tools puis Synchronize Classes. Cela a pour effet de mettre à jour le fichier descripteur avec la nouvelle entité ajoutée (voir balise <class>). Notez que Dali a automatiquement associé l'entité `Article` à la table `Article` correspondante de la base.

   Vous devez à ce stade voir s'afficher des erreurs dans la vue Problems d'Eclipse, du fait que certaines colonnes ne sont pas identiques au nom des colonnes associées dans la base. Vous résoudrez un peu plus loin ce problème de mapping.

4. Répétez les mêmes étapes pour les deux autres entités du modèle webstock extrait (`Inventaire` et `Commande`) et leurs tables associées.

Mapping des champs de la classe

Dans cette étape, vous allez mapper les attributs des classes avec ceux de la table de la base.

Pour cette application, vous utiliserez les types de mappings suivants :

- identifiant de clé
- de base
- One-to-One
- One-to-Many

Création des mappings d'identifiant de clé

Vous devez spécifier l'identifiant de clé primaire associé à chaque entité.

1. Dans l'explorateur de package, ouvrez la classe Article.java.
2. Sélectionnez le champ articleId dans la vue JPA Structure. La vue JPA Details affiche les propriétés du champ.
3. Dans la liste déroulante Map As, sélectionnez Id. Cela a pour effet d'ajouter l'annotation @Id dans le code de la classe `Article`, permettant de définir cette colonne comme clé primaire.
4. Positionnez Insertable et Updatable à false. Cela a pour effet d'ajouter les annotations suivantes :

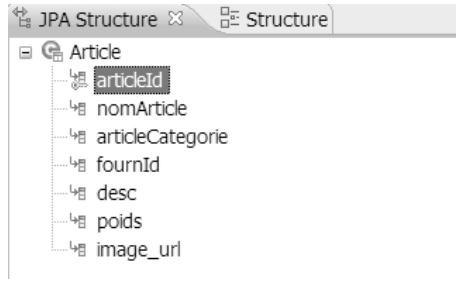
```
@Column(insertable=false, updatable = false)
```

   Ces annotations JPA permettent de spécifier si la colonne doit être utilisée en mode insert ou update.

   Remarquez dans la vue JPA Structure que le champ articleId a été identifié comme clé primaire avec le symbole associé (voir figure 11.13).

Figure 11.13

Structure JPA de l'entité Article



5. Répétez les mêmes étapes pour l'entité `Inventaire` associée à la clé numInventaire pour l'entité et à la colonne numeroInventaire et pour l'entité `Commande` associée à la clé numCommande et à la colonne numeroCommande.

Pour ces deux champs, comme le nom de la colonne dans la base et celui de l'attribut dans l'entité sont différents, il faudra générer l'annotation @Column *via* la liste déroulante Column/Name, comme illustré à la figure11.14.

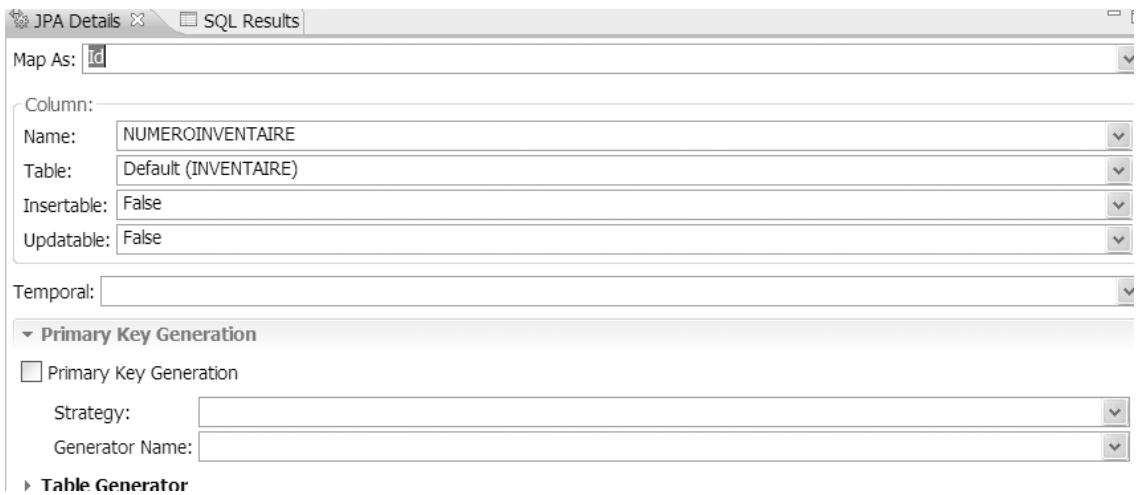


Figure 11.14

Mapping JPA avec Dali

Cela aura pour effet de générer le code suivant (pour l'entité `Commande`) :

```
@Id
@Column(name="NUMEROCOMMANDE", insertable = false, updatable = false)
```

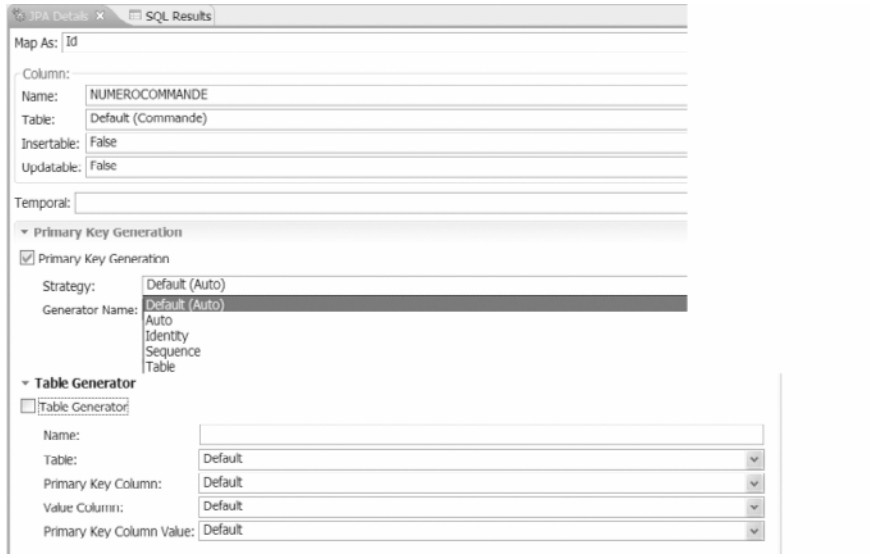
Stratégie de génération des clés primaires

Dali permet de générer le code annoté associé au mode de génération des clés primaires à l'aide de la balise @GeneratedValue.

Cela s'effectue dans la vue JPA Details *via* la configuration des listes déroulantes Strategy et Table Generator, comme illustré à la figure 11.15.

Figure 11.15

Assistant de support à la génération de clés primaires Dali



Le tableau 11.2 récapitule les propriétés de configuration associées à la mise en œuvre d'une stratégie de génération de clés primaires.

Tableau 11.2 Configuration d'une stratégie de génération de clés primaires

Propriété	Description	Valeur par défaut
Primary Key Generation	Définit comment la clé primaire est générée. Ce champ correspond à l'annotation @Generated Value.	
Strategy	– Auto – Sequence : les valeurs d'incrément sont assignées par le biais d'une table sequence. – Identity : les valeurs d'incrément sont assignées par une colonne Identity de la colonne de la base de données. – Table : les valeurs d'incrément sont assignées par une table de la base.	Auto
Generator Name	Nom unique pour la valeur générée	
Table Generator : les champs qui suivent vont définir les tables de la base utilisées pour générer la clé primaire et qui vont être associées à l'annotation @TableGenerator. Ces champs ne s'appliqueront que si la stratégie est de type Table.		
Name	Nom unique du générateur	
Table	Table qui va stocker les valeurs de séquence générées.	
Primary Key Column	Colonne dans la table de génération qui va contenir la clé primaire.	
Value Column	Colonne qui va stocker la valeur générée.	
Primary key Column value	Valeur associée à la colonne clé primaire dans la table servant de génération.	
Sequence Generator : ces champs définissent la séquence spécifique utilisée pour générer la clé primaire et correspondent à l'annotation @SequenceGenerator. Les champs qui suivent ne s'appliquent que lorsque la stratégie est Sequence.		
Name	Nom de la table séquence à utiliser	
Sequence	Nom unique de la séquence	

Pour les trois entités de votre portion de modèle webstock, `Article`, `Commande` et `Client`, vous devez spécifier un mode de stratégie de génération de clés primaires de type Auto (génération automatique), qui est le mode par défaut.

Pour ce faire, procédez comme suit :

1. Cochez l'option Primary Key Generation
2. Spécifiez la stratégie par défaut, Auto.
3. Laissez le champ Generator Name vide.
4. Mappez les attributs restants avec leurs correspondants dans la table de la base (en particulier, pour l'entité `Article`, les attributs articleCategorie avec `ARTICLECATEGORIEID`, fournId avec `FOURNISSEURID` et desc avec `DESCRIPTION`, et, pour l'entité `Commande`, l'attribut etat avec la colonne `ETATCOMMANDE`).

Vous avez achevé la première étape de la configuration. Vous pouvez passer au mapping des types de données de chacun des attributs des entités concernées.

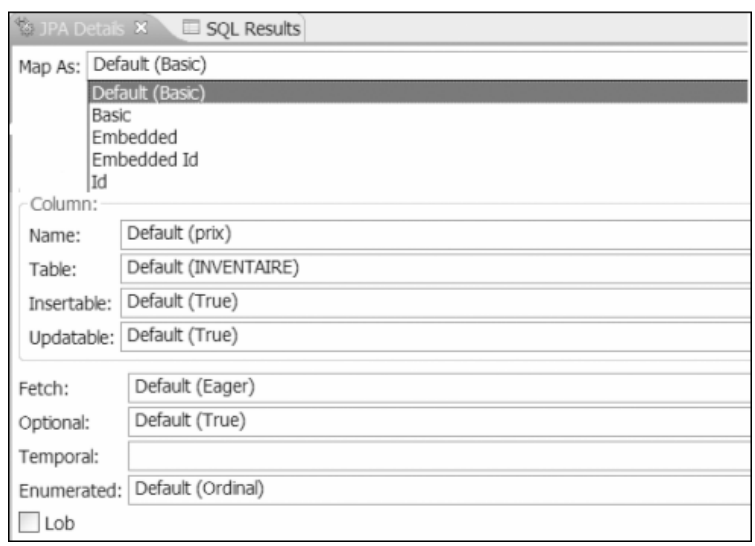
Création des mappings de base

Le mapping de type « basic » permet de mapper des attributs directement avec une colonne de la base. Ce type de mapping ne peut être utilisé qu'avec les types de données primitifs et les « wrappers », par exemple les types string, byte, char, character, date, calendar, time, timestamp, bigDecimal et bigInteger, ainsi qu'avec tout autre type implémentant l'interface Serializable.

Dali propose l'assistant de mapping illustré à la figure 11.16 pour configurer individuellement, selon le type de mapping (champ Map As) choisi, chaque colonne de la table.

Figure 11.16

Assistant de mapping Dali



Le champ Map As propose les valeurs suivantes :

- Default (Basic)
- Basic

- Embedded
- Embedded Id
- Id
- Many to Many
- Many to One
- One to Many
- One to One
- Transient
- Version

Le tableau 11.3 détaille chacune de ces propriétés et fournit des exemples d'application associée à chaque situation avec le code annoté généré. Cela vous sera utile pour l'application des différents types de mapping associés à votre portion de modèle webstock avec Dali.

Tableau 11.3 Propriétés du champ Map As

Propriété	Description	Valeur par défaut	S'applique au mapping suivant

Map As	Définit trois types de mappings :
	– Basic Mapping : correspond à l'annotation @Basic et ne concerne que les types de données Java de base et les wrappers : string, byte, char, character, date, calendar, time, timestamp, bigDecimal, bigInteger ainsi que tout autre type qui implémente l'interface Serializable.
	– Embedded Mapping : à utiliser lorsque vous souhaitez mapper un attribut de l'entité à une instance de type embeddable class, ou classe embarquée. Une classe embarquée est une classe dont les instances sont stockées comme une partie de l'entité mère et qui partage l'identifiant de cette dernière. Cela correspond à l'utilisation d'@Embedded combinée à @AttributeOverride, qui permet de surcharger la colonne d'un objet embarqué pour une entité donnée et sur une propriété particulière. Exemple :
	`@Embeddable`
	`public class EmploymentPeriod {`
	`java.util.Date startDate;`
	`java.util.Date endDate;`
	`...`
	`}`
	`@Entity`
	`public class Employee implements Serializable {`
	`...`
	`@Embedded`
	`@AttributeOverrides({`
	`@AttributeOverride(name="startDate", column=@Column("EMP_START")),`
	`@AttributeOverride(name="endDate", column=@Column("EMP_END"))`
	`)`
	`public EmploymentPeriod getEmploymentPeriod() {`
	`...`
	La classe `EmploymentPeriod` peut être embarquée dans la classe entité `Employee` en utilisant les attributs annotés @AttributeOverrides.
		Basic	

Tableau 11.3 Propriétés du champ Map As *(suite)*

Propriété	Description	Valeur par défaut	S'applique au mapping suivant
Map As *(suite)*	– Embedded Id : permet de spécifier la clé primaire d'un identifiant d'entité de type Embeddable. Ce type de mapping correspond à @EmbeddedId. Constitué d'un ensemble composite de clés primaires appartenant à l'entité, il se rencontre lorsque le mapping est effectué à partir de systèmes dit legacy, dont la clé est constituée de plusieurs colonnes. Il s'applique à des entités de type embarqué. Exemple :		

```
Embeddable
public class EmployeePK implements Serializable
{
    private String name;
    private long id;

    public EmployeePK()
    {
    }

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        this.name = name;
    }

    public long getId()
    {
        return id;
    }
    public void setId(long id)
    {
        this.id = id;
    }

    public int hashCode()
    {
        return (int) name.hashCode() + id;
    }

    public boolean equals(Object obj)
    {
        if (obj == this) return true;
        if (!(obj instanceof EmployeePK)) return false;
        if (obj == null) return false;
        EmployeePK pk = (EmployeePK) obj;
        return pk.id == id && pk.name.equals(name);
    }
}
@Entity
public class Employee implements Serializable
{
    EmployeePK primaryKey;

    public Employee()
    {
    }
```

Tableau 11.3 Propriétés du champ Map As *(suite)*

Propriété	Description	Valeur par défaut	S'applique au mapping suivant

Map As *(suite)*	`@EmbeddedId`
	`public EmployeePK getPrimaryKey()`
	`{`
	`return primaryKey;`
	`}`
	
	`public void setPrimaryKey(EmployeePK pk)`
	`{`
	`primaryKey = pk;`
	`}`
	Les relations de type Many-to-One, Many-to-Many, One-to-Many, One-to-One ont étés décrites dans ce chapitre. Elles génèrent les annotations de support aux relations correspondantes.
		Basic	

Column	Colonnes de la table mappées aux attributs de l'entité (correspond à l'annotation @Column)	Par défaut, les colonnes sont supposées avoir le même nom que les attributs de l'entité.	
Table	Nom de la table de la BD qui contient le nom de la colonne sélectionnée.		

Fetch	Définit la stratégie de chargement des données de la BD :
	– Eager : les données sont chargées au préalable avant leur utilisation.
	– Lazy : les données sont chargées seulement au besoin.
	Exemple:
	`@Entity`
	`public class Employee implements Serializable {`
	`...`
	`@Basic(fetch=LAZY)`
	`protected String getName() {`
	`return name;`
	`}`
		Eager	

Optional	Spécifie si le champ peut être null.	True	

Temporal	Spécifie si le champ mappé appartient à un des types suivants :
	– Date : java.sql.Date
	– Time : java.sql.Time
	– Timestamp : java.sql.Timestamp
	Ce champ correspond à l'annotation @Temporal. Exemple :
	`@Entity`
	`public class Employee {`
	`...`
	`@Temporal(DATE) protected java.util.Date startDate;`
	`...`
	`}`		

Tableau 11.3 Propriétés du champ Map As *(suite)*

Propriété	Description	Valeur par défaut	S'applique au mapping suivant

Enumerated	Spécifie la manière dont les types de données Enum vont persister dans la base :
	– Soit vers une colonne ordinale (en stockant le numéro ordinal de l'enum).
	– Soit vers une colonne de type chaîne de caractères (en stockant la chaîne de caractères représentant l'Enum).
	La représentation de la persistance, par défaut ordinale, peut être surchargée grâce à l'annotation @Enumerated, comme l'illustre la propriété PayScale de l'exemple suivant, dans laquelle l'Enum persiste en tant que string :
	`public enum EmployeeStatus {FULL_TIME, PART_TIME, CONTRACT}`
	`public enum SalaryRate {JUNIOR, SENIOR, MANAGER, EXECUTIVE}`
	`@Entity`
	`public class Employee {`
	`...`
	`public EmployeeStatus getStatus() {`
	`...`
	`}`
	`@Enumerated(STRING)`
	`public SalaryRate getPayScale() {`
	`...`
	`}`		

Lob	Indique que la propriété devrait être persistée dans un Blob ou un Clob selon son type : java.sql.Clob, Character[], char[] et java.lang.String seront persistés dans un Clob. java.sql.Blob, Byte[], byte[] et les types sérialisables seront persistés dans un Blob. Ce champ correspond à l'annotation @Lob. Exemple :
	`@Lob`
	`public String getFullText() {`
	`return fullText;`
	`}`
	`@Lob`
	`public byte[] getFullCode() {`
	`return fullCode;`
	`}`		

Target Entity	Entité persistante sur laquelle l'attribut est mappé.		Tout type de mapping de relation

Mapped By	Champ de la table qui possède la relation. L'association peut être bidirectionnelle, et, dans ce cas, une des extrémités doit être responsable de la mise à jour des colonnes de l'association. C'est là que se mesure l'utilité de cet attribut, qui doit être spécifié avec @OneToOne dans l'entité qui ne définit pas de colonne de jointure, comme dans l'exemple suivant entre l'entité `Employé` et `Badge` :
	`@Entity`
	`public class Employe {`
	`@OneToOne(cascade = CascadeType.ALL)`
	`@JoinColumn(name="badge_fk")`
	`public Badge getBadge() {`
	`...`
	`}`
			Mapping de relations de type mono valuées

Tableau 11.3 Propriétés du champ Map As *(suite)*

Propriété	Description	Valeur par défaut	S'applique au mapping suivant

Mapped By *(suite)*	`@Entity`
	`public class Badge {`
	`  @OneToOne(mappedBy = "badge")`
	`  public Employe getOwner(){`
	`  ...`
	`}`
			Mapping de relations de type mono valuées

Optional	Spécifie si le champ peut être null.	Yes	Mapping de type multi valué
Join Columns	Spécifie une colonne mappée pour joindre une entité. Ce champ correspond à l'attribut @JoinColumn et correspond à la colonne de jointure. Fonctionne si l'attribut « Override Default » est coché.		Mapping de type mono valué
Cascade	Spécifie quels types d'opérations sont propagées à travers l'entité : – All : toutes les opérations. – Persist : effectue en cascade l'opération de persistance (création) sur les entités associées si `persist()` est appelée ou si l'entité est supervisée (par le gestionnaire d'entités). – Merge : effectue en cascade l'opération de fusion sur les entités associées si `merge()` est appelée ou si l'entité est supervisée. – Remove : effectue en cascade l'opération de suppression sur les entités associées si `delete()` est appelée.		Mapping de type multi valuées et mono valuées
Order by	Spécifie l'ordre des objets retournés par une requête : – No Ordering (liste non ordonnée). – Primary Key Order (liste triée par la clé primaire). – Custom Ordering (ordre spécifié).	Primay Key	Tout type de mapping, excepté la relation One-to-One
Inheritance	Une entité peut hériter des propriétés d'autres entités comme dans un modèle objet classique. Dali permet de spécifier une stratégie spécifique pour gérer ce type de relation : – Strategy : Dali propose trois types de stratégies d'héritage : Single Table (par défaut) : toutes les classes dans la hiérarchie sont mappées vers une et une seule table ; Joined table : la racine de l'arborescence est mappée vers une seule table, et toutes les entités filles mappent vers leur propre table ; One Table per class (une table par classe concrète) : chaque classe de la hiérarchie d'héritage est mappée vers une table. Cette stratégie prend en charge les associations de un vers plusieurs bidirectionnelles mais présente un certain nombre d'inconvénients, en particulier pour les relations polymorphes. La stratégie choisie est déclarée au niveau de la classe de l'entité la plus haute dans la hiérarchie en utilisant l'annotation @Inheritance. – Discriminator Column : utilisée pour spécifier le nom de la colonne discriminante si la stratégie d'héritage utilisée est de type Single Table ou Joined table. – Discriminator Type : utilisée pour positionner le type de différenciateur à Char ou Integer. La propriété Discriminator Value doit se confirmer à ce type (par défaut string). – Discriminator Value : spécifie la valeur discriminante utilisée pour différencier une entité dans la hiérarchie d'héritage (string, char, integer). La valeur doit être conforme à la valeur spécifiée dans Discriminator Type. Valeur par défaut string. Ce champs correspond à l'annotation @DiscriminatorValue.	Single Table	

Tableau 11.3 Propriétés du champ Map As *(suite)*

Propriété	Description	Valeur par défaut	S'applique au mapping suivant

Inheritance *(suite)*	Exemple
	`@Entity`
	`@Inheritance(strategy=InheritanceType.SINGLE_TABLE)`
	`@DiscriminatorColumn(`
	`name="Vehiculetype",`
	`discriminatorType=DiscriminatorType.STRING`
	`)`
	`@DiscriminatorValue("Vehicule")`
	`public class Vehicule { ... }`
	`@Entity`
	`@DiscriminatorValue("Passat")`
	`public class Xantia extends Vehicule { ... }`
	Dans cet exemple, la classe parente Vehicule définit la stratégie d'héritage (single table) et la colonne discriminante à l'aide de l'annotation @DiscriminatorColumn. L'annotation @DiscriminatorValue définit la valeur utilisée pour différencier la classe dans la hiérarchie. Le nom de la colonne discriminante est VehiculeType. @Inheritance et @DiscriminatorColumn devraient seulement être définies sur l'entité la plus haute de la hiérarchie.
		Single Table	

Primary Key Join Columns	Définit la clé primaire de la table de la classe fille jointe. Ce champ correspond à @PrimaryKeyJoinColumn.		

Dali identifie automatiquement par défaut les colonnes des entités construites comme mapping par défaut.

Il faut laisser le mapping basic pour chaque attribut des entités `Inventaire`, `Commande` et `Article`. Pour l'attribut dateCommande de l'entité `Commande`, il faut positionner le champ Temporal dans la vue JPA Details à Date. La balise @Temporal (DATE) est automatiquement ajoutée.

Mapping des relations interentités

À ce stade, vous devez modéliser les relations entre les différentes entités de notre modèle.

Relations mono-valuées

Mapping One-to-One

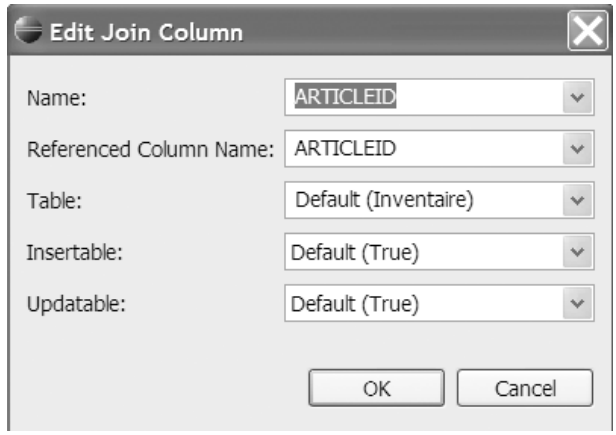
Dans votre modèle, le champ `article` de l'entité `Inventaire` possède une relation One-to-One vers l'entité `Article`, chaque article d'inventaire possédant un et un seul article.

1. Dans la vue Package Explorer, ouvrez la classe Inventaire.java.
2. Dans la même vue, sélectionnez le champ article de l'entité `Inventaire`. La vue JPA Details affiche les propriétés du champ.
3. Dans la liste déroulante Map As de la vue JPA Detail, sélectionnez la propriété One-to-One, et laissez les autres propriétés par défaut.

4. Cochez l'option Overide default, afin de spécifier la colonne de jointure. Dali propose la définition de jointure article_ARTICLEID->ARTICLEID.
5. Cliquez sur Edi pour éditer la jointure. La boîte de dialogue illustrée à la figure 11.17 s'affiche alors.

Figure 11.17

Assistant de définition de jointure Dali



6. Spécifiez `ARTICLEID` dans le champ Name. Dali génère le code suivant :

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

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

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

7. Dans la vue JPA Structure, un symbole s'affiche sur l'attribut `article`, preuve que la relation One-to-One a été prise en compte.
8. Sauvegardez la classe Article.java.

Pour transformer cette relation en One-to-One bidirectionnelle, vous devez ajouter un champ relation à l'entité `Article` afin de pointer en retour sur `Inventaire` :

```
@Entity
public class Article {
//…
@OneToOne (mappedBy="article")
protected Inventaire inventaire;
…
}
```

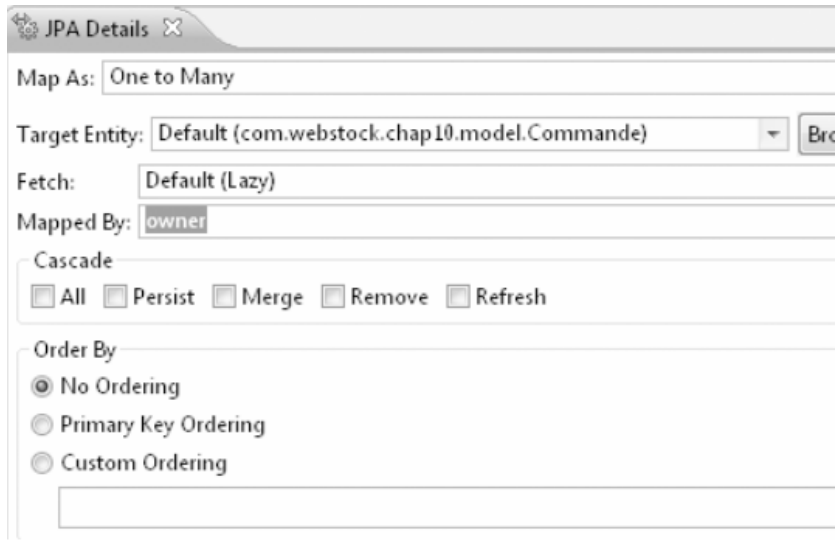
Mapping One-to-Many

Dans votre modèle, l'attribut commandes de l'entité `Client` possède une relation de type One-to-Many vers l'entité `Commande` (chaque client peut avoir plusieurs commandes). L'annotation @OneToMany est ajoutée a un attribut relation de type Collection, où l'entité à l'autre bout de la relation possède ou non un champ relation ou dispose d'une relation monovaluée (Many-to-One) qui pointe en retour sur l'entité.

1. Sélectionnez l'entité `Client` dans la vue Package Explorer.
2. Dans la vue JPA Structure, sélectionnez le champ commandes, puis, dans le champ Map As, choisissez One-to-Many.
3. Dans la liste Mapped By, sélectionnez le champ owner de l'entité `Commande`, comme illustré à la figure 11.18.

Figure 11.18

Mise en œuvre du mapping One-to-Many avec Dali



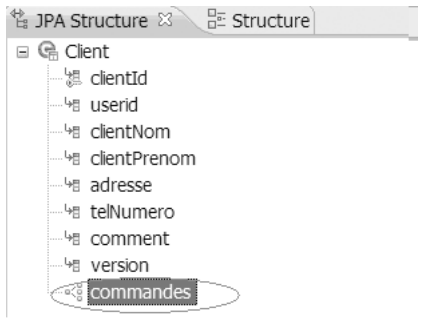
Remarquez le code généré par Dali :

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

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

public Client() {
        commandes = new ArrayList<Commande>();
    }

…

}
```

La vue JPA Structure reflète le type de mapping One-to-Many associé au champ commandes (voir figure 11.19).

Figure 11.19

Vue JPA Structure après application du mapping One-to-Many



Mapping Many-to-One

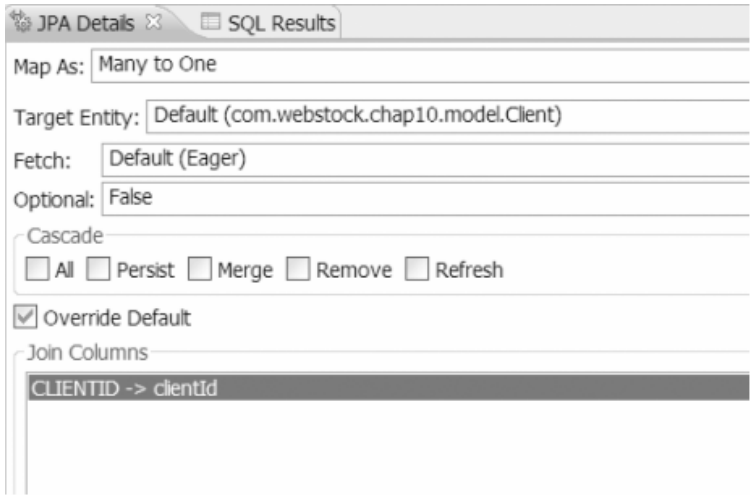
Vous avez défini une relation de type « mapping retour » (back mapping), à partir de la relation One-to-Many définie précédemment. Ce type de mapping sera de type Many-to-One. Vous pouvez ainsi disposer d'une relation bidirectionnelle entre les deux entités.

Ce type d'association sera porté par l'attribut owner de l'entité `Commande`, que l'on pourrait résumer de la façon suivante : « Il existe plusieurs commandes qu'un client peut passer. »

1. Éditez l'entité `Commande` à partir de la vue Package Explorer, si ce n'est déjà fait.
2. Dans la vue JPA Structure, sélectionnez le champ owner de l'entité `Commande`.
3. Dans le champ Map As, sélectionnez Many-to-One, et cochez l'option Override Default.
4. Éditez la colonne de jointure pour qu'elle fasse référence explicitement à la clé étrangère CLIENTID de l'entité `Client` (voir figure 11.20), et laissez les autres champs inchangés.

Figure 11.20

Assistant d'édition de jointure externe Dali



5. Sauvegardez vos modifications.
6. Vérifiez la prise en compte par Dali de la relation ainsi définie en examinant le symbole porté par le champ owner.

Voici le code généré par Dali en fonction de la relation Many-to-One :

```
Public class Commande {
  @Id
  @Column (name= ?NUMEROCOMMANDE?)
  @GeneratedValue
  protected long numCommande;
  @Temporal (DATE)
  protected Date dateCommande;
  protected Article article;
  protected int quaantite;
  @Column (name=?ETATCOMMANDE?)
  protected String etat;
  @ManyToOne(optional=false)
 @JoinColumn(name="CLIENTID", referencedColumnName = "clientId")
 protected Client owner;

…
}
```

En utilisant un type de collection générique, comme `<Commande>`, Dali est capable de déterminer le type d'entité à l'autre bout de la relation, avantage appréciable pour la productivité du développeur JEE. Tout ce qui reste à résoudre en termes de référence de mapping pour la partie @OneToMany de la relation est le nom de la propriété sur cette entité, dans votre cas owner.

Mise en œuvre du mapping de version

Vous avez déjà rencontré l'annotation @Version, qui permet d'ajouter un contrôle de concurrence optimiste à un bean entité.

Dans cette section, vous allez la mettre en œuvre sur les différentes entités qui composent votre modèle. Ce type de stratégie d'optimisation doit être pensé en fonction des contraintes de votre projet et réexaminé ensuite lors des tests de recette et d'exploitation de l'application.

1. Sélectionnez l'entité `Client` dans la vue Package Explorer.
2. Dans la vue JPA Structure, sélectionnez le champ version de l'entité `Commande`. Cela a pour effet d'afficher les détails de la propriété dans la vue JPA Details.
3. Dans le champ Map As de la vue JPA Details, sélectionnez dans la liste déroulante la valeur Version. Le code suivant est ajouté à l'entité `Client` :

```
@Version
protected int version ;
```

Cette modification a pour effet d'ajouter un symbole devant l'attribut Version dans la vue JPA Structure.

Vous avez achevé le mapping des entités de votre modèle. Vous pouvez passer à l'étape de définition des requêtes nommées (named queries) associées aux entités du modèle métier.

Définition des requêtes nommées de l'interface Query

Au chapitre précédent, vous avez abordé les requêtes dynamiques avec Query à l'aide de l'interface EntityManager. Dans ce chapitre, vous allez découvrir les requêtes nommées.

Une requête nommée est définie par l'annotation @NamedQuery, qui définit le nom de la requête. Contrairement aux requêtes dynamiques, conçues à l'aide de l'API Query supportée par JPA, celles-ci ont l'avantage de pouvoir être précompilées lors du déploiement.

Vous pouvez mapper des requêtes JPQL/HQL en utilisant les annotations. @NamedQuery et @NamedQueries peuvent être définies au niveau de la classe ou dans un fichier JPA XML. Leurs définitions sont globales au scope de la Session Factory/Entity Manager Factory.

Une requête nommée est définie par son nom et la chaîne de caractères de la requête réelle précédée de l'attribut query, comme dans l'exemple suivant :

```
@Entity
@NamedQueries({
  @NamedQuery(name = "Commande.findAll",
              query = "select o from Commande o"),
  @NamedQuery(name = "findCommandeByNumCommande",
              query = "select o from Commande o where o.numcommande = :numcommande"),
  @NameQuery (name = "findCommandeByEtat",
              Query="select o from Commande o where o.etat=?4")
})
@Table(name="COMMANDE")
public class Commmande
{
…
}
```

Une entité peut déclarer des instructions JPQL au sein des annotations @NamedQuery pour définir des requêtes « réutilisables ». Les noms @NamedQuery définis doivent être uniques au niveau de l'unité de persistance. La requête findCommndeByNumCommande possède un paramètre, numcommande, mais elle peut également prendre un paramètre indexé, comme la requête findCommandeByEtat précédente.

Liaison des paramètres de la requête

Les requêtes nommées peuvent prendre des paramètres lors de leur invocation ou des paramètres indexés. En supposant que la requête FindClientByUserid définie plus haut soit appelée à partir d'un code client de type bean session, vous aurez :

```
@Stateless
public class GestionCommandeClient implements CommandeMgr
{

  @PersistenceContext(unitName = "MonProjetJPA-Unit")
  private EntityManager em;
…

  /** <code>select o from Commande o</code> */
  public List<Commande> CommandeFindAll() {
    return em.createNamedQuery("Commande.findAll").getResultList();
  }
```

```
  /** <code>select o from Commande o where numcommande = :numcommande</code> */
  public List <Commande> FindByNumCommande(Object numcommande) {
    return em.createNamedQuery("Client.findCommandeByNumeroCommande")
     ➥.setParameter("numcommande",numcommande).getResultList();
  }
}
```

Intégration des entités du modèle logique et mise en œuvre d'un bean client façade

Une fois le modèle logique conçu avec les différentes entités correspondantes selon une approche de conception top-down ou bottom-up (conception partant du modèle logique ou inversement partant du schéma physique sous-jacent), il importe d'invoquer vos entités conçues avec Dali à partir d'un client Java.

Pour ce type de conception, vous vous appuierez sur un classique bean session, qui va servir de façade pour vos entités du modèle. Le bean session `CommandeManager` (voir le code source complet sur la page Web dédiée à l'ouvrage) expose les opérations CRUD sous forme de service, permettant aux clients d'accéder aux entités `Client` et `Commande`.

Vous pourrez implémenter selon le même canevas les opérations de traitement CRUD entre les entités `Inventaire` et `Article`, mais en utilisant cette fois la relation One-to-One. Les services offerts par le bean session façade autorisent la gestion transactionnelle et le maintien de la liaison entre les entités du domaine et la base sous-jacente.

Entités Client et Commande

Rappelons les relations de mapping supportées par les entités `Client` et `Commande` du modèle :

```
@Entity
@NamedQueries({
@NamedQuery(name = "Client.findAll", query = "select o from Client o"),
@NamedQuery(name = "Client.findByUserid", query = "select o from Client o where
➥userid = :userid")
})
public class Client
{

@Idw
@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;


public int getClientId() {
    return clientId;
}

public void setClientId(int clientId) {
    this.clientId = clientId;
}

public int getUserid() {
    return userid;
}

public void setUserid(int userid) {
    this.userid = userid;
}

public String getClientNom() {
    return clientNom;
}

public void setClientNom(String clientNom) {
    this.clientNom = clientNom;
}

public String getClientPrenom() {
    return clientPrenom;
}

public void setClientPrenom(String clientPrenom) {
    this.clientPrenom = clientPrenom;
}

public String getAdresse() {
    return adresse;
}

public void setAdresse(String adresse) {
    this.adresse = adresse;
}

public String getTelNumero() {
    return telNumero;
}

public void setTelNumero(String telNumero) {
    this.telNumero = telNumero;
}

public String getComment() {
    return comment;
}

public void setComment(String comment) {
    this.comment = comment;
}
```

```
public int getVersion() {
    return version;
}

public void setVersion(int version) {
    this.version = version;
}

public Collection <Commande> getCommandes() {
    return commandes;
}

public void setCommandes(Collection <Commande> commandes) {
    this.commandes = commandes;
}


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

@Id
@Column(name="NUMEROCOMMANDE")
@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;


public long getNumCommande() {
    return numCommande;
```

```
}
public void setNumCommande(long numCommande) {
    this.numCommande = numCommande;
}
public Date getDateCommande() {
    return dateCommande;
}
public void setDateCommande(Date dateCommande) {
    this.dateCommande = dateCommande;
}
public Article getArticle() {
    return article;
}
public void setArticle(Article article) {
    this.article = article;
}
```

```
public int getQuantite() {
    return quantite;
}
public void setQuantite(int quantite) {
    this.quantite = quantite;
}
public String getEtat() {
    return etat;
}
public void setEtat(String etat) {
    this.etat = etat;
}
public Client getOwner() {
    return owner;
}
public void setOwner(Client owner) {
    this.owner = owner;
}

}
```

Le bean session CommandeManager

Le bean session `CommandeManager` sert de façade aux beans entité `Client` et `Commande` vus précédemment. Il offre une interface aux opérations du gestionnaire d'entités `persist()`, `merge()` et `remove()` :

```
@Stateless(mappedName="CommandeManager")
public class CommandeManager
  implements CommandeMgr
{
  @PersistenceContext(unitName = "MonProjetJPA-Unit")
  private EntityManager em;

  public CommandeManager() {
  }

  public Object mergeEntity(Object entity) {
    return em.merge(entity);
  }

  public Object persistEntity(Object entity) {
    em.persist(entity);
    return entity;
  }

  /** <code>select o from Client  o</code> */
  public List<Client> queryClientFindAll() {
    return em.createNamedQuery("Client.findAll").getResultList();
  }

  public void removeClient(Client client) {
    customer = em.find(Client.class, client.getClientId());
    em.remove(client);
  }
```

```
  /** <code>select o from Client o</code> */
  public List<Commande> queryCommandeFindAll() {
    return em.createNamedQuery("Commande.findAll").getResultList();
  }

  public void removeCommande(Commande commande) {
    commande = em.find(Commande.class, commande.getNumCommande());
    em.remove(commande);
  }

    /** <code>select o from Client o where userid = :userid</code> */
  public List<Client> queryClientFindByUserid(Object userid) {
    return em.createNamedQuery("client.findByuserid").setParameter("userid",
    userid).getResultList();
  }
}


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

@Remote
public interface CommandeMgr
{
  Object mergeEntity(Object entity);

  Object persistEntity(Object entity);

  List<Client> queryClientFindAll();

  void removeClient(Client client);

  List<Commande> queryCommandeFindAll();

  void removeCommande(Commande commande);

    List<Client> queryClientFindByuserid(Object userid);
}
```

Voici le fichier de persistance associé, adapté à un déploiement final sur la base de données Derby :

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi
➥="http://www.w3.org/2001/XMLSchema-instance" version="1.0" xsi:schemaLocation
➥="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/
➥persistence_1_0.xsd">
<persistence-unit name="default" transaction-type="RESOURCE_LOCAL">
<class>org.eclipse.dali.example.jsf.inventory.model.Inventory</class>
    <class>com.webstock.chap10.Article</class>
         <class> com.webstock.chap10..Inventaire>
    <class> com.webstock.chap10.Commande </class>
         <classe> com.webstock.chap10.Client </class>

    <properties>
        <property name="toplink.logging.level" value="FINEST"/>
        <property name="toplink.jdbc.driver" value
        ➥="org.apache.derby.jdbc.ClientDriver"/>
```

```
        <property name="toplink.jdbc.url" value="jdbc:derby://localhost:1527/WebStockDB;
        ➥create=true"/>
        <property name="toplink.jdbc.user" value="dali"/>
        <property name="toplink.jdbc.password" value="dali"/>
    </properties>
</persistence-unit>

</persistence>
```

Enfin, voici un extrait du code client de test invoqué à partir d'une instance d'une JVM sur les méthodes distantes du bean :

```
public class CommandeClient
{
  public static void main(String [] args) {
    try {
      final Context context = getInitialContext();
      CommandeMgr commandeMgr = (CommandeMgr)context.lookup
      ➥("java:comp/env/ejb/CommandeManager");
      // Invocation des methodes distantes du bean
      commandeMgr.mergeEntity(  entity );
      commandeMgr.persistEntity(  entity );
      System.out.println(commandeMgr.queryClientFindAll(  ) );
      System.out.println(commandeMgr.queryCommandeFindAll(  ) );
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }
…
}
```

Vous pouvez récupérer sur la page Web dédiée à l'ouvrage le code complet de ces extraits et les déployer sur le serveur JBoss 4 en utilisant les scripts Ant de déploiement prévus.

En résumé

Ce chapitre vous a permis de mettre en œuvre l'outillage Dali pour le mapping O/R ainsi que les concepts ayant trait à l'utilisation de l'API JPA.

Le chapitre suivant vous fournira l'occasion de mettre à profit tous ces concepts à la lumière d'une approche de développement centrée sur les modèles et automatisée, avec la solution EclipseUML de l'éditeur Omondo.